View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by RMIT Research Repository

PLANNING IN BDI AGENT SYSTEMS

A thesis submitted in fulfilment of the requirements for

the degree of Doctor of Philosophy

Lavindra Priyalal de Silva
B.Sc. (Hons)

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

September 11, 2009

Declaration

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

Lavindra de Silva,
School of Computer Science and Information Technology,
RMIT University.

To Valérie, for being a silent helper and a patient endurer,
for being a candle flame in dark times . . . for being there.

To my parents, for teaching me endurance, determination,
and patience, which were pivotal in reaching this milestone.

Acknowledgements

It has been a long and sometimes difficult journey, but one which has nonetheless been worthwhile and fruitful due to the guidance and support from many. I would first like to extend my gratitude to my primary supervisor Professor Lin Padgham, and secondary supervisor Dr. Sebastian Sardina. Lin's guidance and insight has been invaluable in shaping this thesis. She has been a great mentor ever since I was an undergraduate student, and she has constantly looked out for and set up new opportunities for me and other students, whether it be networking, scholarships, travel funding, or job interviews. I am deeply grateful to Sebastian for his continuous guidance, for his critical and detailed feedback, and for always taking the time to teach. Almost everything I know today about formal methods is because of Sebastian. I admire both Lin and Sebastian for settling for nothing less than the highest standards.

Besides my PhD supervisors, I was fortunate to have met and worked with a number of other staff members and researchers. I am grateful to Associate Professor James Harland for supervising my project with the DSTO (Defence Science and Technology Organisation), and for giving me many opportunities to teach over the years; Associate Professor Michael Winikoff, my honours supervisor, for introducing me to research, and for his helpful feedback during initial stages of my PhD; Dr. Anthony Dekker, for supervising my DSTO project, and for many interesting discussions over coffee in the months that followed; Dr. Santha Sumanasekara, for the many teaching opportunities over the years; Beti Stojkovska, for much needed help with administrative matters over the years; and all members of the RMIT Agents Group, for their useful feedback on early ideas and practice presentations throughout the course of my honours and PhD degrees. In addition to staff, there were many fellow students who made my time at RMIT memorable. I would like to thank all of them, particularly Gaya (Buddhinath), Toan, Xiang, Simon, Binh, Dhirendra, and David, for the great many chats, and for the unnecessary coffee breaks.

I thank Associate Professor Gal Kaminka, Associate Professor James Thom, and the anonymous reviewer for their useful feedback which helped improve this thesis. I thank RMIT University and DSTO for the opportunity to work on the RMIT-DSTO project on planning, and RMIT also for my honours and PhD scholarships. This thesis was partially funded by the project "Planning and Learning in BDI Agents," which was supported by the Australian Research Council under grant LP0560702, and Agent Oriented Software (AOS). I thank AOS for providing a free license to use JACK for educational purposes.

From thousands of miles away, I have received much support from my relatives in Sri Lanka, Singapore, and Mauritius. For this, I would especially like to thank the Rodrigo family, the (Shantha) de Silva family, Bernard and Nadege Raffa, and Langanee Mendis. In Australia, a handful of friends and family have continuously reminded me that there is a world outside the sometimes apprehensive and melancholic PhD-student-shell. I am deeply grateful to my brother Lasantha in particular, for making life a whole lot easier through the many good times, and also to Nuwan and Ashan for being there through some difficult times. Thanks are also due to Gwenyth, Diane, Benoit, Dronan, Sabrina, Thomas, Umesh, Chandaka, and to the Solomons family, for their help, friendship, and words of encouragement. Although he is no longer with us, the pioneering work on Buddhist-Christian dialogue by my grandfather, the late Rev. Dr. Lynn de Silva, inspired me immensely during the writing of this thesis.

Finally, but most importantly, I would like to express my heartfelt gratitude to my parents Lahan and Peace, for giving my brother and me the freedom to pursue our interests, for their sacrifices over decades to provide us with a good life and education, and for their patience and constant support. Most of all, I thank them for always understanding.

Credits

During the course of this research, a number of papers were published which are based on the work presented in this thesis. They are listed here for reference.

- de Silva, L., Sardina, S., and Padgham, L. First Principles Planning in BDI systems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS-09)*, pages 1105–1112, 2009.
- de Silva, L., and Dekker, A. Planning with Time Limits in BDI Agent Programming Languages. In *Proceedings of Computing: the Australasian Theory Symposium (CATS-07)*, pages 131–139, 2007.
- Sardina, S., de Silva, L., and Padgham, L. Hierarchical planning in BDI agent programming languages: a formal approach. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS-06)*, pages 1001–1008, 2006.
- Dekker, A., and de Silva, L. Investigating organisational structures with networks of planning agents. In *Proceedings of the International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC-06)*, pages 25–30, 2006.
- de Silva, L. and Padgham, L. Planning on demand in BDI systems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-05) Poster Session*, pages 37–40, 2005.
- de Silva, L. and Padgham, L. A comparison of BDI based real-time reasoning and HTN based planning. In *Proceedings of the Australian Joint Conference on Artificial Intelligence (AI-04)*, pages 1167–1173, 2004.

Contents

1 Introduction **1**

2 Background **7**

2.1 The BDI Agent Architecture 7

2.1.1 Practical Reasoning 8

2.1.2 The Abstract BDI Interpreter 10

2.1.3 BDI Agent-Oriented Programming Languages 11

2.1.4 JACK Intelligent Agents 15

2.2 Other Agent Architectures 18

2.3 Automated Planning 23

2.3.1 First Principles Planning 24

2.3.2 Hierarchical Task Network Planning 36

2.4 Combining Agents and Planning 43

2.4.1 First Principles Planning in Agents 44

2.4.2 HTN Planning in Agents 48

3 A HTN Planning Framework for BDI Systems **51**

3.1 Similarities Between the BDI and HTN Approaches 54

3.2 Adding HTN Planning into the CAN BDI Language 61

3.2.1 Presentation of CAN 61

3.2.2 Preliminary Definitions 65

3.2.3 Adding HTN Planning into CAN: the Plan Construct 68

3.2.4 Properties of the Plan Construct 74

4 A First Principles Planning Framework for BDI Systems **83**

4.1 Hybrid Planning 87

4.2 Creating Abstract Planning Operators 90

4.2.1 Assumptions and Preliminary Definitions 92

4.2.2 Preconditions and Postconditions 96

4.2.3 Algorithms 100

4.2.4 An Exploration of Soundness and Completeness 109

4.3 Finding Hybrid-Plans 114

4.4 Validating Hybrid-Plans 117

5 Obtaining a Preferred First Principles Plan **121**

5.1 Preliminary Definitions 125

5.2 MNRMA Hybrid-Plans 126

5.2.1 Non-Redundancy and Minimality 128

5.2.2 Maximal-Abstractness 130

5.3 MNRMA Specialisations of Hybrid-Plans 135

5.4 Preferred Specialisations of Hybrid-Plans 139

5.4.1 Formalisation 141

5.5 Computing Preferred Specialisations 154

6 Implementation **161**

6.1 Comparing the Formal Languages with their Implementations 162

6.1.1 CAN vs. JACK 163

6.1.2 HTN vs. JSHOP 167

6.2 Integrating JSHOP into JACK 171

6.2.1 Mapping JACK to JSHOP 171

6.2.2 Implementation Issues 176

6.3 Integrating Metric-FF into JACK 181

6.4 Improving and Executing Hybrid-Solutions 187

7 Discussion and Conclusion **189**

A Lemmas and Proofs **195**

A.1 Proofs for Chapter 3 195

A.2 Lemmas and Proofs for Chapter 4 195

A.3 Proofs for Chapter 5 199

B Graphs and Trees **205**

List of Tables

3.1 Summary of the mapping from AgentSpeak to HTN 57

4.1 Must literals and may literals of atomic programs and plan-bodies of Figure 4.4. Note that the rightmost column only shows the may literals that are not also must literals. Abbreviations used in the table are as follows: *CA* = *Calibrated*, *HSS* = *HaveSoilSample*, *HMC* = *HaveMoistureContent*, *HPS* = *HaveParticleSize*, *CE* = *ConnectionEstablished*, and *RT* = *ResultsTransmitted*. Each P_i is the plan-body corresponding to plan-rule R_i in the figure. 107

6.1 Summary of the mapping from JACK to JSHOP 177

List of Figures

2.1 A JACK plan-rule for travelling by catching a bus 18

2.2 A simplified planning graph for a Blocks World planning problem. The abbreviation "*Bi*" (e.g., *B*2) is short for "*Blocki*." Nodes are labelled with either the name of an action or a proposition. Nodes with no labels are no-op actions. Solid arrows represent add edges and precondition edges, and dashed arrows represent delete edges. Delete edges have been left out of the second action level for readability. Darker nodes represent the path to a solution for goal atom $On(B3, B2)$. 32

2.3 A simplified representation of a HTN domain $\mathcal{D}$. An arrow below a method indicates that its tasks are ordered from left to right. 39

3.1 A simple Mars Rover agent. An arrow below a plan-rule indicates that its steps are ordered from left to right. The labels adjacent to plan-rules are the resources that they consume: *n*B stands for *n* units of battery, and *n*M stands for *n* units of memory. 52

3.2 CAN's complete set of rules 63

3.3 CANPlan's complete set of rules 75

4.1 The overall framework for first principles planning in BDI systems 84

4.2 A Mars Rover agent. An arrow below a plan-rule indicates that its steps are ordered from left to right. 87

4.3 An inconsistent plan-rule *GoToWorkFridaysPlan* 99

4.4 A slightly modified and extended version of the Mars Rover agent of Figure 4.2. This version has options for navigating and transmitting results, and if the lander is not within range, transmitting involves navigating to the lander and uploading results. 106

5.1 (a) A redundant hybrid-solution h; (b) a hybrid-solution h' with redundancy (actions in bold) removed; and (c) the execution trace of h. 123

5.2 A simple totally-ordered HTN domain. An arrow below a method indicates that its steps are ordered from left to right. The table shows the preconditions and postconditions of the actions. 128

5.3 Refinements for hybrid-solution $[\{(1 : t_0)\}, true]$ (left) and hybrid-solution $[\{(2 : t_1), (3 : t_2)\}, (2 \prec 3)]$ (right) depicted graphically. Dashed rectangles represent constraints on adjacent labelled tasks. 132

5.4 The decomposition tree corresponding to decomposition trace $[\{(1 : t_1)\}, true] \cdot [\{(2 : t_2), (3 : t_3)\}, true] \cdot [\{(4 : a_4), (5 : a_5), (3 : t_3)\}, (4 \prec 5)] \cdot [\{(4 : a_4), (5 : a_5), (6 : a_6), (7 : a_7)\}, (4 \prec 5) \wedge (6 \prec 7)]$. Dotted rectangles stand for primitive tasks/actions, and missing constraints stand for *true*. 140

5.5 A complete decomposition tree $\mathcal{T}$ of task network $d = [\{(1 : t_1), (4 : t_2), (14 : t_6)\}, (1 \prec 4) \wedge (4 \prec 14)]$. Dotted rectangles stand for primitive tasks/actions or empty reductions (node $\langle(17 : \epsilon)\rangle$). 145

5.6 The decomposition tree obtained from the tree in figure 5.5 by projecting on the cut $\{(1 : t_1), (4 : t_2)\}$. 149

6.1 The architecture of our combined framework 163

6.2 JACK plans for the *Navigate* event-goal and its corresponding actions in the Mars Rover agent of Figure 4.2. 165

6.5 Mapping from a JACK plan in the Mars Rover agent of Figure 4.2 to a JSHOP method 176

6.8 Incorporating HTN planning into the Mars Rover agent of Figure 3.1 178

6.9 The JACKPlan specification of plan-rule R_6 in the Mars Rover agent of Figure 4.2 185

List of Algorithms

2.1 BDI-Interpreter() 10

2.2 Linear-Greedy-Justification(σ, C) 29

2.3 Forward-Search(C) 30

4.1 Summarise(Π, Λ) 102

4.2 Summarise-Plan-Body($P, \Pi, \Lambda, \Delta_{in}$) 103

4.3 Summarise-Event($e(\vec{x}), \Pi, \Delta$) 108

5.1 Find-Preferred-Specialisation($h, \mathcal{H}, \mathcal{T}_\tau$) 155

Abstract

Belief-Desire-Intention (BDI) agent systems are a popular approach to developing agents for complex and dynamic environments. These agents rely on context sensitive expansion of plans, acting as they go, and consequently, they do not incorporate a generic mechanism to do any kind of "*look-ahead*" or *offline planning*. This is useful when, for instance, important resources may be consumed by executing steps that are not necessary for a goal; steps are not reversible and may lead to situations in which a goal cannot be solved; and side effects of steps are undesirable if they are not useful for a goal. In this thesis, we incorporate planning techniques into BDI systems.

First, we provide a general mechanism for performing "look-ahead" planning, using Hierarchical Task Network (HTN) planning techniques, so that an agent may guide its selection of plans for the purpose of avoiding negative interactions between them. Unlike past work on adding such planning into BDI agents, which do so only at the implementation level without any precise semantics, we provide a solid theoretical basis for such planning.

Second, we incorporate first principles planning into BDI systems, so that *new* plans may be created for achieving goals. Unlike past work, which focuses on creating low-level plans, losing much of the domain knowledge encoded in BDI agents, we introduce a novel technique where plans are created by respecting and reusing the procedural domain knowledge encoded in such agents; our *abstract* plans can be executed in the standard BDI engine using this knowledge. Furthermore, we recognise an intrinsic tension between striving for abstract plans and, at the same time, ensuring that unnecessary actions, unrelated to the specific goal to be achieved, are avoided. To explore this tension, we characterise the set of "ideal" abstract plans that are non-redundant while maximally abstract, and then develop a more limited but feasible account where an abstract plan is "specialised" into a plan that is non-redundant and as abstract as possible. We present theoretical properties of the planning frameworks, as well as insights into their practical utility.

Chapter 1

Introduction

Recent years have seen an increasing need for delegating to computer software the day to day tasks of humans. Intelligent agent systems are a popular paradigm for building software that exhibit the degree of autonomy and intelligence required to aid humans. By virtue of being autonomous and intelligent by default, such systems eliminate the need for developers to explicitly encode these features into their software applications. This thesis explores adding planning – a particular kind of intelligent reasoning – to a popular and widely used class of intelligent agent systems, namely, Belief-Desire-Intention (BDI) systems.

Autonomy is the ability of an agent to act with little or no intervention from humans. In particular, unlike objects, which are told what to do, intelligent agents have control over their behaviour in that they can decide for themselves whether or not it is appropriate to perform tasks requested from them by external entities (Wooldridge, 2002, p. 25). Tasks are performed in some environment, such as a software environment (e.g., a flight booking agent) or the real world (e.g., a Mars Rover robot). Intelligence arises mainly out of agents being *(i)* autonomous entities; *(ii)* active entities, i.e., having their own threads of execution; and *(iii)* proactive entities, i.e., able to initiate tasks (Wooldridge, 2002, pp. 25, 26). Adding planning to agents adds an additional aspect of intelligence, making for more robust systems.

The BDI agent model has been claimed to provide a more than 300% improvement in efficiency when developing complex software applications (Benfield et al., 2006). While an object-oriented software application is created by the specification of classes, a BDI agent is created by the specification of a set of "recipes," which define how the agent should attempt to solve the

various tasks (e.g., “Book-Flight” or “Schedule-Meeting”) it may encounter during its lifetime. A recipe (e.g., “Book-Emirates-Flight” or “Book-Quantas-Flight”) is a collection of steps for achieving the associated task, combined with a specification of the situations (preconditions) under which the recipe is applicable (e.g., there must be $2000 in the credit card). Steps within a recipe can be basic ones which are directly executable (e.g., invoking a function to make a credit card payment), or more abstract entities – subtasks – that are solved via other recipes. Hence, recipes can be considered hierarchical and partially specified – their details are filled in as execution progresses.

BDI agents are flexible and robust at handling failure. In particular, if it is not possible to solve a task using some associated recipe, an alternative recipe is tried to solve the task, failing only if all associated recipes have failed, or if none are currently applicable. Moreover, BDI agents are well suited for complex and dynamic environments, in particular, those associated with applications that require real-time reasoning and control, such as Unmanned Autonomous Vehicles (UAVs) (Wallis et al., 2002) and Air Traffic Control (Ljungberg and Lucas, 1992). This is because BDI agents multi-task the execution of steps in the real world together with the reasoning about how to solve tasks (e.g., which recipe to choose), thereby lowering the chances of the reasoning being outdated due to changes in the environment by the time execution happens. They also are often able to recover from failure when a wrong decision is taken or something changes in the environment.

One shortcoming of BDI systems is that they do not incorporate a generic mechanism to do any kind of *look-ahead* or *planning* (i.e., hypothetical reasoning). Planning is desirable, or even mandatory in situations where undesired outcomes need to be avoided. In general, planning is useful when *(i)* important resources may be consumed by executing steps that are not necessary for a task; *(ii)* steps are not reversible and may lead to situations from which the task can no longer be solved; *(iii)* executing steps in the real world takes more time than deliberating about the outcome of steps (in a simulated world); and *(iv)* steps have side effects which are undesirable if they are not useful for the task at hand. The three main issues that we address in this thesis for adding planning into BDI agent systems are discussed in the following three sections.

Looking ahead on existing recipes

First, we extend the BDI model so that an agent is able to reason about the consequences of choosing one recipe for solving a task over another. Such reasoning can be useful for guiding the

selection of recipes for the purpose of avoiding negative interactions between them. For example, consider the task of arranging a domestic holiday, which involves the subtasks of booking a (domestic) flight, ground transportation (e.g., airport shuttle) to a hotel, and hotel accommodation. Although the task of booking a flight could be solved by selecting a recipe that books the cheapest available flight, this will turn out to be a bad choice if the cheapest flight lands at a remote airport from where it is an expensive taxi ride to the hotel, and consequently not enough money is left over for accommodation. A better choice would be to book an expensive flight that lands at an airport closer to the hotel, if ground transportation is then cheap, and there is enough money left over for booking accommodation. By reasoning about the consequences of choosing one recipe over another, the agent could guide its execution in order to avoid selecting the recipe that books the cheapest flight.

Look-ahead can be performed on any chosen substructures of recipes; the exact substructures on which it should be performed is determined by the programmer at design time. Look-ahead is not performed on all recipes by default because, although using look-ahead for guiding BDI execution is in some cases more appropriate (e.g., results in a cleaner design) than using preconditions for guiding BDI execution, carefully specified preconditions are often adequate for this purpose.

Planning to find new recipes

The second way in which we incorporate planning into the BDI model is by allowing agents to come up with new recipes on the fly for handling tasks. This is useful when the agent finds itself in a situation where no recipe has been provided to solve a task, but the building blocks for solving the task are available. To find a new recipe, we perform first principles planning, that is, we anticipate the expected outcomes of different steps so as to organise them in a manner that solves the task at hand. To this end, we use the agent's existing repertoire of steps and tasks, i.e., both basic steps (e.g., deleting a file or making a credit card payment) as well as the more complex ones (e.g., a task for going on holiday). In order to anticipate the outcomes of tasks, we compute automatically their intended outcomes and the situations under which they are applicable, using the available library of recipes. Like we do for looking ahead within existing recipes, the programmer can choose the points from which first principles planning should be performed. In addition, such planning could also be done automatically on, for instance, the failure of an important task.

Using tasks for first principles planning, as opposed to using only the basic steps of an agent, is

desirable for two reasons. First, tasks are abstract entities that are well equipped with alternatives (recipes) for handling failure. Second, since a task represents a state of affairs, its associated recipes can be thought of as capturing the *user's intent* (or the user's preferences) on how the associated state of affairs should be brought about. Thus, by using, instead of arbitrary basic steps, some combination of tasks for bringing about a given state of affairs, we are respecting the user's intent on how that state should be brought about, as tasks are eventually solved via associated recipes.

Finding desirable recipes

It is desirable to build recipes using tasks that are as abstract as possible, because such steps are more flexible and robust to failure than those that are less abstract — a higher level of abstraction generally entails a larger collection of alternatives to try if failure occurs. At the same time, however, it is also important to avoid including within recipes tasks that are abstract to the extent that they lead to a mass of unnecessary or redundant basic steps. For example, it is undesirable to have in a newly found recipe an abstract task that involves arranging a holiday for the purpose of booking a flight, when only the flight booking is required. Hence, in this thesis we attempt to find the right balance between formulating recipes that do not lead to redundant basic steps, and keeping steps within recipes as abstract as possible.

Research questions

The research questions we address in this thesis can be summarised as follows.

1. How can we formally describe the process of looking ahead within substructures of existing BDI recipes and guiding recipe selection?

   To this end, we extend the BDI model by incorporating look-ahead capabilities. In particular, we add a look-ahead module into the BDI model, and we provide an operational description of the new system's behaviour.

2. What are the algorithms and data structures for creating new recipes, not already a part of the agent's recipe library, on demand using the agent's existing repertoire of basic and complex steps? In particular, what information is needed from such steps for this purpose, and how can this information be used to create new recipes?

To this end, we define formally what information we need to extract from the existing building blocks of an agent, and we provide algorithms for extracting this information and using it to create new recipes.

3. What are the different formal characterisations of desirability with respect to recipes? Do such characterisations exist that can be realised with computationally feasible algorithms?

   To this end, we define various notions including that of an "ideal" recipe, which is one that does not lead to any redundant steps but is as abstract as possible, and a computationally feasible notion of a "preferred" recipe, for which we provide algorithms and data structures.

4. What are the theoretical properties of the formal frameworks, and how can the frameworks be implemented? What is their practical utility?

   Our frameworks have resulted in significant theoretical and practical benefits to the BDI model. We provide insights into the practical utility of the frameworks by, for instance, highlighting the gaps between the frameworks and their implementations, and showing how some of these gaps can be reduced.

Thesis outline

This thesis is organised as follows. In Chapter 2, we discuss the background material needed to understand this thesis. In Chapter 3, we incorporate look-ahead deliberation into the BDI model. To this end, we make use of a planning technique called Hierarchical Task Network (HTN) planning. We first compare the syntax and semantics of HTN planning with the BDI model, and then show how HTN planning can be incorporated into the BDI model for the purpose of performing look-ahead within BDI recipes. In Chapter 4, we incorporate first principles planning into the BDI model so that new BDI recipes may be obtained. This involves defining formally what information we need to extract from existing recipes to be able to perform first principles planning, providing algorithms and data structures for both automatically extracting this information as well as for obtaining new recipes using this information, and analysing the properties of the formalisms and algorithms. In Chapter 5, we define formally an ideal notion of desirability with respect to recipes, and also two other less than ideal, but still desirable notions of recipes. We provide practical algorithms for one of these notions, and a formal analysis of the different notions and algorithms. In Chapter 6, we discuss the implementation of the formal frameworks and algorithms discussed in

previous chapters, and we give insights into their practical utility. Finally, in Chapter 7, we discuss the contributions and indicate some directions for future work.

Chapter 2

Background

This chapter introduces the background material required to understand this thesis. In particular, it introduces the BDI agent architecture, automated planning, and past work on incorporating automated planning into agent systems. We will also review the different systems we have chosen to implement the research in this thesis, namely, JACK Intelligent Agents, the JSHOP HTN planner, and the Metric-FF first principles planner.

2.1 The BDI Agent Architecture

The BDI (Belief Desire Intention) agent architecture is one particular model of an intelligent agent. Agents developed using this model are called *BDI agents*. To understand what a BDI agent is, we must first define the term *agent*, and moreover, the term *intelligent agent*.

There are various definitions of the term *agent* (Wooldridge and Jennings, 1995; Russell and Norvig, 2002; Müller, 1997; Kaelbling, 1987; Franklin and Graesser, 1997). One widely accepted definition is that suggested by Wooldridge and Jennings (Wooldridge and Jennings, 1995). According to them, an agent is any software system that exhibits the properties of *autonomy* and *situatedness*. Autonomy is the ability of the system to operate with little or no intervention from a human, and situatedness is the ability of the system to inhabit some environment (e.g., a physical or software environment), and to perceive and make changes to it by respectively sensing it and executing actions in it. Not all agents can be considered *intelligent*. For an agent to be classified as an intelligent agent, it needs to, according to Wooldridge and Jennings, exhibit three further properties in addition to the two discussed above. These are described below.

- **Reactivity.** The agent should be able to respond in a timely manner to changes in the environment.
- **Proactiveness.** The agent should be able to behave in a goal-directed manner.
- **Social ability.** The agent should be able to interact with other agents, and possibly humans.

Numerous agent architectures have been built in order to realise some of the properties attributed to intelligent agents. One such architecture is the BDI architecture, which is the focus of this thesis. The BDI architecture is based on Bratman's philosophical theory of *practical reasoning* (Bratman, 1987a; Bratman et al., 1991), which we discuss next.

2.1.1 Practical Reasoning

While theoretical reasoning is focused toward what the agent believes, practical reasoning is focused toward the agent's actions (Wooldridge, 2002, p. 66). In (Bratman, 1987b), Bratman argues that practical reasoning can be thought of as the act of weighing multiple, conflicting considerations for and against conflicting choices, in the light of what the agent believes, desires, values and cares about (Bratman, 1987a, p. 17). More precisely, in practical reasoning, the first step, known as *deliberation*, is to decide *what* state of affairs to bring about from the (possibly conflicting) desires of the agent, and the second step, known as *means-ends reasoning*, is to decide *how* to bring about that state of affairs (Wooldridge, 2002, p. 66). All states of affairs that the agent wants to bring about are called *desires*, the states of affairs that the agent decides to pursue from its set of desires are called *goals*, and the states of affairs that the agent selects and commits to from its set of goals are called *intentions*. Means-ends reasoning is concerned with the adoption of some plan (recipe) of action in order to bring about an "intended" state of affairs.

> For example, consider a person who believes that she has five dollars, and who has the following desires: to go shopping at 1:00, to clean the house at 1:00, and to watch TV at 1:00. After deliberation, she may come to the realisation that she does not have enough money to go shopping, and choose the desire to watch TV instead of the conflicting desire to clean the house. The desire to watch TV is then an intention (and goal) of the person – it is a desire that she has decided to pursue and committed to.

Bratman identifies two important properties related to goals. First, having a goal to bring about some state of affairs, while at the same time having the belief that this state of affairs cannot be achieved, is not rational (Bratman, 1987a, pp. 37, 38). However, having a goal to bring about some state of affairs, while at the same time not having the belief that this state of affairs can be achieved, is rational (Bratman, 1987a, p. 38). Bratman refers to the distinction between these two properties as the *asymmetry thesis*.

Bratman's theory of practical reasoning was formalised by (Cohen and Levesque, 1990) and (Rao and Georgeff, 1991, 1995), so that the relationship between the different mentalistic notions such as beliefs, goals, plans, and intentions may be studied in a formal setting. Both these formalisations give primary importance to intentions, in particular, to the role of intentions in the interplay between the different mentalistic notions. For example, Cohen and Levesque highlights the following as some of the desirable properties of intentions: *(i)* intentions should be maintained for only a finite amount of time; *(ii)* intentions should be dropped if they are believed to be impossible, or believed to have been satisfied; and *(iii)* an agent's actions should be influenced by its intentions, and not go against them.

While the formalisation of Cohen and Levesque describes intentions in terms of beliefs and goals, the work of Rao and Georgeff gives intentions the same level of importance as beliefs and goals. This allows Rao and Georgeff to define different commitment strategies for intentions, and to thereby model different types of BDI agents (Rao and Georgeff, 1991). In particular, they define axioms for capturing three commitment strategies. First, under *blind commitment*, an intention is maintained until the agent believes that she has achieved the intention. Second, under *single-minded commitment*, an intention is maintained until the agent believes that she has achieved the intention, or that it is impossible to achieve. Finally, under *open-minded commitment*, an intention is maintained as long as it is believed to be possible.

The axioms of Rao and Georgeff can be used to capture commitments to *ends* (i.e., future states of affairs) as well as commitments to means (i.e., plans for achieving future states of affairs). By analysing the properties of the different axioms, Rao and Georgeff, like Cohen and Levesque, show formally some desirable properties of rational action. For instance, an agent that is blindly committed to an intention will eventually believe that she has achieved it. Similarly, a single-minded agent will reach the same conclusion only if she continues to believe, until the point at which she believes she has achieved the intention, that the intention is achievable. Finally, an open-

Algorithm 2.1 BDI-Interpreter()

```
1:  Initialise-State()
2:  while true do
3:      options ⇐ Option-Generator(event_queue, B, G, I)
4:      selected_options ⇐ Deliberate(options, B, G, I)
5:      Update-Intentions(selected_options, I)
6:      Execute(I)
7:      Get-New-External-Events()
8:      Drop-Successful-Attitudes(B, G, I)
9:      Drop-Impossible-Attitudes(B, G, I)
10: end while
```

minded agent will eventually believe that she has achieved an intention, provided she maintains it as a goal (and continues to believe it is achievable) until the intention is believed to have been achieved.

2.1.2 The Abstract BDI Interpreter

While the formalisations of Bratman's theory of practical reasoning by Cohen and Levesque and Rao and Georgeff are elegant and have a clear semantics, they are not efficiently computable, and are therefore unsuitable for building practical BDI implementations (Rao and Georgeff, 1995). To address this, an abstract BDI architecture is proposed in (Rao and Georgeff, 1995, 1992), by making certain simplifying assumptions about the theoretical framework, and by modelling beliefs, goals and intentions as data structures (e.g., beliefs as database-like knowledge). The abstract architecture is shown in Algorithm 2.1. Note that variables *event_queue*, B, G and I are all global variables.

The cycle begins with the agent checking its event queue to determine whether there are any pending external event-goals (percepts) from the environment, and generating a set of options (plans) to achieve these event-goals (line 3). From all available plans, the agent then selects a subset to be adopted, and stores these in *selected_options* (line 4). These are then added to the agent's set of *intentions* (line 5), where an intention is a plan that the agent has instantiated and committed to in order to achieve some event-goal. Next, the agent executes a step within any intention in I, which may involve updating the event queue with new internal event-goals (non-primitive actions), or executing a primitive action in the environment (line 6). Finally, any new pending external event-goals are added into the event queue (line 7), and all successful event-goals and

intentions, as well as impossible event-goals and intentions, are removed from the corresponding structures.

This abstract interpreter explores the BDI architecture from more of a practical perspective than the theoretical frameworks discussed earlier. However, because the abstract interpreter leaves out certain details, such as those for the option generator, it is too difficult to investigate the interpreter's theoretical properties and to compare these with the theoretical frameworks discussed. Consequently, the AgentSpeak(L) (Rao, 1996) framework was proposed, which is an operational semantics formalisation (Plotkin, 1981) based on two popular implemented BDI systems called PRS (Procedural Reasoning System) (Georgeff and Ingrand, 1989) and dMARS (d'Inverno et al., 1998).

2.1.3 BDI Agent-Oriented Programming Languages

AgentSpeak(L) belongs to a class of formal languages called BDI agent-oriented programming languages. In this section, we will give an overview of the operational semantics of some of the popular BDI agent-oriented programming languages, namely, AgentSpeak(L) and its variants, 3APL (Hindriks et al., 1999), 2APL (Dastani, 2008), GOAL (Hindriks et al., 2000), CAN (Winikoff et al., 2002), and the operational semantics of Wobcke (Wobcke, 2001). Unless otherwise stated, all these approaches use Plotkin's structural single-step operational semantics (Plotkin, 1981).

AgentSpeak

An AgentSpeak(L) agent is created by the specification of a set of *base beliefs*, representing what the agent believes, and a set of plan-rules called the *plan-library*. The set of base beliefs are encoded as a set of ground atoms. A plan-rule is associated with an event-goal (called an *achievement goal* in AgentSpeak(L)) and contains procedural information on how to handle the corresponding event-goal. This information is made up of entities such as primitive actions, which can be directly executed in the environment, and internal event-goals (non-primitive actions), which require further refinement before their corresponding primitive actions can be executed. The plan-rule can only be used to solve its associated event-goal if the plan-rule's *context condition* is met in the current set of base beliefs, in which case the body of the plan-rule is adopted as an intention. This either amounts to instantiating and adding the body to a set of currently executing intentions (if

the event-goal is external), or updating an existing intention in the set with the new one (if the event-goal is internal).

More formally, an AgentSpeak(L) plan-rule is of the form $+!g : \psi \leftarrow P_1; \ldots ; P_n$, where: $+!g$, called the *triggering event*, indicates that event-goal $!g$ is handled by the plan-rule; ψ is the context condition; and each P_i is either *(i)* an operation $+b$ or $-b$ for respectively adding belief atom b to the agent's set of base beliefs, or for removing b from the agent's set of base beliefs, *(ii)* a primitive action *act* corresponding to any arbitrary operation, *(iii)* an event-goal $!g'$, or *(iv)* a test goal $?g'$, which is used to test whether atom g' holds in the current set of base beliefs.

Because of errors and omissions in the semantics of AgentSpeak(L), many further BDI agent-oriented programming languages were developed, such as (Moreira and Bordini, 2002; d'Inverno and Luck, 1998; Moreira et al., 2003; Bordini et al., 2002; Hindriks et al., 1999; Winikoff et al., 2002; Wobcke, 2001). These either extend AgentSpeak(L), improve it, or are influenced by it in some way. In (d'Inverno and Luck, 1998), a complete syntax and semantics is given for AgentSpeak(L) using the Z specification language (Spivey, 1989). Z is chosen to make more explicit how one could implement AgentSpeak(L) — e.g., to shed some light on the kinds of data structures that could be used to represent an agent's plan-rules. In addition, the authors also identify and address certain mistakes in AgentSpeak(L) and details that were left out, such as a mistake regarding the situations under which the set of intentions can be executed, and details regarding how variables should be bound during execution.

While *Z* is useful as a specification language that sheds some light on implementation specific details, it is not well suited as a language for proving properties of agent systems (Moreira and Bordini, 2002). For this purpose, a complete operational semantics is proposed for AgentSpeak(L) in (Moreira and Bordini, 2002). This semantics incorporates certain features that were left out in AgentSpeak(L), such as semantics for how to execute belief operations $+b$ and $-b$. Further work on AgentSpeak(L) is done in (Hübner et al., 2006), where, like (Winikoff et al., 2002) had done with the CAN language, the authors extend AgentSpeak(L) with the ability to handle failure (i.e., the ability to try alternative plan-rules to achieve an event-goal on the failure of its other plan-rules) and *declarative goals*, which capture more closely some of the desirable properties of goals put forth by Bratman, such as the requirement for goals to be *persistent*, *possible* and *unachieved*. Failure handling is incorporated into AgentSpeak(L) via the triggering event-goal $-!g$, which the programmer can use to specify an alternative plan-rule for handling event-goal $!g$

if a standard plan-rule for handling it has failed; such failure handling plan-rules are of the form $-!g : \psi \leftarrow P_1; \ldots; P_n$. Declarative goals are added to AgentSpeak(L) without modifying the syntax or semantics of AgentSpeak(L). Instead, the authors identify different *patterns* for plan-rules. These patterns are used by the programmer to encode different types of declarative goals.

In (Bordini and Moreira, 2004), the authors prove that the revised version of AgentSpeak(L) proposed in (Moreira and Bordini, 2002) conforms to Bratman's *asymmetry thesis* discussed earlier (recall that this basically required goals to be consistent with beliefs). Finally, in (Bordini et al., 2003), the authors introduce a restricted version of AgentSpeak(L) — called AgentSpeak(F) — and show how one can perform model checking (Clarke et al., 2000) on AgentSpeak(F). In particular, the authors show how guarantees can be obtained for the behaviour of AgentSpeak(F) agents, with respect to specifications expressed as logical formulae.

For example, it could be determined whether a meeting scheduler agent written in AgentSpeak(F), when given as input some person P and her available time slots for the week, is guaranteed to eventually either schedule the meeting for the person and notify her of success, or notify her of the failure to schedule the meeting. More specifically, it could be determined whether condition $(Scheduled(P) \wedge NotifiedOfSuccess(P)) \vee NotifyFailure(P)$ holds at the end of all possible executions of the agent.

3APL and 2APL

Another popular BDI agent-oriented programming language is 3APL (Hindriks et al., 1999). Like extended versions of AgentSpeak(L), 3APL also gives a clean and complete account of the operational semantics of a BDI agent-oriented programming language. In (Hindriks et al., 1998), the authors show that 3APL is more expressive than AgentSpeak(L), i.e., that any AgentSpeak(L) agent can be simulated by a corresponding 3APL agent, but that the converse does not hold. In addition to capturing the functionality of AgentSpeak(L), 3APL allows basic failure handling. Specifically, *failure* plan-rules (called *failure rules* in 3APL) can be written to handle failure, which have higher priority than standard plan-rules that handle event-goals. If both a failure plan-rule and a standard plan-rule are applicable in some situation, the failure plan-rule is tried instead of the standard plan-rule. 3APL also allows the specification of plan-rules which can be used to revise and monitor the agent's existing intentions. For example, a plan-rule can be specified

which replaces all steps within a currently executing intention with the empty plan-body; such plan-rules can be used to drop existing intentions, which is a feature hinted by, but not addressed in AgentSpeak(L). In (van Riemsdijk et al., 2003; Dastani et al., 2003; van Riemsdijk et al., 2005), the authors extend 3APL to handle declarative goals.

2APL (Dastani, 2008) extends 3APL for implementing multi-agent systems. To this end, 2APL includes many new programming constructs, e.g., for implementing external actions and communication actions. Moreover, the semantics of failure rules is different in 2APL in that 2APL failure rules can be applied to revise only failed plan-rules, whereas in 3APL, failure rules can be used to revise any arbitrary plan-rule.

GOAL

While AgentSpeak and 3APL agents do not by default have declarative goals, the main feature of GOAL (Hindriks et al., 2000) agents is that they include declarative goals as part of the language, in the sense that goals describe a state that the agent desires to reach, rather than a task that needs to be performed. Moreover, while AgentSpeak and 3APL agents select predefined plans from a library, GOAL agents only select individual actions from a library. Like other agent-programming languages, actions are selected based on the agent's current mental state. A GOAL agent, then, is the triple $\langle \Pi, \sigma_0, \gamma_0 \rangle$, where Π is a set of actions, σ_0 is a set of initial beliefs, and γ_0 is a set of initial (declarative) goals.

In addition to declarative goals, another key feature in the semantics of GOAL is the inclusion of a default commitment strategy (see Section 2.1.1), namely, the blind commitment strategy. Hence, a GOAL agent drops a goal if and only if it believes that the goal has been achieved. The programmer has the flexibility to use a different strategy if desired.

Operational semantics of Wobcke

In (Wobcke, 2001), Wobcke provides an operational semantics for PRS-like BDI agents, by formalising the abstract BDI interpreter (Algorithm 2.1), rather than an implemented BDI system as done in AgentSpeak(L). The motivation for starting from the interpreter is to keep the semantics as close as possible to the cognitive descriptions of BDI agents in terms of concepts such as beliefs, desires, and goals. The operational semantics of (Wobcke, 2001) allows for convenient constructs within plan-rules such as *if* statements and *while* loops, and moreover, the semantics

makes more explicit the steps of the abstract interpreter. For example, the Deliberate function in line 4 of Algorithm 2.1 is defined as a function that returns an arbitrary plan-rule from those that have the highest priority amongst those in *options*, and the Execute function in line 6 is defined as one that executes one step of an intention and returns the remaining intention along with the state (beliefs) resulting from the execution. The semantics presented also allows basic failure handling, by forcing a failed action to be retried repeatedly until it succeeds.

CAN

In (Winikoff et al., 2002), the CAN (Conceptual Agent Notation) BDI agent-oriented programming language is introduced. CAN can be thought of as a superset of AgentSpeak(L), providing additional features such as failure handling and declarative goals. To accommodate declarative goals, the plan language of CAN includes the construct $\mathsf{Goal}(\phi_s, P, \phi_f)$, which, intuitively, states that (declarative) goal ϕ_s should be achieved using (procedural) plan-body P, failing if ϕ_f becomes true. The operational semantics provided in (Winikoff et al., 2002) for goal-programs captures some of the desired properties of goals mentioned before, such as persistent, possible and unachieved. For example, if the program P within goal-program $\mathsf{Goal}(\phi_s, P, \phi_f)$ has completed execution, but condition ϕ_s is still not true, then P will be re-tried; moreover, if ϕ_s becomes true during the execution of P, the goal-program will succeed immediately. While in variants of AgentSpeak(L) and in different versions of 3APL failure has to be explicitly programmed by the user via, respectively, the use of the triggering event-goal $-!g$ and failure plan-rules, CAN has sophisticated failure handling mechanisms *built into* the framework. This allows CAN agents to try alternative plan-rules to solve an event-goal when steps within a plan-body fail on execution, or when a plan-rule's context condition is not met. Finally, unlike the operational semantics discussed so far, CAN includes semantics for concurrency, allowing steps within intentions to be interleaved. For example, an intention to go out for a movie can be interleaved with the intention to buy bread, by buying bread on the way to (or on the way back from) the movie, as opposed to buying bread before leaving for the movie or after reaching home from the movie.

2.1.4 JACK Intelligent Agents

Much of the operational semantics for BDI agent-oriented programming languages such as (Rao, 1996; Winikoff et al., 2002; Wobcke, 2001; Hübner et al., 2006) were largely influenced by practi-

cal BDI implementations such as PRS (Georgeff and Ingrand, 1989; Ingrand et al., 1992), dMARS (d'Inverno et al., 1998), Jason (Bordini et al., 2007) and JACK (Busetta et al., 1999). Besides these implementations, there are numerous other agent development platforms based on the BDI agent architecture, such as (Machado and Bordini, 2002; Huber, 2001; Pokahr et al., 2003; Bordini et al., 2002; Morley and Myers, 2004). From the available options, we choose JACK Intelligent Agents (Busetta et al., 1999) to implement the algorithms proposed in this thesis.

JACK is a leading edge, commercial BDI agent development platform, used for industrial software development (Jarvis et al., 2003; Wallis et al., 2002). It has similar core functionality to a collection of BDI systems, originating from the PRS (Georgeff and Ingrand, 1989) and dMARS (d'Inverno et al., 1998) systems. JACK is built on top of Java, with the following additions: *(i)* constructs to support BDI concepts such as event-goals and plans-rules (called respectively events and plans in JACK); *(ii)* a compiler that converts JACK syntax into standard Java code; and *(iii)* a kernel to manage things such as concurrent intentions, the default behaviour of an agent in the event that a failure occurs, and the default behaviour of an agent when reacting to external event-goals. Being based on Java, JACK inherits all the advantages of Java, such as object oriented programming, and strong typing, which helps reduce programming errors caused by mis-typing.

A JACK agent is created by identifying its plan-rules, event-goals that are external, event-goals that are internal (i.e., those set by the agent for itself), beliefs, and finally, the elementary Java classes that are required to manipulate the agent's resources. Such Java classes could, for example, have functions for querying an external database, or for performing internal mathematical calculations. Plan-rules are similar to plan-rules in the BDI agent-oriented programming languages discussed so far, in that they have a context condition, and their body consists of a collection of primitive and non-primitive steps. In addition, since JACK is a practical BDI system, JACK plan-rules provide a variety of features not supported in BDI agent-oriented programming languages, such as *meta-plans* for dynamic, programmed choice of the most appropriate plan-rule, and *maintenance conditions* for ensuring that solutions pursued are aborted if the world changes in unspecified ways. The beliefs of a JACK agent are expressed using a database, which allows complex queries to be performed, as well as the encoding of *unknown* facts (e.g., the agent may not know what the weather is outside) in addition to facts that are either *true* or *false*. Encoding unknown beliefs in this way is not possible in any of the BDI agent-oriented programming languages discussed so far because they all follow the *closed world assumption* (Reiter, 1987).

An example of a JACK plan-rule is shown in Figure 2.1. This plan-rule could be one of many plan-rules used to handle an event-goal for travelling from some initial location to a destination. The *#uses data* declarations specify which belief databases this plan-rule accesses – each of these databases are specified in a separate JACK file. The *logical* declarations are used to specify that this plan-rule uses the logical variables called loc, $locBus$, $tktCost$, and $balance$. The *maxWalkingDist* variable is a standard Java integer specifying that the maximum walking distance is 500 metres. The context condition (i.e., the code within the *context()* function) specifies that this plan-rule is only applicable if there is a bus stop within 500 metres from the agent's current location, and if the agent has enough money to pay for the bus ticket. (Note that *as_int()* and *as_string()* are simply used to extract respectively a Java integer instance and Java string instance from the corresponding logical variables.) Observe that the context condition invokes the Java function *distance*, in order to obtain the distance between two locations (the code for this function has been omitted).

The body of the plan-rule specifies that the agent should first walk to the bus stop (line 30), then buy a ticket at the bus stop (line 32), and finally, that the agent should catch the bus (line 36). Walking to the bus stop and catching the bus are non-primitive steps, involving the posting of event-goals *Walk* and *CatchBus*, respectively. Buying a bus ticket, on the other hand, is a primitive step that can be directly executed in the world, by instantiating the standard Java class *BuyBusTicket* with appropriate arguments. Since buying a bus ticket results in the bank balance decreasing (we assume that buying a ticket always succeeds), the belief regarding the bank balance is modified in lines 33 and 34.

The JACK execution engine works in a similar way to the abstract BDI interpreter shown in Algorithm 2.1. The JACK engine repeatedly monitors the event queue for external event-goals from the environment, and converts event-goals into intentions, by selecting and instantiating associated plan-rules whose context conditions are met with respect to the agent's current beliefs. Intentions are executed by giving each of them a programmer controllable time slice. Executing a primitive step may involve, for example, querying an external database for information, or physically moving the wheels of a robot. If an event-goal within an intention fails during execution, an alternative plan-rule for achieving the event-goal is found (if available) and added to the intention

```
1
2  plan CatchBus extends Plan
3  {
4     #handles event Travel travel;
5
6     #posts event Walk walk;
7     #posts event CatchBus catchBus;
8
9     #uses data BusStopLocations busLocs;
10    #uses data BankBalance bankBal;
11    #uses data BusTicketPrices busPrices;
12    #uses data CurrentLocation myLoc;
13
14    private int maxWalkingDist = 500;
15    logical string $loc, $locBus;
16    logical int $tktCost, $balance;
17
18    context()
19    {
20       myLoc.query($loc) &&
21       busLoc.query($locBus) &&
22       maxWalkingDist <= distance($loc.as_string(), $locBus.as_string()) &&
23       busPrices($loc.as_string(), $locBus.as_string(), $tktCost) &&
24       bankBal.query($balance) &&
25       $tktCost.as_int() <= $balance.as_int();
26    }
27
28    body()
29    {
30       @subtask(walk.post($loc.as_string(), $locBus.as_string()));
31
32       @action(new BuyBusTicket($locBus.as_string()));
33       bankBal.remove($balance.as_int());
34       bankBal.add($balance.as_int() − $tktCost.as_int());
35
36       @subtask(catchBus.post($locBus.as_string()));
37    }
38
39    private int distance(int loc1, int loc2)
40    {
41       ...
42    }
43 }
```

Figure 2.1: A JACK plan-rule for travelling by catching a bus

structure. If no such alternative exists, the event-goal fails, causing the plan-rule/intention containing the event-goal to also fail. This causes the agent to look for a new plan-rule to achieve the event-goal handled by the failed plan-rule, creating a form of “backtracking.” Such sophisticated failure handling by “backtracking” is also captured by the CAN language discussed before.

2.2 Other Agent Architectures

There are many agent architectures besides the BDI architecture. In this section, we give an overview of some of the well known agent architectures. In what follows, we will give examples using the popular *Blocks World* (Gupta and Nau, 1992) domain. In this domain, there are blocks

placed on a table with enough space to hold all the blocks, and the arm of a robot for doing tasks such as picking up a block from the table, stacking a block on top of another block, and placing a block on the table.

Logic based agent architectures

Traditional agent architectures, such as Agent Oriented Programming (Shoham, 1993), ConGolog (Lespérance et al., 1995), and Concurrent MetateM (Fisher, 1994; Barringer et al., 1989) use symbolic representations and reasoning in order to define the beliefs and the behaviour of an agent. In these architectures, an agent's beliefs are represented as logical formulae, and an agent's behaviour arises out of deductions performed on the formulae. Specifically, deductions are performed by applying a supplied set of *deduction rules* to the belief formulae, and these deductions lead to predicates corresponding to executable actions.

The ConGolog (Concurrent Golog) architecture is based on the Situation Calculus (Mccarthy and Hayes, 1969), which is an adapted version of the Predicate Calculus to cater for dynamically changing worlds (Lespérance et al., 1995). In ConGolog, the world is represented as a *situation*, which changes only when an agent performs an action in it. The act of performing an action in a situation is represented by the term $do(act, s)$, where act is the action performed on situation s. For example, $Open(Door1, do(open(Door1), s))$ is the situation resulting from performing action $open(Door1)$ in situation s. Actions have preconditions and effects, which are specified using axioms. For example, the precondition axiom $Poss(open(Door1), s) \equiv Closed(Door1, s)$ states that it is only possible to open $Door1$ if it is closed in situation s; and effect axiom $Poss(open(Door1), s) \supset Open(Door1, do(open(Door1), s))$ states that the effect of opening $Door1$ in situation s is that the door is open in the situation resulting from performing the action. It is also possible to define complex actions in ConGolog by making use of standard programming constructs such as procedures, *if* statements, and *while* loops. For example, the following could be a complex action for the Blocks World domain:

proc $unstack_all$
  [**while** [$(\exists block1, block2)\ on(block1, block2)$]
    do $unstack(block1, block2)$
  endWhile]

endProc.

This procedure, named *unstack_all*, repeatedly unstacks blocks until all blocks are on the table. Given the set of all the axioms A of the domain, such as the precondition and effect axioms described above, running the above ConGolog procedure amounts to theorem proving in order to determine the following:

$$A \models (\exists s)Do(unstack_all, S_0, s).$$

In words, running the procedure involves obtaining a binding for the situation s that results from performing procedure *unstack_all* in the initial situation S_0. For example, assuming that the Blocks World domain has only the three blocks $block1$, $block2$ and $block3$, where initially $block1$ is on the table, $block2$ is stacked on top of $block1$, and $block3$ is stacked on top of $block2$, a possible binding for s is $s = do(unstack(block2, block1), do(unstack(block3, block2), S_0))$. This binding states that, first, $block3$ should be unstacked from $block2$, and then $block2$ should be unstacked from $block1$. The sequence of actions encoded in bindings are executed in the real world.

Like ConGolog, the Concurrent MetateM architecture is a logical agent architecture based on theorem proving, which is used for, among other things, the concise specification and prototyping of reactive agent systems (Fisher, 1994). In this architecture, an agent is given a specification in temporal logic capturing the behaviour that the agent should exhibit. The specification is encoded as a set of rules of the form $antecedent \Rightarrow consequent$, where the antecedent is a temporal logic formula relating to the past, and the consequent is a temporal logic formula relating to the present and future. Intuitively, such a rule reads: “if the antecedent holds in the past, do the consequent in the present and/or future.” Hence, if the antecedent of a rule is met with respect to the agent’s history, the rule can ‘fire’, causing the consequent to be executed in the world. If the consequent allows more than one option, for example, to go by bus or to go by train, the agent performs deductive reasoning in order to choose an option that will eventually lead to a successful execution.

Like Concurrent MetateM the Agent Oriented Programming paradigm (Shoham, 1993) uses a temporal language for specifying agents in terms of notions such as beliefs, decisions and capabilities. For example, belief $B^3_a Open(Door1)^5$ means that at time 3, agent a believes that $Door1$ will be open at time 5. Actions in this framework are treated as facts, and thereby considered in-

stantaneous. An agent commits to actions by performing deduction on its belief base after taking into consideration new percepts (messages) from the environment.

Although logic based agent architectures are elegant and have a clear semantics, they are not always practical due to the complexity of theorem proving (Wooldridge, 2002, p. 54). Moreover, such architectures do not model accurately human decision making – humans do not use purely logical techniques when making decisions (Wooldridge, 2002, p. 65). Consequently, the *practical reasoning* model of agency was developed, which we discussed in Section 2.1.1.

Reactive agent architectures

Although less complex than logic based agent architectures, practical reasoning still relies on symbolic representations and reasoning. As a result, the control system of a practical reasoning agent can still be quite complex. In order to discard the need for symbolic reasoning altogether, an entirely different agent architecture was proposed, called (among other things) the *reactive* agent architecture. This architecture does not rely on symbolic reasoning, but instead, it is based on the idea that intelligence emerges from an agent's interaction with its environment. Consequently, agents are provided with simple interacting behaviours, and a mechanism for evolving intelligent behaviour from the interaction between the simpler behaviours.

From the early reactive agent architectures such as PENGI (Agre and Chapman, 1987), the *subsumption architecture* (Brooks, 1986), the *agent network architecture* (Maes, 1989), and *universal plans* (Schoppers, 1987), perhaps the most popular architecture has been the subsumption architecture. In this architecture, agents are built in layers, with the lowest layer having the most generic behaviours, and higher layers having more specific behaviours. For example, when building a mobile robot, the first (lowest) layer may be to avoid contact with objects; the second layer may be to wander around without hitting any objects; the third layer may be to explore the world by observing distant, reachable places and heading toward them; and the fourth layer may be to build a map of the environment, and to plan routes from one place to another (Brooks, 1986). Higher layers have lower priority than the lower layers, which allows lower layers to inhibit higher level behaviours. More concretely, an agent constructed using the subsumption architecture has a set of behaviours of the form $cond \rightarrow act$, where $cond$ is a set of percepts and act is an action. A behaviour b is selected for execution if any percept in its condition is visible in the current environment, and if there is no other behaviour with higher priority than b. The subsumption architecture

has been used successfully for building numerous robots such as Mars Rovers (Steels, 1990) and indoor office robots (Brooks, 1990).

Like the subsumption architecture, agents are constructed in the agent network architecture (Maes, 1989) by the specification of a set of modules with preconditions and effects. However, unlike the subsumption architecture, in addition to the precondition of a module, there is also an "activation level" for determining whether a module can be fired. The higher the activation level of a module, the better are its chances of being fired. The modules are linked to each other to form a network, by connecting together modules whose preconditions and postconditions match. Executing a module within a network may either result in an action being executed in the world, or in the activation level of another module being increased. Like the agent network architecture (Maes, 1989), PENGI (Agre and Chapman, 1987) also uses a network structure for generating complex behaviours. In particular, based on the percepts fed into the network, actions are suggested by the network. Actions in PENGI are generated by simple, low level structures, capturing the routine activities of the agent.

In (Schoppers, 1987), a structure called a *universal plan* is built offline using the most basic behaviours of the agent, such as stacking a block on top of another block. A universal plan is a decision tree encoding, as options and as the root node, the predicates that the agent may come across during execution, and as leaf level nodes, the executable actions. The actions within such a tree are typically generic – i.e., they typically only mention variables. For example, the root node of such a tree could be $On(block1, block2)$, with one option being $Clear(block2)$, another being $\neg Clear(block2)$, and with the $Clear(block2)$ option leading to option $Holding(block1)$ and then to leaf level node (action) $stack(block1, block2)$. The decision tree structure defines the behaviour of the agent at runtime. Specifically, the architecture works by traversing the decision tree until an executable action is reached. This action is then executed, and the process is repeatedly continuously. Such decision trees are called "universal plans" because they are always "applicable," i.e., there is always a path through a decision tree irrespective of the state of the world.

There are many advantages of reactive architectures, such as simplicity, low computational complexity, and robustness against failure (Wooldridge, 2002, p. 96). However, these architectures are not without their shortcomings. According to Wooldridge (Wooldridge, 2002, p. 97), some of the shortcomings of reactive agent architectures are as follows. First, since such architectures make decisions only based on the current state of the world, it is not clear how agents can make

decisions that take into account future states. One such decision could be to travel by bus today in order to have enough money left over to take public transport tomorrow. Second, there is no principled approach for building reactive agents, since such agents are constructed based on experimentation. Finally, in the layered reactive architectures, where complex behaviour arises from interactions between the behaviours of different layers, it is not always straightforward to understand the dynamics of the interactions between these behaviours.

Hybrid agent architectures

In an effort to combine purely reactive architectures with those that use symbolic reasoning, *hybrid agent architectures* emerged. In these architectures, the lowest layer exhibits reactive behaviour, and higher layers exhibit more proactive behaviour. For example, in the InteRRaP (Müller, 1997) hybrid architecture, the lowest layer contains low-level behaviours similar to those in the subsumption architecture, allowing the agent to respond quickly to changes in its environment; the intermediate layer deals with the planning of typical tasks, and has access to a hierarchical BDI-like plan-library which allows the agent to perform more sophisticated, goal-directed reasoning than the lowest layer; and the highest layer allows the agent to reason about and cooperate with other agents. Percepts from the environment arrive at the lowest layer, which either handles them or passes them on to higher layers. The higher layers may make use of the functionalities provided by the lower layers in order to handle the percept. Likewise, in the TouringMachines (Ferguson, 1992) hybrid architecture, the lowest layer is composed of a set of low-level behaviours such as obstacle avoidance, the intermediate layer captures the agent's proactive behaviour with a BDI-like plan-library, and the highest layer captures the agent's social aspects, by modelling itself and other agents, as well as detecting and avoiding conflicts between the goals of multiple agents.

2.3 Automated Planning

In Section 2.1.1, we introduced *means-ends reasoning*, that is, deciding *how* to achieve a goal of the agent. In the BDI architecture, this decision is made by the programmer — the agent is supplied with a library of plan-rules for achieving the different goals that the agent may come across. However, it is also possible for the agent to construct such plan-rules from scratch, when necessary, using its primitive building blocks – actions. *Automated Planning* is the deliberation

process that involves choosing and organising an agent's actions, by anticipating their expected outcomes (Ghallab et al., 2004, p. 1).

Automated planning can be broadly classified into *domain independent planning* (also called *classical planning* and *first principles planning*) and *domain dependent* planning. In domain independent planning, the planner takes as input a description of the initial state of the world, models of all the actions available to the agent, and a goal to achieve – i.e., a state of affairs. The planner then attempts to put the actions into an order such that when they are executed in that order from the initial state, the goal is achieved. Domain dependent planning takes as input additional domain control knowledge specifying which actions should be selected and how they should be ordered at different stages of the planning process. In this way, the planning process is more focused, resulting in plans being found faster in general than first principles planning. However, using such control knowledge also restricts the space of possible plans.

In this section, we discuss first principles planning, and one approach to domain dependent planning called *Hierarchical Task Network* (HTN) planning.

2.3.1 First Principles Planning

The first classical planner was STRIPS (Fikes and Nilsson, 1971). The input for STRIPS is an *initial state* and a *goal state* — which are both specified as sets of facts — and a set of *operators*.[1] An operator has a *precondition* encoding the conditions under which the operator can be used, and a *postcondition* encoding the outcome of applying the operator. We will now be more precise. A *state* is a set of ground atoms, and an *initial state* and a *goal state* are states. An *operator* o is a 4-tuple $\langle name(o), pre(o), del(o), add(o)\rangle$, where *(i)* $name(o) = act(\vec{x})$, the name of the operator, is a symbol followed by a vector of distinct variables such that all free variables in $pre(o), del(o)$, and $add(o)$ also occur in $act(\vec{x})$; and *(ii)* $pre(o), del(o)$ and $add(o)$, called respectively the precondition, *delete-list* and *add-list*, are sets of atoms. The delete-list specifies which atoms should be removed from the state of the world when the operator is applied, and the add-list specifies which atoms should be added to the state of the world when the operator is applied. An operator $\langle name(o), pre(o), del(o), add(o)\rangle$ is sometimes, for convenience, represented as a 3-tuple $\langle name(o), pre(o), post(o)\rangle$, where $post(o) = add(o) \cup \{\neg l \mid l \in del(o)\}$ is a set of literals that combines the add-list and delete-list by treating atoms to be removed/deleted from the belief base as

[1]The following definitions are mainly from (Ghallab et al., 2004).

negative literals. We make use of this definition extensively in Chapter 4. Finally, an *action* is a ground instance of the name of an operator, and a *primitive plan* σ is a sequence of actions.

Given an initial state $\mathcal{I}$, a goal state $\mathcal{G}$ and a set of operators Op, a *classical planning problem* C is the tuple $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. The planner's task is to find a primitive plan that achieves the goal state $\mathcal{G}$ when executed from the initial state $\mathcal{I}$, with respect to the given set of operators Op. Such plans are called *primitive solutions* (or correct primitive plans) for the given planning problem. Before we define the notion of a primitive solution, we will illustrate with an example the notions presented so far.

Suppose we have the following initial state in a Blocks World domain:

$\{OnTable(Block1), On(Block2, Block1), Clear(Block2), ArmEmpty\}$.

This initial state specifies that $Block1$ is on the table, $Block2$ is on top of $Block1$, $Block2$ is clear (i.e., there is no other block on top of it), and that the robot's arm is empty. An operator for picking up any block $block1$ from the table could be the following:

$\langle pickup(block1),$
$\quad \{Clear(block1), OnTable(block1), ArmEmpty\},$
$\quad \{OnTable(block1), ArmEmpty\},$
$\quad \{Holding(block1)\}$
$\rangle$.

The operator states that $block1$ can be picked up if there are no blocks on top of it, if it is on the table, and if the robot's arm is empty. The delete-list specifies that $block1$ is no longer on the table, and that the arm is no longer empty. Finally, the add-list specifies the arm is holding $block1$.

Similarly, the following $unstack(block1, block2)$ operator is used for unstacking a block $block1$ that is on top of some other block $block2$:

$\langle unstack(block1, block2),$

$\{On(block1, block2), Clear(block1), ArmEmpty\}$,

$\{On(block1, block2), ArmEmpty\}$,

$\{Holding(block1), Clear(block2)\}$

$\rangle$.

This operator states that to unstack a block *block*1 from some other block *block*2, *block*1 must be on top of *block*2, *block*1 must be clear, and the robot's arm has to be empty. The delete-list specifies that *block*1 is no longer on *block*2 and that the arm is no longer empty. Finally, the add-list states that the arm is holding *block*1 and that *block*2 is now clear. Other operators such as $putdown(block1)$ and $stack(block1, block2)$ can be specified in a similar manner.

When an action is applied to a state, the atoms in its delete-list are removed from the state, and the atoms in its add-list are added to the state. For example, the result of applying action $unstack(Block2, Block1)$ to state $\{OnTable(Block1), On(Block2, Block1), Clear(Block2), ArmEmpty\}$ is the state $\{OnTable(Block1), Clear(Block1), Clear(Block2), Holding(Block2)\}$. Then, formally, given a set of operators *Op*, a state $\mathcal{S}$, and an action *act*, the *result* of applying *act* to $\mathcal{S}$ relative to *Op*, denoted $Res(act, \mathcal{S}, Op)$, is defined as follows (recall *act* is ground):

$$Res(act, \mathcal{S}, Op) = \begin{cases} (\mathcal{S} \setminus del(o)\theta) \cup add(o)\theta & \text{if } o \in Op \text{ and } act = name(o)\theta \text{ and } \mathcal{S} \models pre(o)\theta; \\ undefined & \text{otherwise.} \end{cases}$$

Similarly, we can define the result of applying a sequence of actions to a state as follows. Given a set of operators *Op*, a state $\mathcal{S}$ and a sequence of actions $act_1 \cdot \ldots \cdot act_n$, the *result* of applying the *sequence* $act_1 \cdot \ldots \cdot act_n$ to $\mathcal{S}$ relative to *Op*, denoted $Res^*(act_1 \cdot \ldots \cdot act_n, \mathcal{S}, Op)$, is defined inductively as follows:

$$Res^*(act_1 \cdot \ldots \cdot act_n, \mathcal{S}, Op) = \begin{cases} Res(act_1, \mathcal{S}, Op) & \text{if } n = 1; \\ Res^*(act_2 \cdot \ldots \cdot act_n, Res(act_1, \mathcal{S}, Op), Op) & \text{if } n > 1; \\ \mathcal{S} & \text{otherwise.} \end{cases}$$

Intuitively, Res^* states that the result of applying a sequence of actions to a state $\mathcal{S}_0$ is the result of applying the first action of the sequence to $\mathcal{S}_0$ to obtain state $\mathcal{S}_1$, followed by the result of

applying the second action of the sequence to obtain state $\mathcal{S}_2$, and so on, until state $\mathcal{S}_n$ is obtained by applying the last action of the sequence to state $\mathcal{S}_{n-1}$. State $\mathcal{S}_n$ is called the *final state*.

Now we can define what a primitive solution is. Recall that, intuitively, a primitive solution is a primitive plan that achieves a goal state, from an initial state, with respect to a set of operators. Formally, a *primitive solution* for a classical planning problem $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$ is a primitive plan σ such that $Res^*(\sigma, \mathcal{I}, Op) \models \mathcal{G}$, i.e., the preconditions of actions in σ are satisfied, and the final state entails the goal state.

For example, consider the following initial state:

$\{OnTable(Block1), On(Block2, Block1), Clear(Block2), ArmEmpty\}$.

Next, consider the following goal state to swap the two blocks, i.e., to place $Block1$ on to of $Block2$:

$\{OnTable(Block2), On(Block1, Block2), Clear(Block1), ArmEmpty\}$.

Then, a possible primitive solution is the following, with respect to the set of operators mentioned so far in previous examples:

$\sigma = unstack(Block2, Block1) \cdot putdown(Block2) \cdot pickup(Block1) \cdot stack(Block1, Block2)$.

The solution states that the goal state is achieved by unstacking $Block2$ from $Block1$, putting $Block2$ on the table, picking up $Block1$, and finally, stacking $Block1$ on $Block2$.

Non-redundant solutions

In addition to correctness, many domains require that plans adhere to certain other properties. This is because correct plans can still have shortcomings, such as *non-minimality* and *redundancy*. A primitive solution of length n for a classical planning problem is said to be *non-minimal* if a primitive solution of length less than n exists for the problem. A primitive solution for a planning

problem is said to be *redundant* if one or more actions can be removed from the solution and still have a solution. Of particular relevance to this thesis is the notion of non-redundancy (also called *perfect justification*). According to (Knoblock et al., 1991) and (Fink and Yang, 1992), the notion of non-redundancy is defined as follows.

Definition 1. (Perfect Justification (Fink and Yang, 1992)) A primitive solution σ for a classical planning problem $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$ is a *perfect justification* for C if there does not exist a proper subsequence σ' of σ such that σ' is a primitive solution for C. ■

For example, consider the initial state and goal state in the previous example. Suppose that the primitive solution for this problem is the following:

$\sigma = unstack(Block2, Block1) \cdot stack(Block2, Block1) \cdot unstack(Block2, Block1) \cdot putdown(Block2) \cdot pickup(Block1) \cdot stack(Block1, Block2)$.

Observe that the second and third actions – stacking $Block2$ on $Block1$, and then unstacking $Block2$ from $Block1$ – are redundant actions. (Alternatively, the first two actions can also be considered redundant.) The removal of these two actions will not cause the resulting primitive plan to be incorrect.

Unfortunately, finding perfectly justified primitive solutions is NP-hard (Fink and Yang, 1992). Consequently, Fink and Yang propose a greedy algorithm that finds an "almost" perfectly justified primitive solution in polynomial time. An adapted version of this algorithm is shown in Algorithm 2.2. The algorithm works by determining whether an action *act* in a primitive solution σ is necessary. To this end, the action is removed from σ (line 2) to obtain σ', and then it is determined whether there is any other action act' in σ' whose precondition is no longer satisfied as a result of removing *act* from σ. If so, action act' is removed from σ' (line 8). This process continues until all actions are removed from σ' whose preconditions are not satisfied as a result of removing action *act* from σ. If the final value of σ' achieves the goal state from the initial state (i.e., it is correct), then the initially removed action *act* is considered unnecessary, and the algorithm is called recursively with the new primitive solution σ'. On the other hand, if the final value of σ' is not correct, then the algorithm tries to remove a different action from σ.

Algorithm 2.2 Linear-Greedy-Justification(σ, C)

Input: Solution σ for classical planning problem $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$.
Output: A primitive solution that is an "almost" perfect justification.

1: **for** each $act \in \sigma$ **do**
2: $\sigma' \Leftarrow \sigma$ with act removed
3: $\mathcal{S} \Leftarrow \mathcal{I}$
4: **for** $act' \Leftarrow$ first action in σ' **to** last action in σ' **do**
5: **if** $\mathcal{S} \models pre(o)\theta$, where $o \in Op$ and $act' = name(o)\theta$ **then**
6: $\mathcal{S} \Leftarrow Res(act', \mathcal{S}, Op)$
7: **else**
8: remove act' from σ'
9: **end if**
10: **if** $\mathcal{S} \models \mathcal{G}$ **then**
11: **return** Linear-Greedy-Justification(σ', C)
12: **end if**
13: **end for**
14: **end for**
15: **return** σ

To illustrate how the algorithm works, consider the following redundant primitive solution from the previous example.

$\sigma = unstack(Block2, Block1) \cdot stack(Block2, Block1) \cdot unstack(Block2, Block1) \cdot putdown(Block2) \cdot pickup(Block1) \cdot stack(Block1, Block2)$.

Suppose the algorithm removes action $stack(Block2, Block1)$ from σ. This will result in the following primitive plan σ':

$\sigma' = unstack(Block2, Block1) \cdot unstack(Block2, Block1) \cdot putdown(Block2) \cdot pickup(Block1) \cdot stack(Block1, Block2)$.

However, the precondition of the second action in σ' does not hold in the state that results from applying the first action in the initial state – once $Block2$ is unstacked, it cannot be unstacked once more. Consequently, the algorithm removes the second action from σ' to obtain the following primitive plan:

σ' = $unstack(Block2, Block1) \cdot putdown(Block2) \cdot pickup(Block1) \cdot stack(Block1, Block2)$.

Since σ' is a primitive solution for the given planning problem, the algorithm is called recursively with σ' as an argument. However, no more actions can be removed from σ', resulting in it being returned as a greedily justified primitive solution (which, in this example, is also a perfectly justified solution).

Algorithms for planning from first principles

So far, we have discussed different notions related to first principles planning, such as the notion of an operator, a primitive solution, and a perfectly justified primitive solution. Next, we discuss some of the algorithms for finding primitive solutions for a given classical planning problem.

Algorithm 2.3 Forward-Search(C)

Input: Classical planning problem $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$.
Output: A primitive solution for C, or *failure* if no such solution exists.

```
1:  if I ⊨ G then
2:      return the empty plan
3:  end if
4:  applicable ⇐ {name(o)θ | o ∈ Op, name(o)θ is a ground instance of name(o), I ⊨ pre(o)θ}
5:  if applicable = ∅ then
6:      return failure
7:  end if
8:  for each act ∈ applicable do
9:      I' ⇐ Res(act, I, Op)
10:     σ ⇐ Forward-Search(⟨I', G, Op⟩)
11:     if σ ≠ failure then
12:         return act · σ
13:     end if
14: end for
15: return failure
```

The most basic planning algorithm is the *forward search* algorithm. An adapted version of the forward search algorithm in (Ghallab et al., 2004, p. 70) is shown in Algorithm 2.3. The input for this algorithm is a classical planning problem, and the output is a primitive solution for the problem. First, the algorithm finds all actions that are applicable in initial state $\mathcal{I}$, and saves these in the set *applicable* (line 4). From this set, an action is picked arbitrarily, and the result of

applying this action in state $\mathcal{I}$ is obtained as $\mathcal{I}'$ (line 9). Next, the algorithm is recursively called with the new state $\mathcal{I}'$. If the recursive call returns a primitive solution for problem $\langle \mathcal{I}', \mathcal{G}, Op \rangle$ — i.e., the goal state is eventually reached after applying some sequence of actions to $\mathcal{I}'$ (line 1) — then the result of the forward search is attached to the end of action *act*, and the resulting plan returned as a primitive solution for C. Otherwise, a different action is picked from *applicable* and the process is repeated. If none of the actions in *applicable* can be used as the first action of a sequence of actions that leads to the goal state, then *failure* is returned.

There are many state of the art planners based on forward search (e.g., (Bonet and Geffner, 1999; Hoffmann and Nebel, 2001; Refanidis and Vlahavas, 2002; Do and Kambhampati, 2001; Hoffmann and Brafman, 2006)). Of particular relevance to our work is the FF (Hoffmann and Nebel, 2001) planning system. The main idea behind FF is that delete-lists of operators in *Op* are ignored. More specifically, given a planning problem $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$, a *relaxed planning problem* $C' = \langle \mathcal{I}, \mathcal{G}, Op' \rangle$ is obtained where $Op' = \{\langle name(o), pre(o), \emptyset, add(o) \rangle \mid o \in Op\}$. In addition, FF, like its predecessor HSP (Bonet and Geffner, 1999), also uses an efficient *heuristic function* in order to determine what the most promising actions are in the set of applicable actions (line 8 of Algorithm 2.3). The search can then be biased toward the more promising actions, allowing solutions to be found faster in general than would be possible by selecting actions arbitrarily. The heuristic function is based on the Graphplan algorithm (Blum and Furst, 1995), which we discuss in detail next.

The Graphplan algorithm is based on the concept of a *planning graph*. A planning graph is a directed, levelled graph, that is, a graph in which nodes are split into levels, and an edge only connects two nodes from adjacent levels. There are two types of nodes: *proposition nodes* and *action nodes*. There are three types of edges: *precondition edges*, *add edges* and *delete edges*, representing the preconditions, add-lists and delete-lists of operators, respectively. The levels of a planning graph alternate between *proposition levels*, i.e., those containing only proposition nodes, and *action levels*, i.e., those containing only action nodes, with the first level being a proposition level. Any node at an action level i in the planning graph is connected by a precondition edge to each of the atoms in its precondition, which occur at proposition level i in the graph. Similarly, the action node is connected by an add edge to each of the atoms in its add-list, which occur at proposition level $i + 1$ in the graph. The same is true for atoms in the delete-list of the action node. An example of such a planning graph for a particular planning problem from a Blocks World

domain is shown in Figure 2.2.

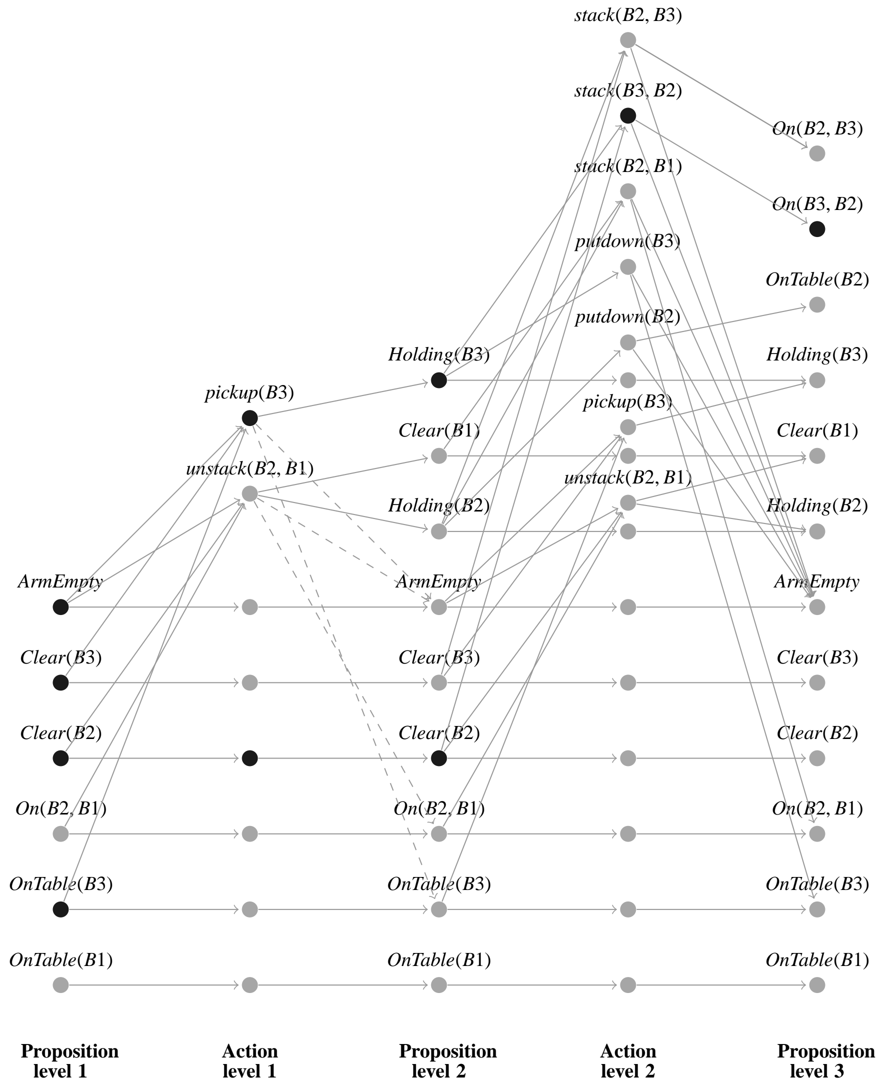


Figure 2.2: A simplified planning graph for a Blocks World planning problem. The abbreviation "*Bi*" (e.g., *B*2) is short for "*Blocki*." Nodes are labelled with either the name of an action or a proposition. Nodes with no labels are no-op actions. Solid arrows represent add edges and precondition edges, and dashed arrows represent delete edges. Delete edges have been left out of the second action level for readability. Darker nodes represent the path to a solution for goal atom $On(B3, B2)$.

Next, we describe how a planning graph is built, and how it is used as a heuristic, given a classical planning problem. First, all propositions (ground atoms) in the initial state are added to the first proposition level of the (initially empty) planning graph. Second, actions are created, with respect to the set of operators *Op*, whose preconditions are met with respect to the first

proposition level. Third, all such actions are added to the first action level of the graph, and the precondition edges corresponding to those actions are added to the graph. Any action level i also contains one *no-op* (dummy) action for each proposition at proposition level i. No-op actions are simply used for "carrying forward" propositions to the next proposition level. Fourth, the propositions in the add lists and delete lists of all actions in the first action level are added to the second proposition level of the graph, and the corresponding add edges and delete edges created. This process continues until either *(i)* all propositions in the goal state are present in the current proposition level, none of them are mutually exclusive (e.g., p and $\neg p$), and a correct plan can be extracted from the graph, or *(ii)* the planning graph *levels off*, that is a proposition level is reached that is identical to the previous proposition level, which indicates that no solution exists. Note that, while a planning graph can be created in polynomial time, the extraction of a correct plan from the graph cannot be done in polynomial time.

To determine whether a correct plan can be extracted from the planning graph at some proposition level n, the algorithm performs recursive *backward search*. Unlike forward search, backward search starts from the goal state and works backward toward the initial state. In Graphplan, backward search is performed using the planning graph to guide the search, as follows. The search begins from propositions $\mathcal{G}_n = \mathcal{G}$ in the goal state, which occur at proposition level n. First, the algorithm obtains a set of actions Δ_{n-1} occurring in action level $n-1$ that are connected by add edges to propositions in $\mathcal{G}_n$. Next, the algorithm finds the corresponding set of propositions $\mathcal{G}_{n-1}$ at proposition level $n-1$ that are connected by precondition edges to actions in Δ_{n-1}. The algorithm continues in this manner until the first proposition level – the initial state – is reached. Note that at any given action level i, there may be many possible sets of actions Δ_i that could be created. This is because a proposition in a set $\mathcal{G}_{i+1}$ could be brought about by many actions at action level i, and set Δ_i only includes one action per proposition. The algorithm also takes into account actions that are mutually exclusive. For example, if one action requires proposition p to hold, and another requires $\neg p$ to hold, only one of them is included in Δ_i. Information about mutually exclusive actions is added to the graph when it is constructed. At any given action level i, the algorithm may have to try backward search with multiple values for Δ_i before it finds one that eventually leads to the initial state.

To illustrate how a planning graph is built, and how backward search can be performed on such a graph, consider the planning graph in Figure 2.2. Suppose that

we have the following initial state:

$\mathcal{I} = \{OnTable(Block1), OnTable(Block3), On(Block2, Block1), Clear(Block2), Clear(Block3), ArmEmpty\}$,

and the following goal state:

$\mathcal{G} = \{On(Block3, Block2)\}$.

To build the graph, the initial state is added as the first proposition level of the graph, as shown in the figure. To create the first action level, all actions that are applicable with respect to the first proposition level are obtained. These are actions $pickup(Block3)$ and $unstack(Block2, Block1)$, whose preconditions are met in the first proposition level. Next, propositions that appear in the first level (carried forward by no-ops), as well as those that are brought about by the two applicable actions are added to the second proposition level. (A dashed arrow incident on a proposition node indicates that the proposition is removed by the corresponding action's delete-list.) The rest of the graph is built in a similar manner.

When the third proposition level is reached, all propositions in the goal state $\mathcal{G}$ – i.e., $On(Block3, Block2)$ – occur at this level. Consequently, the algorithm starts the backward search process. The search starts from proposition $On(Block3, Block2)$, and then moves one step backward in the graph, finding (non-conflicting) actions that bring about the proposition, i.e., action $stack(B3, B2)$. The propositions at the second proposition level, which correspond to the preconditions of this action, are then selected, i.e., propositions $Holding(B3)$ and $Clear(B2)$. At the next step, action $pickup(B3)$ and a no-op action are selected, since they bring about propositions $Holding(B3)$ and $Clear(B2)$. Finally, the search process ends when the first proposition level is reached. The primitive solution $pickup(B3) \cdot stack(B3, B2)$ is then extracted from the path traversed.

In the FF planner, the Graphplan algorithm is used as a heuristic to guide forward search. In particular, a planning graph is built for a relaxed classical planning problem by using a modified

Graphplan algorithm in order to: *(i)* estimate the most promising actions from the set of applicable actions (line 8 of Algorithm 2.3); and *(ii)* estimate the distance to the goal state from the current state. Given a state $\mathcal{S}$ at some point in the planning process, and a goal state $\mathcal{G}$, the most promising actions are considered to be the ones that appear in the first action level of a planning graph built for $\mathcal{S}$ and $\mathcal{G}$, and moreover, those that are "connected" by some path to atoms in $\mathcal{G}$; the distance to the goal is the number of actions occurring in this path. Such a path is obtained in polynomial time by basically starting from the proposition level i in the graph containing all atoms in $\mathcal{G}$, collecting actions that bring about those atoms, obtaining the preconditions of those actions, collecting actions that bring about those preconditions, and so on, until the first action level is reached. Note that, unlike the backward search process for extracting a correct plan from a planning graph, finding actions in a graph that are connected to the goal state does not involve recursion.

> For example, suppose forward search begins from the planning problem given in the previous example. Then, although actions $pickup(Block3)$ and $unstack(Block2, Block1)$ will both be included in the set *applicable* at line 8 of Algorithm 2.3, action $unstack(B2, B1)$ will not be considered to be a promising action, because in the planning graph shown in Figure 2.2, action $unstack(B2, B1)$ is not connected to the goal atom $On(Block3, Block2)$. Moreover, the distance to the goal at this point in the forward search is 2 – at least two actions are needed to achieve the goal state.

The FF planning system has been used as the basis for many subsequent planners. Some examples relevant to this thesis are (Hoffmann, 2003; Botea et al., 2005). Metric-FF is the planner we have chosen for incorporating first principles planning into the JACK BDI agent platform. In Metric-FF, the FF algorithm is extended to handle more expressive preconditions and effects. In particular, numerical calculations are allowed in preconditions and effects, which are useful when using the planner in conjunction with real world BDI applications. For example, a precondition such as the following is possible: $(and\ (At\ Robot1\ loc1)\ (At\ Robot2\ loc2)\ (<\ (-\ loc1\ loc2)\ 10))$, which requires $Robot1$ and $Robot2$ to be less than ten distance units apart.

In Macro-FF (Botea et al., 2005), the FF planner is extended with the ability to use macro actions, that is, sequences of (standard) actions. Macro actions are automatically learnt from primitive solutions for sample planning problems. These actions are then used in the planning

process, by treating them as standard actions. The authors show that macro actions speed up forward search because it is possible to select a single macro action to achieve some state (e.g., the goal state), instead of selecting multiple standard actions to achieve the same state. Moreover, the authors argue that macro actions are useful when performing backward search within a planning graph: since standard actions within a macro action are always compatible, there is no need to take into account the possibility of those standard actions being mutually exclusive.

2.3.2 Hierarchical Task Network Planning

Unlike first principles planners, which focus on bringing about states of affairs or "goals-to-be," Hierarchical Task Network (HTN) planners, like BDI systems, focus on solving *abstract/compound tasks* or "goals-to-do." Abstract tasks are solved by decomposing (refining) them repeatedly into less abstract tasks, by appealing to a given library of *methods*, until only *primitive tasks* (actions) remain. Methods contain procedural control knowledge for constraining the exploration required to solve abstract tasks – an abstract task t is solved by using only the tasks specified in a method associated with t. In this thesis, we mostly follow the definitions of HTN planning from (Erol et al., 1996).

We will now be more precise about the notions associated with HTN planning. In the previous section, we defined a classical planning problem as a tuple $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. A HTN planning problem $\mathcal{P}$, on the other hand, is a 3-tuple $\langle d, \mathcal{I}, \mathcal{D} \rangle$, where d is a *task network*, $\mathcal{I}$ is an initial state, and $\mathcal{D}$ is a HTN *planning domain*. In turn, a HTN planning domain $\mathcal{D}$ is a tuple $\langle Op_{htn}, Me \rangle$, where Op_{htn} is a set of HTN operators and Me is a set of methods. The objective of the HTN planner is to solve task network d by starting from state $\mathcal{I}$, and by making use of the set of methods Me and set of operators Op_{htn}.

Intuitively, a task network is a partially ordered collection of tasks. Before we formally define a task network, we define a (compound or primitive) task as a syntactic construct of the form $\alpha(\vec{t})$, where $\vec{t}$ is a vector of function-free terms. Then, a task network is a syntactic construct of the form:

$$[\{(n_1 : \alpha_1), \ldots, (n_m : \alpha_m)\}, \phi],$$

where the first component is a set of labelled tasks, and the second component is a *task network formula* — informally, a formula of constraints. Labels are used to distinguish between multiple non-

unique tasks occurring in the task network. The task network is solved by solving each task in its first component, while conforming to the task network formula. Formally, a *task network formula* is a boolean formula constructed from negation, disjunction and the following entities: *(i) variable binding constraints* of the form $(t = t')$, where t and t' are variables or constants; *(ii) ordering constraints* of the form $(n \prec n')$ with task labels n and n'; and *state constraints* of the form (l, n), (n, l) and (n, l, n') with task labels n and n', and with literal l. A variable binding constraint $(t = t')$ indicates that variable or constant t must be equivalent to variable or constant t', e.g., $(name = John)$. An ordering constraint $(n \prec n')$ indicates that task with label n should precede the task with label n'. State constraints (l, n) and (n, l) indicate that literal l should hold immediately before the task with label n, and that literal l should hold immediately after the task with label n, respectively. Finally, state constraint (n, l, n') indicates that literal l should hold between tasks with labels n and n'. Task labels can also be of the form $first[n_1, \ldots, n_m]$ and $last[n_1, \ldots, n_m]$, so that we can refer respectively to the task that starts first and to the task that ends last among the set $\{n_1, \ldots, n_m\}$.

Next, we define the structures that are used to solve tasks occurring in a task network. Primitive tasks occurring in a task network are handled by operators. Like a STRIPS operator, a *HTN operator* is a syntactic construct of the form:

$$[operator\ act(\vec{x})\ (pre:\ \{l_1, \ldots, l_m\})\ (post: \{l'_1, \ldots, l'_n\})],$$

where: $act(\vec{x})$, a primitive task, is the name of the operator ($\vec{x}$ is a vector of distinct variables); $l_1, \ldots, l_m$ is a set of literals corresponding to the operator's precondition; and $l'_1, \ldots, l'_n$ is a set of literals corresponding to the operator's postcondition. Like STRIPS operators, all variables occurring in the precondition and postcondition of a HTN operator also occur in $\vec{x}$. A primitive task act will have exactly one corresponding operator in the set Op_{htn}, i.e., exactly one operator in Op_{htn} with a name that unifies with act. Compound tasks occurring in a task network are handled by methods. A *method* is a syntactic construct of the form (α, d), where α is a compound task and d is a task network. A compound task can have more than one associated method in Me. A method indicates that one way to solve compound task α is to decompose it into task network d and to solve d.

For example, consider the HTN domain $\mathcal{D} = \langle Op_{htn}, Me \rangle$ illustrated graphically in Figure 2.3. (Note that this Blocks World encoding is slightly different to that used

in the previous section in that a block can be picked up from the table as well as from on top of another block). The top-level task in this domain is the compound task $unstack(b1, b2)$, which is used for moving a block $b1$ that is on top of a block $b2$ to the table. The set of methods in this domain is $Me = \{m_1, m_2\}$. Observe that method $m_1 = (unstack(b1, b2), d')$, where task network d' is the following:

$$d' = [\{(n_1 : pickup(b1, b2)), (n_2 : putdown(b1))\}, (n_1 \prec n_2) \land \phi],$$

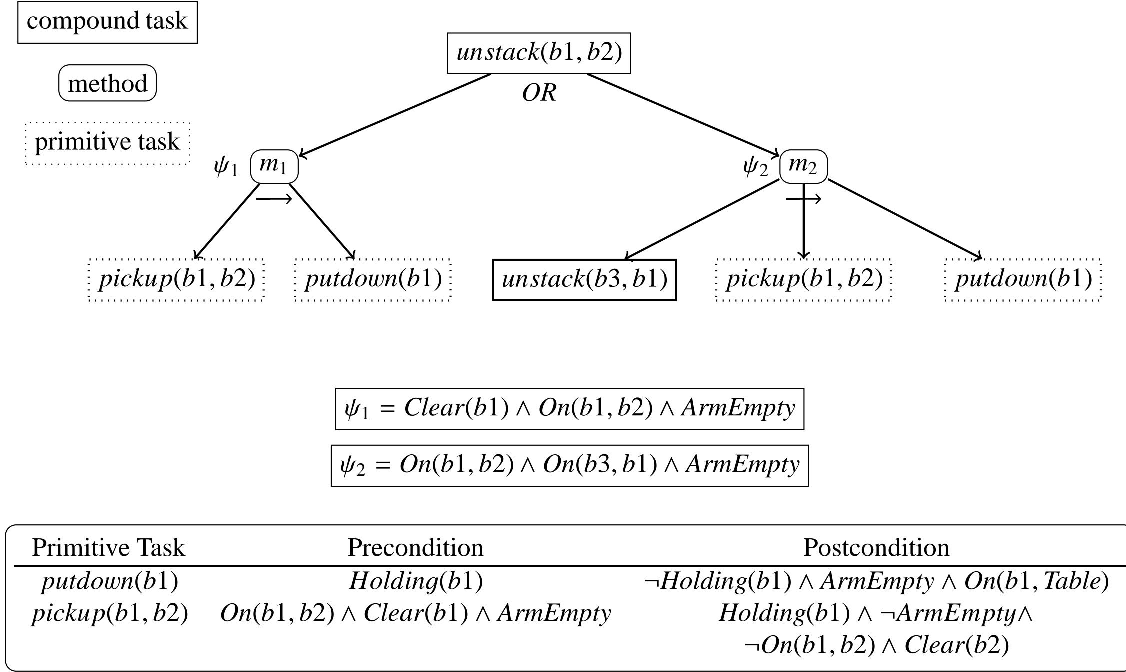
and where $\phi = (Clear(b1), n_1) \land (On(b1, b2), n_1) \land (ArmEmpty, n_1)$. Tasks $pickup(b1, b2)$ and $putdown(b1)$ are primitive tasks for, respectively, picking up a block $b1$ that is on top of a block $b2$, and for placing a block $b1$ that is currently in the robot's arm onto the table. The task network formula of d' states that $pickup(b1, b2)$ must precede $putdown(b1)$, and that initially (i.e., before task $pickup(b1, b2)$): $b1$ should be clear, $b1$ should be on top of $b2$, and that the robot's arm should be empty. Hence, method m_1 can move block $b1$ only if there are no other blocks on $b1$. Otherwise, method m_2 must be used.

Observe from the figure that method $m_2 = (unstack(b1, b2), d'')$, where task network d'' is the following:

$$d'' = [\{(n_1 : unstack(b3, b1)), (n_2 : pickup(b1, b2)), (n_3 : putdown(b1))\}, (n_1 \prec n_2) \land (n_2 \prec n_3) \land \phi'],$$

and where $\phi' = (On(b1, b2), n_1) \land (On(b3, b1), n_1) \land (ArmEmpty, n_1)$. Observe that this task network can handle the case where there are one or more blocks stacked on top of $b1$, by first clearing $b1$ — i.e., recursively moving each block on top of $b1$ to the table — and then moving $b1$ from $b2$ to the table.

Intuitively, given a HTN planning problem $\mathcal{P} = \langle d, \mathcal{I}, \mathcal{D} \rangle$ (where $\mathcal{D} = \langle Op_{htn}, Me \rangle$), the HTN planning process works as follows. First, an applicable reduction method (i.e., one whose precondition is met in the current state) is selected from Me and applied to some compound task in d. This will result in a new, and typically "more primitive" task network d'. Then, another reduction method is applied to some task in d', and this process is repeated until a task network is obtained



Primitive Task	Precondition	Postcondition
$putdown(b1)$	$Holding(b1)$	$\neg Holding(b1) \wedge ArmEmpty \wedge On(b1, Table)$
$pickup(b1, b2)$	$On(b1, b2) \wedge Clear(b1) \wedge ArmEmpty$	$Holding(b1) \wedge \neg ArmEmpty \wedge$ $\neg On(b1, b2) \wedge Clear(b2)$

Figure 2.3: A simplified representation of a HTN domain $\mathcal{D}$. An arrow below a method indicates that its tasks are ordered from left to right.

containing only primitive tasks (actions). At any stage during the planning process, if no applicable method can be found for a compound task, the planner "backtracks" and tries an alternative reduction for a compound task previously reduced.

To be more precise about the HTN planning process, we first define what a reduction is. Suppose that $d = [s, \phi]$ is a task network, $(n : \alpha) \in s$ is a labelled compound task occurring in d, and that $m = (\alpha', d') \in Me$ is a method that may be used to decompose α (i.e., α and α' unify). Then, $reduce(d, n, m)$ denotes the task network that results from decomposing labelled task $(n : \alpha)$ in task network d using method m. Informally, such decomposition involves updating both the set s in d, by replacing labelled task $(n : \alpha)$ with the tasks in d' (by arbitrarily renaming task labels), and the constraints ϕ in s to take into account constraints in d'. The set of all possible reductions of task network d is then defined as follows:[2]

$$red(d, \mathcal{D}) = \{d' \mid d' = reduce(d, n, m), (n : \alpha) \in s, m \in Me\}.$$

[2]Note that in the original formalisation of function *red* in (Erol et al., 1996), there is a third argument, namely, a state. We have omitted this argument because it is not needed.

If all compound tasks in a given initial task network can eventually be replaced by primitive tasks via reductions, the resulting final primitive task network is used to find a *completion* of the task network, that is, an ordering and grounding of the primitive tasks in the final task network such that the ordering conforms with the constraints imposed on those tasks by the network. More precisely, a plan σ is a completion of a primitive task network d (i.e., one containing only primitive tasks) at state $\mathcal{I}$, denoted $\sigma \in comp(d, \mathcal{I}, \mathcal{D})$, if σ is a total ordering of the primitive tasks in a ground instance of d, such that σ is executable in $\mathcal{I}$ (i.e., all preconditions of actions in σ are satisfied), and σ satisfies the constraint formula in d. We refer the reader to (Erol et al., 1996) for more detail about reductions and completions.

Finally, by using sets $red(d, \mathcal{D})$ and $comp(d, \mathcal{I}, \mathcal{D})$, one can easily define the set of plans $sol(d, \mathcal{I}, \mathcal{D})$ that solves a HTN planning problem $\mathcal{P} = \langle d, \mathcal{I}, \mathcal{D} \rangle$ as $sol(d, \mathcal{I}, \mathcal{D}) = \bigcup_{n<\omega} sol_n(d, \mathcal{I}, \mathcal{D})$, where $sol_n(d, \mathcal{I}, \mathcal{D})$ is, in turn, defined as follows:

$$
\begin{aligned}
sol_1(d, \mathcal{I}, \mathcal{D}) &= comp(d, \mathcal{I}, \mathcal{D}), \\
sol_{n+1}(d, \mathcal{I}, \mathcal{D}) &= sol_n(d, \mathcal{I}, \mathcal{D}) \cup \bigcup_{d' \in red(d, \mathcal{I}, \mathcal{D})} sol_n(d', \mathcal{I}, \mathcal{D}).
\end{aligned}
$$

Intuitively, the set of primitive plans that solves a HTN planning problem $\langle d, \mathcal{I}, \mathcal{D} \rangle$ is the set of all completions of all primitive task networks that can be obtained from zero or more reductions of d. We call such primitive plans *primitive plan solutions* to distinguish them from primitive solutions, which achieve some goal state, and from primitive plans, which are arbitrary sequences of actions.

As one example of how the HTN planning process works, consider the HTN domain $\mathcal{D}$ depicted in Figure 2.3. Suppose task network $d_1 = [s_1, \phi_1]$ from Figure 2.3, where $s_1 = \{(n : unstack(Block1, Block2))\}$ and $\phi_1 = true$. Moreover, suppose we have the HTN planning problem $\mathcal{P} = \langle d, \mathcal{I}, \mathcal{D} \rangle$, where:

$\mathcal{I} = \{On(Block2, Table), On(Block1, Block2), On(Block3, Block1), Clear(Block3), ArmEmpty\}$.

Then, observe that the reduction of the labelled compound task $(n : unstack(Block1, Block2)) \in s_1$ using method m_1 — that is, $reduce(d_1, n, m_1)$ — results in the following primitive task network:

$d_2 = [\{(n_1 : pickup(Block1, Block2)), (n_2 : putdown(Block1))\}, (n_1 \prec n_2) \wedge \phi_2],$

where $\phi_2 = (Clear(Block1), n_1) \wedge (On(Block1, Block2), n_1) \wedge (ArmEmpty, n_1)$. However, observe that the completion of d_2 is $comp(d_2, \mathcal{I}, \mathcal{D}) = \emptyset$, because constraint $(Clear(Block1), n_1)$ does not hold with respect to initial state $\mathcal{I}$ — $Clear(Block1)$ is not true in $\mathcal{I}$.

Next, consider, instead, the reduction of labelled compound task $(n : unstack(Block1, Block2)) \in s_1$ using method m_2. Observe that the result of this decomposition is the following task network:

$d_3 = [\{(n_1 : unstack(Block3, Block1)), (n_2 : pickup(Block1, Block2)), (n_3 : putdown(Block1))\}, (n_1 \prec n_2) \wedge (n_2 \prec n_3) \wedge \phi_3],$

where $\phi_3 = (On(Block1, Block2), n_1) \wedge (On(Block3, Block1), n_1) \wedge (ArmEmpty, n_1)$. Since there is a compound task *unstack* occurring in d_3, it needs to be reduced further before a primitive task network can be obtained. Suppose method m_1 is used for this reduction. The resulting primitive task network d_4 is then the following:

$d_4 = [\{(n_4 : pickup(Block3, Block1)), (n_5 : putdown(Block3)), (n_2 : pickup(Block1, Block2)), (n_3 : putdown(Block1))\}, (n_4 \prec n_2) \wedge (n_5 \prec n_2) \wedge (n_2 \prec n_3) \wedge (n_4 \prec n_5) \wedge \phi_4],$

where $\phi_4 = (Clear(Block3), n_4) \wedge (On(Block3, Block1), n_4) \wedge (ArmEmpty, n_4) \wedge (On(Block1, Block2), n_4)$. Observe that the contents of the task network in method m_1 is incorporated into task network d_4. In particular: *(i)* the constraint formula of method m_1 is added as a conjunction to the constraint formula of task network d_4; *(ii)* labelled tasks of m_1 are added, after the renaming of task labels, to the set of labelled tasks of d_4; and *(iii)* old constraints of d_4 — e.g., $(n_1 \prec n_2)$ — are updated to accommodate the new task labels.

The final step is to obtain the completion $comp(d_4, \mathcal{I}, \mathcal{D})$ of task network d_4.

The completion is composed of the following primitive plan solution:

$\sigma = pickup(Block3, Block1) \cdot putdown(Block3) \cdot pickup(Block1, Block2) \cdot putdown(Block1)$.

Observe that plan σ is a primitive plan solution — it is executable in $\mathcal{I}$, and ϕ_4 can be satisfied with respect to the initial state $\mathcal{I}$.

The style of HTN planning we have described so far is called *partially-ordered* HTN planning. This is because it is not necessary for tasks in a task network to be ordered in any way. In fact, it is legal for the constraint formula to not have any constraints at all. This allows tasks to be executed in parallel with other tasks, by overlapping their subtasks.

For example, consider a Blocks World domain in which there are two robot arms. Suppose we have the following initial task network:

$d = [\{(n_1 : unstack(Block1, Block2)), (n_2 : unstack(Block3, Block4))\}, true]$.

This task network specifies that the two unstack operators can be performed in parallel, that is, their decompositions can be interleaved. One example of such an interleaving is the following primitive plan solution:

$\sigma = pickup(Block1, Block2) \cdot pickup(Block3, Block4) \cdot putdown(Block3) \cdot putdown(Block1)$.

In the example in Figure 2.3, however, all tasks within task networks are *totally-ordered*, that is, all tasks occurring in a task network have a (possibly implicit) ordering enforced relative to all other tasks occurring in the network. While partial-order HTN planning is more expressive than total-order HTN planning (Nau et al., 1998), and it has the advantage of preventing excessive backtracking by not committing to the ordering of steps prematurely, total-order HTN planning also has its advantages. First, since the ordering of tasks is known in advance, total-order HTN planners know the complete state of the world at each step in the planning process. Consequently, powerful preconditions can be written such as those that do numerical computations or interact

with external information sources. Second, the complexity of total-order HTN planning is significantly less than that of partial-order HTN planning (Nau et al., 1998). This is because, without the need to interleave subtasks belonging to compound tasks, total-order HTN planners do not have the additional complexity of handling interactions between subtasks.

JSHOP total-order HTN planner

One of the most popular implementations of a total-order HTN planner is JSHOP, which we have chosen for incorporating HTN planning into the JACK agent development platform. JSHOP is a Java version of the Lisp based SHOP (Nau et al., 1999) (Simple Hierarchical Ordered Planner) total-order HTN planner, whose successor, SHOP2 (Nau et al., 2003), won one of the top four prizes at the 2002 International Planning Competition.[3] Both JSHOP and SHOP have been integrated into many different types of applications (Muñoz-Avila et al., 2001; Dix et al., 2003; Nau et al., 2005).

A JSHOP planning problem, like a HTN planning problem, is a 3-tuple $\langle \vec{\alpha}, \mathcal{I}, \mathcal{D} \rangle$, where $\vec{\alpha}$ is a sequence of (primitive and compound) tasks, $\mathcal{I}$ is the initial state, and $\mathcal{D} = \langle Op_{htn}, Me \rangle$ as before. Unlike a HTN method, a JSHOP method is of the form (: *method* α $[h]$ ψ T), where α is a compound task, ψ is a conjunction of literals representing the precondition of the method, T, called the *tail*, is a sequence of (primitive and compound) tasks, and h is an optional name for the method. Note that the precondition of a JSHOP method corresponds to a constraint formula of a HTN method, and that the precondition needs to be satisfied for the corresponding method to be applicable. Task decomposition in JSHOP works like that in HTN, except that tasks are decomposed in the same order in which they are specified in the input sequence of tasks $\vec{\alpha}$ and in the tails of methods.

2.4 Combining Agents and Planning

A number of studies have focused on combining automated planning with agent architectures. Both HTN-style planning as well as first principles planning techniques have been incorporated. With first principles planning, an agent can obtain new plans that are not already a part of the programmer supplied plan-library, whereas HTN-style planning allows an agent to look-ahead

[3] http://ipc.icaps-conference.org/

on its existing plans in order to obtain a viable decomposition of the plan. In this section, we review the works that combine agents and planning, and compare them, when appropriate, with the research questions we address in this thesis.

2.4.1 First Principles Planning in Agents

The Propice-Plan (Despouys and Ingrand, 1999) framework is the combination of the IPP (Koehler et al., 1997) first principles planner and an extended version of the PRS (Ingrand et al., 1992) BDI system. In Propice-Plan, the IPP planner is used to obtain new PRS plan-rules at runtime when none of the existing plan-rules are applicable for an event-goal that the agent wants to achieve. To formulate plans, IPP uses the plan-rules of PRS, by treating these plan-rules as operators. In particular, the precondition of an operator is taken as the context condition of the corresponding plan-rule, and the postcondition of the operator is taken as the effects of the corresponding plan-rule, which are supplied for each plan-rule by the programmer. The goal state to plan for is the primary effect of the event-goal that failed, which is also supplied by the programmer. Solutions found by IPP are returned to PRS, which executes them by mapping their actions back into ground plan-rules.

The issues addressed by the Propice-Plan system are similar to the research questions we address in this thesis. In particular, we are also interested in planning from first principles in order to obtain new plan-rules not already in the agent's library. However, there are also important differences between our work and that of (Despouys and Ingrand, 1999). First, we are interested in planning with the event-goals of the agent, as opposed to planning with the plan-rules of the agent. This is because we want plans found to be *flexible* like typical BDI plans. BDI plans are flexible in that they are built from high-level abstract goals, for which different alternatives may be tried if necessary. In the work of (Despouys and Ingrand, 1999), on the other hand, a plan returned by the planner will have committed to a sequence of ground plan-rules. The second difference is that we are interested in finding plans that are non-redundant, i.e., those that can be decomposed into primitive steps that are necessary for achieving the goal state at hand. Plans found in the work of (Despouys and Ingrand, 1999), however, do not address this issue of redundancy: their plans may be decomposed into steps that are not necessary for achieving the goal state.

With the experience gained from the Propice-Plan system, (Lemai and Ingrand, 2004) propose the IxTeT-eXeC system, which is built specifically with robotic architectures in mind, such as

Mars Rovers. IxTeT-eXeC is a combination of PRS and the IxTeT (Laborie and Ghallab, 1995) planner, which allows an expressive temporal specification of operators. Unlike Propice-Plan, IxTeT-eXeC gives more control to the planner than the BDI system. Initially, IxTeT-eXeC is given a top-level goal state to achieve by the user. IxTeT-eXeC then uses the IxTeT planner to formulate a complete solution for the goal state in terms of the operators in the domain, which correspond to, essentially, leaf-level event-goals in PRS (i.e., those handled only by plan-bodies that do not mention any event-goals). The solution is then executed by IxTeT-eXeC by sending each individual operator in the solution to PRS, one at a time. PRS executes a given operator by mapping it into the corresponding event-goal, and then executing it using the BDI execution mechanisms, which may involve (local) failure recovery by trying alternative leaf-level plan-rules. The plan-rules are composed of primitive steps that can be directly executed by the robot. Finally, PRS sends a report back to the planner indicating the result (e.g., success or failure) of executing the event-goal. If during the execution of a plan found by IxTeT a new goal arrives from the user, the old plan is repaired (if necessary) to take into account this new goal.

The focus of our research is different from (Lemai and Ingrand, 2004) in that we want the BDI system to maintain full control on when to use the planner, rather than using the BDI system only to execute plans found by the planner. Moreover, we are interested in using BDI systems in a manner that exploits their full potential by having flexible plan-libraries with different levels of abstraction. In the work of (Lemai and Ingrand, 2004), the BDI system is only used for basic execution, that is, for the decomposition of event-goals directly into primitive actions.

In (Meneguzzi et al., 2004a,b), the X-BDI (da Costa Móra et al., 1998) model is extended with first principles planning capabilities. The X-BDI model is a traditional cognitive BDI agent architecture based on notions such as beliefs, desires and intentions. The reasoning process of X-BDI involves the following steps. First, a set of *eligible desires* is obtained from the agent's set of desires, where an eligible desire is one that meets certain rationality constraints put forth by Bratman, such as being unachieved. Second, the set of eligible desires is refined further to obtain a set of *candidate desires*, which are desires that are both possible (i.e., a plan exists to handle them) and consistent as described in Section 2.1.1. Finally, the set of candidate desires are used to obtain a set of *primary intentions*, which are plans corresponding to the agent's commitment to achieve its candidate desires. First principles planning is introduced into the X-BDI model to replace the algorithm which selects the set of candidate desires from the set of eligible desires. In particular,

candidate desires are essentially eligible desires for which plans can be found.

While X-BDI allows a logical and declarative specification of BDI agents, it does not lend itself well to practical and efficient implementations (Meneguzzi and Luck, 2007). On the other hand, we use a practical BDI agent-oriented programming language. More importantly, however, unlike their work, our work focuses on: *(i)* finding plans that re-use and respect the hierarchical domain information inherent in the agent's plan-library, whereas in their work planning is performed with basic, primitive actions of the agent, thereby not making use of, and possibly not conforming to the domain information inherent in the library; and *(ii)* using first principles planning solely for the purpose of means-ends reasoning — i.e., deciding *how* to bring about a state of affairs (see Section 2.1.1) — whereas in their work, first principles planning is used primarily to aid in deliberation — i.e., deciding *what* state of affairs to bring about.

Another approach that incorporates first principles planning into a BDI system is (Meneguzzi and Luck, 2008, 2007). In this work, planning is added into the AgentSpeak BDI agent programming language. Like our work, (Meneguzzi and Luck, 2007) allows calls to the planner to be made at any programmer specified point in the agent's plan-library, and moreover, the planning is performed for a goal state that is supplied by the programmer. The domain information used by the planner is automatically extracted (at runtime) from the primitive actions belonging to the agent, which are encoded as leaf-level AgentSpeak event-goals. A plan found is executed by the agent by mapping actions in the plan back into their corresponding event-goals. Like the previous work described, the work of (Meneguzzi and Luck, 2008, 2007) also performs first principles planning with the primitive actions of the agent, rather than with higher level entities like we do in this thesis.

The intermediate layer of the InteRRaP hybrid architecture discussed in Section 2.2 can also plan from first principles in case a plan-rule is not available in its library. However, like some of the systems described above, the solutions generated seem to be composed entirely of primitive actions.

Apart from the systems that combine first principles planning and BDI-like systems, there are also systems that add planning into other agent architectures. Of particular relevance to our work are systems that combine first principles planning with the Golog (Levesque et al., 1997) action language, which has been successfully used for robot control. In (Claßen et al., 2007), IndiGolog (Sardina et al., 2004) – an implementation of Golog – is extended with the FF (Hoffmann and

Nebel, 2001) classical planning system. IndiGolog already supports planning from first principles via its *achieve*(G) procedure, where G is a goal state formula to achieve. In (Claßen et al., 2007), another similar construct is added to the language, which amounts to calling the FF planner. The returned plan (if any) – a sequence of primitive actions – is executed within the IndiGolog engine. The objective of this work is twofold: *(i)* to provide a translation from IndiGolog actions into a version of PDDL (Planning Domain Definition Language); and *(ii)* to show that by using the FF planner for planning, as opposed to the built-in IndiGolog procedure, an efficiency improvement can be gained.

Compared to our work, the work of (Claßen et al., 2007) uses a more expressive language to describe primitive actions, which has the ability to specify things such as quantification within preconditions. Still, since the plans found by FF are a sequence of the agent's primitive actions, as opposed to the more abstract entities that make up our plans, the procedural information inherent in Golog procedures are not exploited, and the plans are not flexible.

On the other hand, (Baier et al., 2007; Fritz et al., 2008) addresses the issue of planning from first principles in ConGolog – Golog with support for specifying concurrency – in a way that respects and exploits the domain control knowledge inherent in ConGolog programs. To this end, they provide a translation from a subset of the language of ConGolog into PDDL operators. The translation takes into account the domain control knowledge inherent ConGolog programs. Specifically, these operators ensure that primitive solutions resulting from the planning process conform to the ConGolog programs given. Moreover, (Baier et al., 2007) provides different heuristics for planning, which show how planning speed can be improved when the domain control knowledge encoded in the operators is effectively used.

The issue we address in this thesis of conforming to the procedural information inherent in BDI programs is similar to the issue addressed in (Baier et al., 2007; Fritz et al., 2008) of conforming to the domain control knowledge inherent in ConGolog programs. However, while the solutions found by the first principles planner in this thesis can contain abstract (and hence flexible) BDI entities corresponding to event-goals, solutions found in the work of (Baier et al., 2007; Fritz et al., 2008) are composed entirely of primitive actions. Furthermore, although these primitive solutions do conform to the given ConGolog programs, they may still have redundant steps, which is undesirable in our work.

In addition to the above differences with Golog based languages, our work is also different

to such languages because they are cognitive agent languages, with no explicit notions of entities such as event-goals, plan-rules, plan selection and failure, whereas our approach is linked to a family of (practical) BDI agent-oriented programming languages and systems.

Finally, work such as (Clement et al., 2007; Tambe and Zhang, 2000) deal with planning in order to coordinate actions/plans belonging to multiple agents. In (Tambe and Zhang, 2000), a state in the state space is an agent's model of the overall state of the team to which the agent belongs, and actions used for planning are team actions, i.e., those that typically affect the entire team. Unlike the works mentioned, this thesis deals with single-agent planning. However, we discuss in detail the work of (Clement et al., 2007) in Chapter 4, where we extend their algorithms for the purpose of adding first principles planning into the CAN language.

2.4.2 HTN Planning in Agents

Perhaps some of the first systems to incorporate HTN-style look-ahead into agents are (Lyons et al., 1991; Mcdermott, 1991). In these systems, the task of the planner is to continuously revise the plan-rules of the agent in order to make the agent behave in a more goal directed manner. For example, the XFRM (Mcdermott, 1992) system incorporates HTN-style look-ahead into RPL (Mcdermott, 1991) (Reactive Plan Language), a BDI-like language which has many similarities with its ancestors PRS and RAP (Firby, 1987). Execution in XFRM begins when the agent is given a set of top-level event-goals to achieve. While the agent tries to achieve these event-goals via decomposition, the planning component continuously looks ahead on the agent's currently executing plan-rule in order to assist the agent in avoiding future failures. To this end, the planning component revises the agent's plan-rule(s), e.g., by adding constraints on the ordering of steps, or forcing the agent to follow a particular decomposition.

Although these works have certain similarities with the research questions we address in this thesis, we are only interested in performing HTN look-ahead at programmer specified points in an agent's library. In this way, HTN-style look-ahead is only used when it is needed. Moreover, although we are also interested in guiding the agent along successful (virtual) decompositions, we do not do this by revising the agent's currently executing plan-rules. This is because conforming to the user's intent is important in our work, and such modifications may result in plan-rules that no longer conform to the user's intent.

In the Cypress system (Wilkins and Myers, 1995; Wilkins et al., 1995), the SIPE-2 (Wilkins,

1990) HTN planning system is combined with an extended version – PRS-CL – of the PRS BDI system. After identifying similarities between the syntax and semantics of PRS-CL and SIPE-2, the authors combine the two systems via the *Act* language, which is a superset of the languages of PRS-CL and SIPE-2. The programmer writes the domain specification in the Act language, which is converted, at runtime, into the languages of PRS-CL and SIPE-2 as and when needed.

The system works by using the SIPE-2 HTN planner to look-ahead on PRS-CL event-goals up to a level of abstraction decided by the programmer for the given domain. Once an abstract plan is returned by SIPE-2, the PRS-CL execution engine fills in the remaining details, by decomposing the event-goals in the plan completely, down to the level of primitive actions. For this to work, certain plan-rules are only allowed to be used by SIPE-2, and the others are only allowed to be used by PRS-CL. In this way, the planner does not decompose a given event-goal beyond a certain level of abstraction, and PRS-CL does not execute event-goals that are above a certain level of abstraction. The rationale behind this is that it is often not feasible (e.g., due to time constraints) for the planner to look-ahead up to the smallest level of detail, and that the executor should not execute very abstract event-goals without performing any look-ahead. Therefore, in some sense, solutions returned by SIPE-2 are flexible – they are composed of abstract entities whose exact refinements are handled by PRS-CL. In addition to using look-ahead for solving very abstract event-goals, the SIPE-2 planner is also used by PRS-CL when certain types of failures occur during execution, such as when no plan-rules exist to handle an event-goal.

In comparison with Cypress, our system always focuses on decomposing event-goals completely, i.e., up to the level of primitive actions, because we are interested in getting a guarantee that a given event-goal has some successful (virtual) decomposition. While it is also possible to get such guarantees in Cypress by forcing both PRS-CL and SIPE-2 to use the same set of plan-rules, this will result in solutions found by SIPE-2 being composed entirely of primitive actions. Consequently, while failure will only be detected in Cypress on the failure of a primitive action, in our system, failure may be detected earlier due to a plan-rule being inapplicable, as the HTN planner is used only as a means for guiding the BDI system in choosing appropriate plan-rules at choice points.

In the RETSINA (Paolucci et al., 1999) system, agents solve their top-level event-goals by performing HTN decomposition. If the information required to decompose some lower level event-goal is not available at the time of planning, the agent then suspends the decomposition, locates the

relevant information gathering actions in the plan being developed that would obtain the necessary information, and then executes these actions. Once the information is obtained, the decomposition of the top-level event-goal continues. RETSINA also makes use of Rationale Based Monitoring (Veloso et al., 1998) in order to monitor conditions that are related to the plan being developed. If while a plan is being developed a change in the environment makes a monitored conditions false, the planning process is abandoned.

In comparison with our work, RETSINA agents always perform HTN look-ahead, unless information needs to be gathered from the environment. In our work, on the other hand, the agent typically follows standard BDI execution, and uses the HTN planner only at points during the execution where the programmer has deemed it necessary to perform HTN look-ahead. However, we assume that all the information necessary for HTN planning is available before the planner is called, whereas they do not make this assumption.

Finally, none of the systems mentioned above provide a formal integration of a HTN semantics into a BDI agent-oriented programming language, and an analysis of the properties of such an integration, which we do in this thesis.

Chapter 3

A HTN Planning Framework for BDI Systems[†]

In this chapter, we incorporate look-ahead deliberation in the style of Hierarchical Task Networks (HTN) into BDI agents. Such look-ahead is desirable, or even mandatory in situations where undesired outcomes need to be avoided. For instance, an agent may want to reason about the consequences of choosing one expansion of a task over another, for guiding the selection of recipes to avoid negative interactions between them.

We choose HTN planning because of the similarities it shares with BDI systems in problem representation and in reasoning, but also because HTN semantics and implementations are well understood in the planning community (Nau et al., 2005; Erol et al., 1996). We first explore in detail the similarities mentioned in past work (e.g., (Wilkins et al., 1995)) between the two approaches, and then exploit these similarities. To this end, we incorporate HTN planning into the semantics and infrastructure of a BDI agent programming language, in a precise and formal manner. Our new BDI infrastructure includes HTN planning as a built-in feature that the agent programmer can use when required. We show that the new infrastructure is provably more expressive than HTN systems alone, and that it allows the programmer to, under certain restrictions, rule out BDI executions that are bound to fail.

[†]The updated version of CAN in Section 3.2 was developed primarily by Sebastian Sardina (co-supervisor of the author of this thesis), with some participation from the author of this thesis, from Lin Padgham (supervisor of the author of this thesis and co-author of (Winikoff et al., 2002)), and in discussion with Michael Winikoff (first author of (Winikoff et al., 2002)). Part of the work presented in this chapter has been previously published in (de Silva and Padgham, 2004, 2005; Sardina et al., 2006).

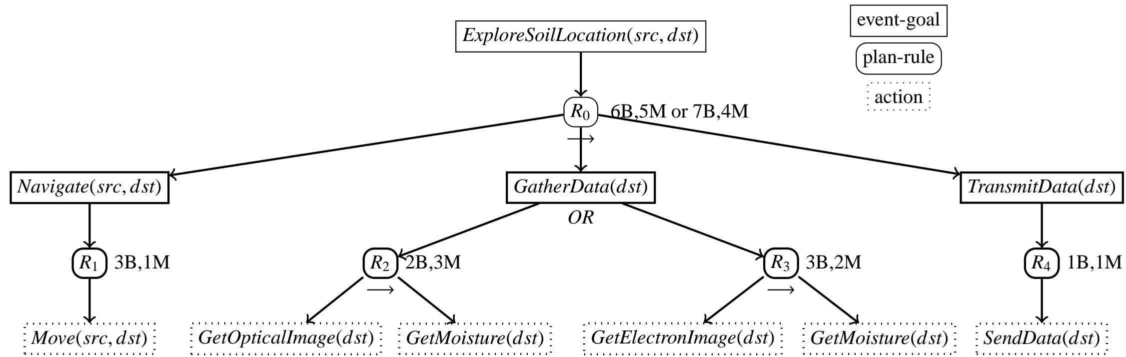


Figure 3.1: A simple Mars Rover agent. An arrow below a plan-rule indicates that its steps are ordered from left to right. The labels adjacent to plan-rules are the resources that they consume: nB stands for n units of battery, and nM stands for n units of memory.

Although frameworks do exist that incorporate some type of look-ahead planning as a built-in feature of BDI-style systems (e.g., (Ambros-Ingerson, 1987; Wilkins and Myers, 1998; Despouys and Ingrand, 1999; Graham et al., 2003; Paolucci et al., 1999; Knoblock, 1995)), these are mostly implemented systems with no precise semantics, and with little or no programmer control over when to plan. It is worth noting that, in fact, look-ahead procedures can sometimes be explicitly programmed into existing BDI systems. However, such procedures would in general be domain dependent, fairly complex, and would not be tightly integrated with the infrastructure support provided by the BDI agent platform. It is sometimes also possible to avoid look-ahead altogether, by carefully specifying context conditions of plan-rules. The types of BDI applications that require look-ahead are those in which there are potentially negative interactions between different branches of an event-goal, and these interactions need to be predicted and avoided during execution.

As one example of the value of the kind of look-ahead planning we propose, consider Figure 3.1, which shows a simple Mars Rover agent. The rover's top-level event-goal is to carry out a soil experiment at some destination *dst* from its current location *src*, which involves navigating to the destination, gathering data from the destination, and then transmitting the data to the lander. Suppose that the actions shown in the figure require the following resources (where *GetOpticalImage* gets a coloured image using an optical microscope, and *GetElectronImage* gets a greyscale image using an electron microscope, which has a higher magnification than the op-

tical microscope):

1. *Move*: three units of battery and one unit of memory;
2. *GetOpticalImage*: one unit of battery and two units of memory;
3. *GetElectronImage*: two units of battery and one unit of memory; and
4. *GetMoisture*, *SendData*: one unit of battery and one unit of memory.

Since each microscope has one advantage and one disadvantage compared with the other (i.e., colour versus greyscale, and high magnification versus low magnification), we assume that it does not matter which microscope is used if there are at least two units of battery and two units of memory.

Next, suppose plan-rules have the following context conditions: *(i)* R_1 is applicable only if the rover is at location *src*, and there are at least three units of battery and there is at least one unit of memory for moving to the destination; *(ii)* R_4 is applicable only if the rover has data for the destination, and there is at least one unit of memory and battery for sending the data to the lander; *(iii)* R_2 is applicable only if the rover is at the destination, and there are at least two units of battery and three units of memory for both getting an image with the optical microscope and extracting moisture content from the soil; *(iv)* R_3 is applicable only if the rover is at the destination, and there are at least three units of battery and two units of memory for both getting an image with the electron microscope and extracting moisture content from the soil; and finally, *(v)* R_0 is applicable only if the rover is at *src*, and one or both of the following conditions hold: *(a)* there are at least six units of battery and five units of memory (for navigating, gathering data by getting an image with the optical microscope, and sending data); or *(b)* there are at least seven units of battery and four units of memory (for navigating, gathering data by getting an image with the electron microscope, and sending data).

Now, observe that if the rover initially has seven units of battery and four units of memory, the rover will successfully navigate to the destination, but it may then select R_2 instead of R_3 (since both are applicable) and not have enough memory to transmit data. Similarly, if the rover initially has six units of battery and five units of memory, the rover will successfully navigate to the destination, but it may then

select R_3 instead of R_2 and not have enough battery to transmit data.[1] However, if the agent were able to perform look-ahead on R_0 from the initial state in which there are seven (respectively six) units of battery and four (respectively five) units of memory, the agent would realise that plan-rule R_3 (respectively R_2) is the one that leads to a successful (virtual) decomposition of R_0, and it would select R_3 (respectively R_2) during execution.

This chapter is organised as follows. In Section 3.1, we compare a typical BDI agent programming language, namely AgentSpeak (Rao, 1996), with the HTN language of (Erol et al., 1996), in order to identify their similarities and differences, which involves mapping from BDI entities to HTN entities. Then, in Section 3.2, we create a new language, namely CANPlan, by incorporating HTN planning into the CAN BDI agent programming language (Winikoff et al., 2002), which is based on AgentSpeak. Incorporating HTN planning into CAN involves *(i)* adding a planning module into the CAN semantics, and providing an operational semantics that defines the behaviour of the new module (Section 3.2.3); and *(ii)* exploring the theoretical properties of the new framework (Section 3.2.4).

3.1 Similarities Between the BDI and HTN Approaches

While BDI agent systems are focused on the *execution* of agent programs in dynamic environments, HTN systems are concerned with *hypothetical reasoning* about actions and their potential interactions within a whole plan for achieving a task. Despite their different purposes, however, BDI systems and HTN planners share many similarities. These include how knowledge is represented, as well as how this knowledge is manipulated to solve problems. Despite integrated systems such as (Wilkins et al., 1995; Paolucci et al., 1999) which incorporate some of the strengths of each approach, and despite there being past work that mentions similarities between the two approaches (e.g., (Clement and Durfee, 1999, 2000; Firby, 1989; Wilkins et al., 1995; Paolucci et al., 1999)), there does not appear to be any work which formally maps between the domain representations of the two approaches. In Sections 2.1.3 and 2.3.2, we described the conceptual entities of the BDI and HTN approaches. In this section, we compare and contrast the two approaches, and

[1] Note that, although such failure can be avoided by writing a context condition for R_0 that requires at least seven units of battery and five units of memory, such a context condition is too restrictive, as it will rule out the possibility of exploring a soil location with fewer resources (e.g., six units of battery and five units of memory).

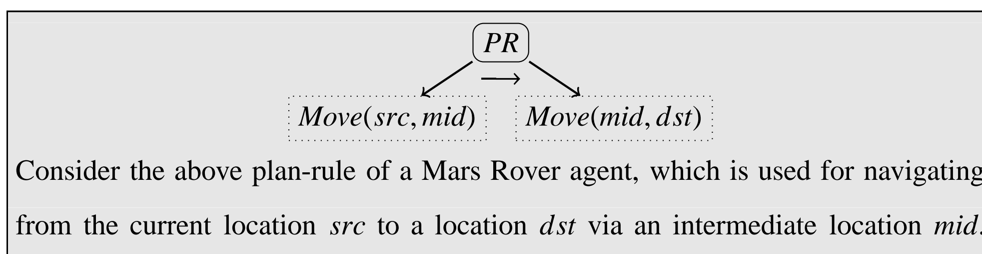
provide a mapping from BDI entities to HTN entities.

In terms of syntax, both BDI and HTN approaches assume an explicit representation of the agent's knowledge, namely, the belief base and state, respectively. Moreover, the domain information of each approach is described procedurally, in the form of plan-rules and reduction methods, respectively.

Let us once again consider Figure 3.1. In addition to representing a BDI plan-library, this structure also represents a HTN method-library. In particular, the rectangles represent both BDI event-goals (or achievement goals) and HTN compound (or non-primitive) tasks; the rounded rectangles represent both BDI plan-rules and HTN methods; and the dotted rectangles represent both BDI actions and test conditions, as well as HTN primitive tasks and state constraints, respectively.

In terms of semantics, the two approaches formulate solutions in a similar manner. Both approaches decompose higher-level tasks into lower-level tasks, by appealing to a given library of recipes. While a BDI system decomposes an event-goal into a plan-body program using a relevant and applicable plan-rule from the plan-library, a HTN system decomposes a compound task into a task network using a relevant and applicable reduction method from the method-library. If the path of decompositions pursued for solving a particular task ends up not working, both systems backtrack, i.e., return to a higher-level task, to pursue an alternative path of decompositions. However, due to a difference in the meaning of an *action* in the two approaches, the reasons for backtracking are different.

While actions in the BDI approach are executed in the real world, actions (primitive tasks) in the HTN approach only make changes to its internal model of the world. As a result, while the intended outcome of executing a BDI action can only be confirmed by (external) events from the environment, the intended outcome (i.e., postcondition) of a HTN primitive task is guaranteed on its (virtual) execution.



Consider the above plan-rule of a Mars Rover agent, which is used for navigating from the current location *src* to a location *dst* via an intermediate location *mid*.

Suppose the precondition of the action on the left is $At(src)$, the precondition of the action on the right is $At(mid)$, the plan-rule's context condition always succeeds, and that the rover is at *src*.

Now, observe that if action $Move(src, mid)$ does not move the rover from *src* to *mid* (e.g., the action fails), then action $Move(mid, dst)$ will not be applicable, resulting in the plan-rule failing.[2] In HTN planning, on the other hand, action $Move(mid, dst)$ will always be applicable, because action $Move(src, mid)$ will always bring about its intended effect $At(mid) \wedge \neg At(src)$ within the context of the planner.

Due to the difference in the meaning of an action, the following differences also arise in the semantics of backtracking in the two approaches. In the HTN approach, backtracking is performed when it is predicted via complete look-ahead, that a solution being pursued will not work due to unavoidable conflicts between tasks. Consequently, backtracking involves "ignoring" the solution pursued so far, from the point at which backtracking begins, up to the point at which an alternative decomposition is tried. On the other hand, backtracking is performed in a BDI system due to its inability to predict the consequences of actions. More specifically, a BDI agent backtracks when an action executed (in the real world) does not bring about its intended outcome. Backtracking in BDI does not, however, involve "ignoring" the solution pursued so far, because actions have already been executed in the real world. Rather, backtracking is done to try and achieve the failed event-goal using a different plan-rule in a potentially new world state.

Mapping the AgentSpeak BDI Language to the HTN Language

We will now give a precise account of how BDI entities can be mapped to HTN entities. Such a mapping is essential to be able to use BDI entities for HTN planning, which we do in the next section. The BDI agent programming language we have chosen for this mapping is AgentSpeak (Rao, 1996; Moreira and Bordini, 2002; Bordini et al., 2002; d'Inverno and Luck, 1998), in particular, the original formalisation in (Rao, 1996).[3] AgentSpeak is a high-level plan language that attempts to extract the essence of a class of implementable agent platforms, such as PRS (Georgeff

[2]Of course, if the rover is somehow moved to *mid* by some other means, then the second action will indeed be applicable.

[3]Note that a formal mapping from CAN entities to HTN entities can be found in the proof of Theorem 2.

and Ingrand, 1989) and dMars (d'Inverno et al., 1998). More importantly, AgentSpeak forms the basis for later BDI agent programming languages such as CAN (Winikoff et al., 2002) and 3APL(Hindriks et al., 1999). The HTN language we have chosen is that of (Erol et al., 1996), which is one of the most widely used HTN formalisations in the literature. Table 3.1 provides a summary of the conceptual mapping from AgentSpeak entities to HTN entities.

AgentSpeak Entities	HTN Entities
set of base beliefs	state
belief operations ($+b$ and $-b$)	primitive task
action (*act*)	dummy primitive task
achievement goal ($!g$)	compound task
test goal ($?g$)	state constraint
plan-context	state constraints
sequencing (;)	ordering constraints
plan-body	task network
plan-rule	method
plan-library	set of methods

Table 3.1: Summary of the mapping from AgentSpeak to HTN

Set of base beliefs to state

As mentioned before, a *set of base beliefs* and a *state* are both representations of knowledge about the world in AgentSpeak and HTN, respectively. Moreover, a set of base beliefs and a state have the same form: they are both sets of ground atoms. An explicit mapping is therefore not needed.

Belief operation to primitive task

An AgentSpeak agent updates its knowledge about the world using *belief operations* $+b$ and $-b$.[4] Similarly, a HTN planner updates its knowledge about the world using *primitive tasks*.

The mapping is done as follows. First, a unique primitive task is assigned to each unique belief operation of the agent (recall from Section 2.3.2 that a primitive task is simply a symbol followed by a vector of terms). Second, for each such primitive task, an operator is created to handle it, with

[4]Actually, in AgentSpeak, the agent programmer is not allowed to specify belief operations in plan-bodies — updates to the set of base beliefs are only performed internally by AgentSpeak, when belief-update events arrive from the environment. However, we do provide a mapping for belief operations in the style of CAN, because they are an important BDI feature that has been included in improvements and extensions of AgentSpeak (Moreira and Bordini, 2002; Bordini and Moreira, 2004; Bordini et al., 2002).

precondition *true*, and with postcondition $\{b\}$ if the corresponding belief operation is $+b$, or $\{\neg b\}$ if the corresponding belief operation is $-b$.

Action to dummy primitive task

In AgentSpeak, an *action* is used to make changes to the real world. Although AgentSpeak (and CAN) actions do have intended outcomes, such actions represent arbitrary operations, and their intended outcomes are not explicitly specified. For this reason, we provide a model of actions in the next section. For now, however, we will simply assume that the intended outcomes of actions are specified as belief operations, and map actions to *dummy* HTN primitive tasks, i.e., primitive tasks with the precondition *true* and with the empty postcondition.

Achievement goal to compound task

An *achievement goal* in AgentSpeak and a *compound task* in HTN both correspond to a task that the agent wants to solve. Moreover, as mentioned before, both achievement goals and compound tasks are solved in a similar manner, by appealing to a given set of recipes. Mapping an achievement goal to a compound task is straightforward since they are both essentially just names (i.e., a symbol followed by a vector of terms) representing the relevant plan-rules and reduction methods, respectively.

Plan-body to task network

An AgentSpeak *plan-body* and a HTN *task network* is respectively one possible way of solving an achievement goal and a compound task. While a plan-body selected is added to the agent's set of intentions and executed, a task network selected is added to the solution being pursued. In terms of syntax: a plan-body is of the form $P_1; \ldots; P_k$, where each P_i is an (achievement or test) goal, action or belief operation; and a HTN task network is a tuple of the form $[\{(n_1 : t_1), \ldots, (n_m : t_m)\}, \phi]$, where the element on the left is a set of labelled (compound or primitive) tasks, and the element on the right is a task network formula.

Mapping from a plan-body to a task network consists of two parts: *(i)* mapping all entities within the plan-body, excluding test goals, to a set of labelled tasks; and *(ii)* mapping test goals within the plan-body to state constraints – testing for conditions can only be specified in HTN as state constraints within a task network formula.

For the first part of the mapping, we create a set S of labelled tasks from plan-body $P_1; \ldots ; P_k$, by adding to S the labelled task $(i : t_i)$ for every element P_i in the plan-body that is not a test goal, where t_i is the HTN entity corresponding to (i.e., HTN mapping of) P_i.

For the second part of the mapping, we create a formula of HTN constraints ϕ from the plan-body by *(a)* adding state constraint (i, g) as a conjunction for every test goal $P_j = ?g$ mentioned in the plan-body, where $i < j$ is the largest task label occurring in S; and *(b)* adding an ordering constraint $(i \prec j)$ as a conjunction for every task label i, j occurring in S such that $i < j$. Recall from Section 2.3.2 that (i, g) entails that g must hold immediately after task label i, and that $(i \prec j)$ entails that the task labelled i must precede the task labelled j.

In order to cater for situations in which the first program mentioned in the plan-body is a test goal (in which case state constraint (i, g) cannot be specified since there are no elements S), we initially add to S the labelled dummy task $(0 : dummy)$.

For example, consider plan-body $P = ?p; +q; -r$, which first tests for condition p, then performs the belief operation $+q$, and finally performs belief operation $-r$. Initially, the set of labelled tasks is

$$S = \{(0 : dummy)\}.$$

In the first part of the mapping, the set S is populated with the HTN entities corresponding to the BDI entities within plan-body P. A possible result of this first step is the set

$$S = \{(0 : dummy), (2 : addQ), (3 : delR)\},$$

where $addQ$ is the primitive task corresponding to belief operation $+q$, and $delR$ is the primitive task corresponding to belief operation $-r$. Observe that the labels of tasks added to S are the positions within P of the corresponding BDI entities; therefore, since the first program of P is a test condition, task $addQ$, which corresponds to the second program of P, is given label 2. In the second part of the mapping, since program $?p$ is the only test condition within P, the initial value of the constraint

formula is simply

$$\phi = (0, p),$$

which states that condition p must hold right at the beginning, i.e., immediately after the dummy task. Finally, formula ϕ is updated with ordering constraints to obtain formula

$$\phi = (0, p) \land (0 \prec 2) \land (0 \prec 3) \land (2 \prec 3),$$

which additionally states that the dummy task precedes the other two tasks, and that primitive task *addQ* precedes primitive task *delR*. Therefore, the final HTN mapping of plan-body $P = ?p; +q; -r$ is task network $[S, \phi]$.

Plan-context to state constraints

In AgentSpeak, a plan-rule's *context* specifies the conditions under which the plan-rule is applicable for an achievement goal, with respect to the set of base beliefs. Similarly, HTN *state constraints* within the constraint formula of a method are used (among other things) to specify the conditions under which the method is applicable for a compound task, with respect to the state.

Mapping a plan-context to state constraints is straightforward. Since a plan-rule's context is simply a conjunction of literals of the form $l_1 \land \ldots \land l_n$, the corresponding formula of state constraints is $\phi = (l_1, 0) \land \ldots \land (l_n, 0)$, where 0, as shown in the previous mapping, is the label of the first (dummy) task in a task network.

Plan-rule to method

A *plan-rule* (or plan) in AgentSpeak is of the form $+!g : l_1 \land \ldots \land l_m \leftarrow P_1; \ldots; P_n$, where $+!g$ is the triggering event,[5] $l_1 \land \ldots \land l_m$ is the plan-context, and $P_1; \ldots; P_n$ is the plan-body. In

[5] Actually, a triggering event is of the form $+b$, $-b$, $+!g$, $+?g$, $-?g$, or $-!g$, where $!g$ is an achievement goal, $?g$ is a test goal and b is a base belief. There are two things to note regarding triggering events. First, like (Hindriks et al., 1998), we do not consider triggering events of the form $+?g$, $-?g$, and $-!g$ in our mapping because the operational semantics of such triggering events are neither provided nor clear in (Rao, 1996). In (Hübner et al., 2006), an informal semantics for $-!g$ is given where this is used as a means to facilitate "backtracking," i.e., the trying of alternative plans on the failure of a plan to solve an achievement goal. We will compare the "backtracking" mechanisms of HTN and BDI systems later in this chapter. Second, since $+b$ and $-b$ triggering events are *external* events from the *environment* notifying the agent of a change that occurred (Bordini and Moreira, 2004), we do not need to consider these in our mapping.

HTN, a *method* is of the form $(\alpha, [\{(n_1 : t_1), \ldots, (n_m : t_m)\}, \phi])$, where α is a compound task and $[\{(n_1 : t_1) \ldots (n_m : t_m)\}, \phi]$ is a task network. The mapping from a plan-rule to a method relies on the following mappings discussed previously: *(i)* achievement goal to compound task, *(ii)* plan-body to task network, and *(iii)* plan-context to state constraints. In particular, the only additional step required is to add the state constraints corresponding to the plan-context as a conjunction to the task network formula corresponding to the plan-body.

3.2 Adding HTN Planning into the CAN BDI Language

In (Winikoff et al., 2002), the CAN (Conceptual Agent Notation) BDI agent programming language is introduced. CAN is a high-level plan language which, like AgentSpeak, attempts to extract the essence of a class of implementable agent platforms. We choose CAN from the numerous available options (e.g., AgentSpeak (Rao, 1996) and 3APL (Hindriks et al., 1999; van Riemsdijk et al., 2003)) because it includes semantics for sophisticated BDI failure handling.

In this section, we present first an updated version of CAN, which *(i)* is cleaner than the version in (Winikoff et al., 2002), and *(ii)* incorporates variable binding details that were omitted in (Winikoff et al., 2002). We then incorporate HTN planning into this new language.

3.2.1 Presentation of CAN

A CAN BDI agent is created by the specification of a *belief base* $\mathcal{B}$, i.e., a set of formulas from some logical language, and a *plan-library* Π, i.e., a set of plan-rules. However, since in practice the belief base is a set of ground atoms, and since we need to map belief bases to *states* in the planning literature (which are sets of ground atoms), we assume that a CAN belief base is a set of ground atoms. The language of the plan-library is the language of first-order logic with equality, excluding functions and universal quantification (therefore, all free variables are existentially quantified). A plan-rule is of the form $e : \psi \leftarrow P$, where e is an *event-goal* and ψ is the *context condition*. The component P within a plan-rule is called a *plan-body* or *program*, which is built using the following components: primitive actions (*act*) that the agent can execute directly; operations to add ($+b$) and remove ($-b$) beliefs; tests for conditions ($?\phi$); and event-goal programs or (internal) achievement goals ($!e$). Complex programs can be specified using sequencing ($P_1; P_2$)

and parallelism ($P_1 \| P_2$). The *user language* of CAN, then, is described by the following grammar:

$$P ::= act \mid +b \mid -b \mid ?\phi \mid !e \mid P_1; P_2 \mid P_1 \| P_2.$$

In the original version of CAN in (Winikoff et al., 2002), the user language also includes the declarative goal-program construct $\mathsf{Goal}(\phi_s, P, \phi_f)$, which, intuitively, states that (declarative) goal ϕ_s should be achieved using (procedural) program P, failing if ϕ_f becomes true. The operational semantics provided in (Winikoff et al., 2002) for goal-programs captures some of the desired properties of declarative goals, such as *persistent*, *possible*, and *unachieved*. For example, if the program P within goal-program $\mathsf{Goal}(\phi_s, P, \phi_f)$ has completed execution, but condition ϕ_s is still not true, then P will be re-tried; moreover, if ϕ_s becomes true during the execution of P, the goal-program will succeed immediately. However, we do not deal with declarative goals in our work, because we are only interested in adding planning to a typical BDI agent programming language. We refer the reader to (Sardina and Padgham, 2007; Sardina et al., 2006) for details on how declarative goals can be incorporated into CANPlan.

In addition to the user language, there are also auxiliary plan forms which are used by CAN internally, when assigning semantics to constructs. These are the programs *nil*, $P_1 \triangleright P_2$, and $(\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)$. Intuitively, *nil* is the empty program—there is nothing left to execute, program $(\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!)$ is a set of relevant plan-rules for some event-goal, and program $P_1 \triangleright P_2$ means that program P_1 should be executed first, and that program P_2 should be executed if and only if P_1 fails. The *full language* of CAN is therefore described by the following grammar:

$$P ::= nil \mid act \mid ?\phi \mid +b \mid -b \mid !e \mid P_1; P_2 \mid P_1 \triangleright P_2 \mid P_1 \| P_2 \mid (\!|\psi_1 : P_1, \ldots, \psi_n : P_n|\!).$$

Since the language of CAN allows variables in certain programs, we frequently make use of notions associated with variable bindings or *substitutions*.

Definition 2. (Substitution) A *substitution* θ is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where $x_1, \ldots, x_n$ are distinct variables, and each t_i is a term such that $t_i \neq x_i$. We say that θ is a *ground substitution* if $t_1, \ldots, t_n$ are ground terms. Finally, θ is a *variable renaming substitution* for some expression E if each variable occuring in E is in $\{x_1, \ldots, x_n\}$ and $t_1, \ldots, t_n$ are distinct variables. ■

As usual, $\theta\theta'$ denotes the composition of substitutions θ and θ', and given an expression E and a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, we use $E\theta$ to denote the expression obtained from E by simultaneously replacing each occurrence of x_i in E with t_i, for all $i \in \{1, \ldots, n\}$.

The operational semantics of CAN is given by a set of transition rules in the style of Plotkin's structural single-step operational semantics (Plotkin, 1981). A *transition* $C \longrightarrow C'$ denotes that *configuration* C yields configuration C' in a single execution step. Similarly, $C \longrightarrow$ denotes that there is some configuration C' that can be reached by performing a single execution step from C. The relation $\stackrel{*}{\longrightarrow}$ denotes the reflexive transitive closure of $\longrightarrow$. The *transition relation* on a configuration is defined using one or more derivation rules. Derivation rules have an *antecedent* and a *conclusion*: the antecedent can either be empty, or it can consist of transitions and auxiliary conditions; the conclusion is a single transition.

$$\frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle}\ Event$$

$$\frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i\theta}{\langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P_i\theta \triangleright (\!|\Delta \setminus \{\psi_i : P_i\}|\!)\rangle}\ Sel$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, act\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle}\ act_t \qquad \frac{\mathcal{B} \models \phi\theta}{\langle \mathcal{B}, \mathcal{A}, ?\phi\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle}\ ?$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, +b\rangle \longrightarrow \langle \mathcal{B} \cup \{b\}, \mathcal{A}, nil\rangle}\ +b \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, -b\rangle \longrightarrow \langle \mathcal{B} \setminus \{b\}, \mathcal{A}, nil\rangle}\ -b$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil\,; P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle}\ Seq_t \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1; P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P'; P_2)\rangle}\ Seq$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \triangleright P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P' \triangleright P_2)\rangle}\ \triangleright$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil \triangleright P')\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle}\ \triangleright_t \qquad \frac{P_1 \neq nil \quad \langle \mathcal{B}, \mathcal{A}, P_1\rangle \not\longrightarrow}{\langle \mathcal{B}, \mathcal{A}, (P_1 \triangleright P_2)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P_2\rangle}\ \triangleright_f$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil \parallel P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle}\ \parallel_{t_1} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, (P \parallel nil)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle}\ \parallel_{t_2}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \parallel P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P' \parallel P_2)\rangle}\ \parallel_1 \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \parallel P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P_1 \parallel P')\rangle}\ \parallel_2$$

Figure 3.2: CAN's complete set of rules

The derivation rules of CAN are shown in Figure 3.2. In these rules, a configuration, called a *basic configuration*, is the tuple $\langle \mathcal{B}, \mathcal{A}, P\rangle$, where $\mathcal{B}$ is a belief base, P is a plan-body, and component $\mathcal{A}$ is a sequence of actions, which is used to keep track of the actions executed so far. This component will be used in a derivation rule introduced later in this chapter.

The *Event* rule collects all the relevant plan-bodies for the corresponding event-goal, along with their associated context conditions, and stores them in $(\!|\Delta|\!)$. The *Sel* rule selects an applicable plan-body from $(\!|\Delta|\!)$, i.e., one whose corresponding context condition is met in the current belief base, and schedules the plan-body for execution. The $+b$ rule adds belief atom b to the belief base, and similarly, rule $-b$ removes belief atom b from the belief base. Rules *Seq* and Seq_t handle the execution of two programs in a sequence in the usual way: the former rule takes a single step on the program on the left, and the latter rule replaces program *nil* with the program on the right. Like the *Seq* rule, the $\triangleright$ rule takes a single step on the program on the left. The $\triangleright_f$ rule handles the case where the executing program P_1 has failed – i.e., where P_1 cannot make a transition – by selecting and scheduling the alternative program P_2 for execution. The $\triangleright_t$ rule handles the case where the program on the left has successfully executed, by replacing the entire program with *nil*. The act_t rule states that the execution of any action trivially succeeds.

Rule ? handles the execution of a test condition $?\phi$: the test condition succeeds if formula ϕ holds in the current belief base, and it fails otherwise, that is, it cannot make a transition. Observe from the antecedent of this rule that a substitution θ is applied to ϕ. There are two things to note regarding substitutions. First, although not shown in our semantics, configurations must include a substitution to keep track of bindings obtained so far for variables during the execution of a plan-body, so that the stored bindings can be applied to variables that occur again in the remaining plan-body. Second, observe that variables may be shared among programs occurring in a larger program, and that programs may fail during execution. For example, in program $P_1 \triangleright P_2$, the same variable, say x, may occur in both P_1 and P_2, and P_1 may eventually fail. Therefore, the semantics should be able to handle the "removal" of bindings given to variables occurring in failed programs, so that variables may be bound once again by the other programs. For example, if a binding is given to variable x when P_1 is tried, and P_1 then fails, P_2 should be allowed to obtain a different binding for x. However, for legibility, and because reassigning bindings to variables is not necessary for the semantics of our planning framework, we keep substitutions implicit in places where they need to be carried across multiple rules. We refer the reader to (Hindriks et al., 1999; Sardina and Padgham, 2010) for an account of how substitutions can be carried across derivation rules.

Finally, the $\|_1$ and $\|_2$ rules handle the execution of two programs in parallel by nondeterministically selecting either the program on the left, or the one on the right, and then performing a single

step on the selected program, and rules $\|_{t_1}$ and $\|_{t_2}$ remove the program *nil* from parallel programs.

3.2.2 Preliminary Definitions

In this section, we present some preliminary background material from (Sardina et al., 2006). In particular, we focus on two things: first, how an agent evolves from one state to another, that is, rules for *agent configurations*, which work at a level above those defined so far for *basic configurations*; and second, what it means for a plan-body program to (weakly) *simulate* another plan-body program. This notion is needed in the next section to show that if the expressivity of a BDI agent is limited in a certain way, then HTN planning is no more than a look-ahead mechanism on standard BDI execution.

Note that in the original operational semantics of CAN, an agent includes a *goal base* $\mathcal{G}$ that keeps track of the declarative goals that the agent has already committed to via goal-programs. Although this goal base is not utilised in the original CAN semantics, it can potentially be used at the agent level execution for reasoning about goals, e.g., for conflict detection and resolution as done in (Thangarajah et al., 2003b). Since we do not deal here with the $\mathsf{Goal}(\phi_s, P, \phi_f)$ construct, we exclude the goal base $\mathcal{G}$ from our agents.

An agent configuration, or just an agent, is a tuple of the form $\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle$, where $\mathcal{N}$ is the name of the agent, Λ is an action-library, Π is a plan-library, $\mathcal{B}$ is a belief base, $\mathcal{A}$ is the sequence of actions that the agent has executed so far, and Γ is the set of current intentions (that is, plan-body programs). Observe, therefore, that a basic configuration is simply an agent configuration that *(i)* focuses on a single intention, and *(ii)* does not contain the static components $\mathcal{N}$, Λ, and Π. Transitions between agent configurations are dictated by the following three rules:

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} \ A_{step}$$

$$\frac{e \text{ is an external event-goal}}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{!e\} \rangle} \ A_{event}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \not\longrightarrow}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\} \rangle} \ A_{clean}$$

Rule A_{step} performs a single step in one intention. More specifically, if it is possible to perform a single step in one of the intentions in Γ, rule A_{step} replaces that intention with the result of performing a single step in it. Rule A_{event} creates a new intention from a new external event-

goal. Note that we assume that the agent is made known of relevant changes in the environment via external event-goals. Such event-goals could arrive from some external source or from the agent's own sensors. Finally, rule A_{clean} removes from the intention base a top-level intention that has *(i)* successfully completed, i.e., the intention *nil*; or *(ii)* failed, i.e., one that cannot make a transition. Intuitively, this is in accordance with the BDI theory of (Rao and Georgeff, 1991), where an intention is dropped if it has been achieved, or it is impossible to achieve (see Section 2.1.1).

Next, we move on to the technical definitions. First, we define the meaning of an *agent execution*. Intuitively, an agent execution is a sequence of agent configurations, where each configuration in the sequence is obtained by performing a single transition on the previous configuration.

Definition 3. (BDI Execution) A BDI *execution* E of an agent $C_0 = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0 \rangle$ is a, possibly infinite, sequence of agent configurations $C_0 \cdot C_1 \cdot \ldots \cdot C_n \cdot \ldots$ such that $C_i \Longrightarrow C_{i+1}$, for every $i \geq 0$. A *terminating* execution is a finite execution $C_0 \cdot \ldots \cdot C_n$ where $C_n = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_n, \mathcal{A}_n, \{\} \rangle$.[6] An *environment-free* execution is one in which rule $\mathcal{A}_{event}$ has not been used — that is, there have been no changes in the environment. ■

To be able to define what it means for a program to simulate another program, we need to first define what it means for two executions to be *equivalent*. For this notion, given an execution, we only take into account configurations within the execution in which changes occur in either the executed actions $\mathcal{A}$ or the belief base $\mathcal{B}$ – i.e., configurations that do not change with respect to either of these entities are disregarded. The rationale behind this is that we are sometimes not interested in what the agent executes, unless the execution changes it's internal beliefs, or updates its sequence of executed actions. An execution with unchanged configurations removed is called a *derived execution*. Before we define this notion, given any basic configuration $\langle \mathcal{B}, \mathcal{A}, P \rangle$, we define the *projection* of the first component of the tuple as $C|_{\mathcal{B}}$, the second component of the tuple as $C|_{\mathcal{A}}$, and the third component of the tuple as $C|_P$. Then, formally, if $E = C_0 \cdot \ldots \cdot C_n$ is a finite execution, the *derived execution* $\overline{E}$ of E is the sequence of configurations obtained from E by removing all configurations C_j in the sequence such that $C_j|_{\mathcal{B}} = C_{j-1}|_{\mathcal{B}}$ and $C_j|_{\mathcal{A}} = C_{j-1}|_{\mathcal{A}}$.

In addition to the notion of a derived execution, we also use the following notation to track the evolution of an intention within an execution. Suppose $C_0 \cdot \ldots \cdot C_n$ is a normal or derived

[6] Note that $\Gamma_n = \{\}$ is possible if, for example, an external event-goal is added via rule A_{event}, the event-goal turns out to have no associated plan-rules, and consequently, the event-goal is removed via rule A_{clean}.

execution and P is an intention in C_0 (i.e., $P \in \Gamma_0$). Then, the sequence $P_0 = P, P_1, \ldots, P_n$ denotes P's evolution within the execution, where for any $i \in \{0, \ldots, n\}$, $P_i \in \Gamma_i \cup \{\epsilon\}$, where ϵ is used to denote that the intention had been removed from the intention base at some C_j, for $j \leq i$. Observe that ϵ is not a program itself, but just an auxiliary meta-level notation.

For example, consider agent configuration $C = \langle \mathcal{N}, \Lambda, \Pi, \{q\}, act, \{(+p; +q), +r\}\rangle$. Observe that *(i)* intention base Γ contains the two intentions $+p; +q$ and $+r$; *(ii)* the sequence of executed actions $\mathcal{A}$ contains one action act; and that *(iii)* the belief base $\mathcal{B}$ of the agent is $\{q\}$. Now, consider the following agent level execution of configuration C:[7]

$\langle \mathcal{N}, \Lambda, \Pi, \{q\}, act, \{(+p; +q), +r\}\rangle \cdot$
$\langle \mathcal{N}, \Lambda, \Pi, \{p, q\}, act, \{+q, +r\}\rangle \cdot$
$\langle \mathcal{N}, \Lambda, \Pi, \{p, q\}, act, \{nil, +r\}\rangle \cdot$
$\langle \mathcal{N}, \Lambda, \Pi, \{p, q\}, act, \{+r\}\rangle \cdot$
$\langle \mathcal{N}, \Lambda, \Pi, \{p, q, r\}, act, \{nil\}\rangle \cdot$
$\langle \mathcal{N}, \Lambda, \Pi, \{p, q, r\}, act, \emptyset\rangle$.

Observe that the third, fourth and sixth configurations in the execution are unchanged configurations, because each of these configurations have the same belief bases and action sequences as the configurations immediately before them. Observe, further, that the evolution of intention $+p; +q$ within this execution is $(+p; +q), +q, nil, \epsilon, \epsilon, \epsilon$, and that the evolution of intention $+r$ within this execution is $+r, +r, +r, +r, nil, \epsilon$.

Based on the notions of a derived execution and the evolution of an intention, we can now define what it means for two agent executions to be equivalent. Specifically, we are interested in agent executions that are equivalent modulo particular intentions. This is because, in order to define what it means for a program P' to simulate another program P, relative to an execution of P, we need to know whether P' can produce the same execution, modulo the (possibly different) programs P and P'.

[7] Note that the transition to the last configuration happens via rule A_{clean}.

Definition 4. (Equivalent Executions) Two, possibly derived, agent executions $C_0 \cdot \ldots \cdot C_n$ and $C'_0 \cdot \ldots \cdot C'_n$ are said to be *equivalent modulo intentions* iff for every $0 \leq i \leq n$, the configuration $C'_i = \langle \mathcal{N}_i, \Lambda_i, \Pi_i, \mathcal{B}_i, \mathcal{A}_i, \Gamma'_i \rangle$, where $C_i = \langle \mathcal{N}_i, \Lambda_i, \Pi_i, \mathcal{B}_i, \mathcal{A}_i, \Gamma_i \rangle$. Also, the two executions are *equivalent modulo intentions $P_0 \in \Gamma_0$ and $P'_0 \in \Gamma'_0$* if they are equivalent modulo intentions and for every $0 \leq i \leq n$, $(\Gamma'_i \setminus \{P'_i\}) = (\Gamma_i \setminus \{P_i\})$, where P_i (P'_i) is P_0's (P'_0's) evolution within execution $C_0 \cdot \ldots \cdot C_i$ ($C'_0 \cdot \ldots \cdot C'_i$). ■

Finally, we define some basic terms associated with the execution of an intention, and we define what we mean by a program simulating another program.

Definition 5. (Intention Execution) Let E be a BDI execution $C_0 \cdot C_1 \cdot \ldots \cdot C_n$ for an agent $C_0 = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}_0, \mathcal{A}_0, \Gamma_0 \rangle$, where $P_0 \in \Gamma_0$, and $P_0, P_1, \ldots, P_n$ is the evolution of P_0 within the execution. Then, intention P_0 in C_0 has been *fully* executed in E if $P_n = \epsilon$; otherwise P_0 is currently *executing* in E. In addition, intention P_0 in C_0 has been *successfully* executed in E if $P_i = nil$, for some $i \leq n$; intention P_0 has *failed* in E if it has been fully but not successfully executed in E. ■

We say that a program P' simulates another program P, relative to an execution E of P, if P' has an execution E' that is equivalent to E modulo (respectively) P' and P, and E' is successful if E is successful.

Definition 6. (Program Simulation) Let P, P' be programs and let E be an execution of a configuration $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P\} \rangle$, where $P, P' \notin \Gamma$. Program P' *simulates* program P *in execution* E iff there is an execution E' of configuration $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P'\} \rangle$ such that *(a)* $\overline{E}$ and $\overline{E'}$ are equivalent modulo respectively P and P'; and *(b)* if P has been successfully executed in E, so has P' in E'. We say that P' *simulates* P iff P' simulates P in every execution of any configuration. ■

3.2.3 Adding HTN Planning into CAN: the Plan Construct

In order to integrate HTN planning into the revised CAN language described in Section 3.2.1, we need to address several issues. First, we need to provide a model of actions. Second, we need to keep the full language as uniform as possible. Third, we need to give the BDI programmer control over when to perform HTN planning. Fourth, we need to determine what BDI domain knowledge

the HTN planner will use – we want the planner to re-use as much information as possible from the existing BDI domain. Finally, we need to determine how to execute a HTN solution from within the BDI execution cycle, while at the same time monitoring the execution if possible.

We address the last four issues by introducing a new program construct into the CAN language, namely, Plan(P), where P is a plan-body program. Intuitively, this construct means "*execute program P only if there is a complete hierarchical decomposition for P*." In this way, an agent executing a program within the Plan construct performs a complete HTN look-ahead search before committing to even the first step in the program. Before providing an operational semantics to define the behaviour of the Plan construct, we first discuss our representation and semantics of actions.

Actions

In the BDI languages proposed by (Rao, 1996; Winikoff et al., 2002), actions are not modelled – an action is defined as any arbitrary operation that is always applicable, and one that always succeeds. In contrast, we consider actions to be the usual basic means of an agent to make changes to its environment. This view of actions is especially important in our work because we are interested in adding planning (HTN and classical) to BDI agents, and planners require an explicit representation of the preconditions and postconditions of actions.

In order to incorporate actions into CAN, we add to the definition of an agent a STRIPS-like *action-library* Λ, containing rules of the form $act : \psi_{act} \leftarrow \Phi^-_{act}; \Phi^+_{act}$, one for each action type in the domain. Like CAN actions, the action name *act* can correspond to any arbitrary operation (e.g., a low-level function in C that activates a robot's actuators). All that we are interested in capturing is the action's precondition ψ_{act} and its postcondition, i.e., its add list of atoms Φ^+_{act} and delete list of atoms Φ^-_{act}. The language of the action-library is the same as that of the plan-library. Moreover, like the definition of a classical planning operator (Ghallab et al., 2004, p. 28), the following conditions hold for our action-rules: *(i)* free variables in ψ_{act}, Φ^-_{act} and Φ^+_{act} are also free in *act*; *(ii)* ψ_{act} is a conjunction of literals; and *(iii)* *act* is a symbol followed by a vector of distinct variables.

For example, action $move(x, y, z)$, which moves object x from y to z, could be repre-

sented in the action-library Λ as follows:

$$move(x, y, z) : Free(z) \wedge At(x, y) \longleftarrow \{Free(z), At(x, y)\}; \{Free(y), At(x, z)\}.$$

An alternative, more involved approach for modelling actions would be to follow (Hindriks et al., 1999) and to assume that a partial function $\mathcal{T}$, specifying the update semantics of basic actions, is given. More precisely, if $\mathcal{T}(act, \mathcal{B})$ is defined, it yields the new updated belief base $\mathcal{B}'$; otherwise, the action's precondition is not met in $\mathcal{B}$. However, for simplicity, we stick to the STRIPS-like action library described.

The rule that defines the behaviour of an action is shown next. This rule states that an action is executed by *(i)* selecting an applicable action-rule (if any) from the action-library Λ; *(ii)* adding the action to the sequence of actions $\mathcal{A}$; and *(iii)* updating belief base $\mathcal{B}$ with the add and delete lists of the action.

$$\frac{act' : \psi \leftarrow \Phi^-; \Phi^+ \in \Lambda \quad act'\theta = act \quad \mathcal{B} \models \psi\theta}{\langle \mathcal{B}, \mathcal{A}, act \rangle \longrightarrow \langle (\mathcal{B} \setminus \Phi^-\theta) \cup \Phi^+\theta, \mathcal{A} \cdot act, nil \rangle} \; act$$

Finally, we add another reasonable restriction to the definition of an action: we require the post-conditions of actions to be *consistent*. This restriction ensures, to a certain extent, that actions are written with appropriate care.

Definition 7. (Consistent Actions) Let Λ be an action-library and let $act : \psi \leftarrow \Phi^-; \Phi^+ \in \Lambda$ be an action-rule. Then, $act : \psi \leftarrow \Phi^-; \Phi^+$ is <u>*consistent*</u> relative to Λ iff for all ground instances $act\theta$ of act and belief bases $\mathcal{B}$, if $\mathcal{B} \models \psi\theta$, then $\Phi^+\theta \cup \{\neg b \mid b \in \Phi^-\theta\}$ is consistent. ■

Plan construct

We now provide operational semantics to define the behaviour of the Plan construct. To do this, we first introduce two types of (labelled) transitions on basic configurations: bdi-type transitions and plan-type transitions. Intuitively, bdi-type transitions are used for the standard BDI execution cycle, and plan-type transitions are used for (internal) deliberation steps within a *planning* context. By distinguishing between these two types of transitions, we are able to disallow certain rules, such as those dealing with failure handling, from being used in a planning context. We write $C \xrightarrow{\mathsf{t}} C'$ to specify a single step transition of type t, where t is either bdi or plan. When no label is specified

on a transition, both types apply.

Next, we show the main operational rule for the Plan construct. This rule, which is influenced by the Σ construct of (De Giacomo and Levesque, 1999), states that a basic configuration $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle$ can evolve into a configuration $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle$ if the following two conditions are met: *(i)* configuration $\langle \mathcal{B}, \mathcal{A}, P \rangle$ can evolve into configuration $\langle \mathcal{B}', \mathcal{A}', P' \rangle$, and *(ii)* it is possible to reach a *final* configuration from $\langle \mathcal{B}', \mathcal{A}', P' \rangle$ in a finite number of planning steps.

$$\frac{\langle \mathcal{B}, \mathcal{A}, P \rangle \overset{\mathsf{plan}}{\longrightarrow} \langle \mathcal{B}', \mathcal{A}', P' \rangle \quad \langle \mathcal{B}', \mathcal{A}', P' \rangle \overset{\mathsf{plan}_*}{\longrightarrow} \langle \mathcal{B}'', \mathcal{A}'', nil \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \overset{\mathsf{bdi}}{\longrightarrow} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle} \ \mathsf{Plan}$$

There are also two simpler rules associated with the Plan construct. These are shown below.

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(nil) \rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil \rangle} \ \mathsf{Plan}_t$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P \rangle \overset{\mathsf{plan}}{\longrightarrow} \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \overset{\mathsf{plan}}{\longrightarrow} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle} \ \mathsf{Plan}_{\mathsf{P}}$$

Rule Plan_t deals with the trivial case of planning on program *nil*, by simply removing the Plan construct from the program. Rule $\mathsf{Plan}_{\mathsf{P}}$ handles the Plan construct within a planning context: if a $\mathsf{Plan}(P)$ program is encountered during an execution that is already within a planning context, rule $\mathsf{Plan}_{\mathsf{P}}$, unlike rule Plan, avoids looking ahead on P, and performs instead a single (arbitrary) plan-step on P. The reason for performing a single step on P is that it is not clear what it means to perform planning when already within a planning context. Therefore, any (nested) Plan construct encountered from within a planning context is essentially ignored.

Certain transition rules only make sense in the context of BDI execution, in particular, the rules that deal with failure handling. As mentioned before, the concept of BDI-style failure handling, where on the failure of a step in some state an alternative is tried from that state, does not exist in HTN planning – HTN solutions do not include such failures. The rule that handles BDI-style failure is $\rhd_f$ from Section 3.2.1. We make this rule unavailable during planning by making it a bdi-type transition, and we refer to the new version of the rule as $\rhd_f^{bdi}$. In addition to this modification, we also need to modify slightly the agent level rules A_{step} and A_{clean} from Section 3.2.2, so that

they are defined in terms of bdi-type transitions. The three updated rules are shown below:

$$\frac{P_1 \neq nil \quad \langle \mathcal{B}, \mathcal{A}, P_1 \rangle \overset{\mathsf{bdi}}{\not\longrightarrow}}{\langle \mathcal{B}, \mathcal{A}, (P_1 \rhd P_2) \rangle \overset{\mathsf{bdi}}{\longrightarrow} \langle \mathcal{B}, \mathcal{A}, P_2 \rangle} \rhd_f^{bdi}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \overset{\mathsf{bdi}}{\longrightarrow} \langle \mathcal{B}', \mathcal{A}', P' \rangle}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} A_{step}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \overset{\mathsf{bdi}}{\not\longrightarrow}}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\} \rangle} A_{clean}$$

Observe that, with the alternative rule $\rhd_f^{bdi}$, only the BDI execution cycle would be allowed to try alternative plans from Δ for an event-goal upon the failure of some plan previously tried from Δ. Therefore, although a program of the form $(?false \rhd (\!|\Delta|\!))$ has no transition within a planning context, this program does have a transition within a BDI context – program $(\!|\Delta|\!)$ will be tried. Since the above rules are not available in the planning context, planning does not merely amount to looking ahead on the BDI execution cycle.

Let us explain how the Plan construct works with an example. Consider an agent with the following four plan-rules (where P_2 is some plan-body):

$e_0 : true \leftarrow \mathsf{Plan}(!e_1)$

$e_1 : p \leftarrow +u; !e_2$

$e_1 : \neg u \leftarrow +r$

$e_2 : q \leftarrow P_2$

Suppose the initial belief base of the agent is $\mathcal{B}_0 = \{p\}$. To understand how program $\mathsf{Plan}(!e)$ works, let us first consider the execution of program $!e_1$ alone.

In the first execution step, rule *Event* is applied to obtain the set of relevant plans Δ for event-goal e_1:

$$\frac{\overline{\Delta = \{(p : +u; !e_2), (\neg u : +r)\}}}{\langle \mathcal{B}_0, \mathcal{A}, !e_1 \rangle \longrightarrow \langle \mathcal{B}_0, \mathcal{A}, (\!|\Delta|\!) \rangle} Event$$

In the next step, an applicable plan-body is selected from Δ and scheduled for

execution. Since both plan-bodies in Δ are applicable in this case, suppose the plan-body selected is $+u; !e_2$:

$$\frac{\overline{(p : +u; !e_2) \in \Delta \quad \mathcal{B}_0 \models p}}{\langle \mathcal{B}_0, \mathcal{A}, (\!|\Delta|\!) \rangle \longrightarrow \langle \mathcal{B}_0, \mathcal{A}, (+u; !e_2) \rhd (\!|\{\neg u : +r\}|\!) \rangle} \; Sel$$

In the next step, the belief operation in program $+u; !e_2$ is performed; in particular, belief atom u is added to belief base $\mathcal{B}_0$ to obtain belief base $\mathcal{B}_1 = \{p, u\}$:

$$\frac{\overline{\langle \mathcal{B}_0, \mathcal{A}, +u; !e_2 \rangle \longrightarrow \langle \mathcal{B}_1, \mathcal{A}, nil; !e_2 \rangle} \; +b}{\langle \mathcal{B}_0, \mathcal{A}, (+u; !e_2) \rhd (\!|\{\neg u : +r\}|\!) \rangle \longrightarrow \langle \mathcal{B}_1, \mathcal{A}, nil; !e_2 \rhd (\!|\{\neg u : +r\}|\!) \rangle} \; \rhd$$

Two steps later, the set of relevant plans Δ for event-goal $!e_2$ is obtained:

$$\frac{\dfrac{\overline{\Delta = \{q : P_2\}}}{\langle \mathcal{B}_1, \mathcal{A}, !e_2 \rangle \longrightarrow \langle \mathcal{B}_1, \mathcal{A}, (\!|\Delta|\!) \rangle} \; Event}{\langle \mathcal{B}_1, \mathcal{A}, !e_2 \rhd (\!|\{\neg u : +r\}|\!) \rangle \longrightarrow \langle \mathcal{B}_1, \mathcal{A}, (\!|\Delta|\!) \rhd (\!|\{\neg u : +r\}|\!) \rangle} \; \rhd$$

Now, observe that the only plan in Δ is not applicable for event-goal e_2, because $\mathcal{B}_1 \not\models q$. Consequently, a transition is not possible from configuration $\langle \mathcal{B}_1, \mathcal{A}, (\!|\Delta|\!) \rangle$. Moreover, the alternative plan-body $+r$ in $(\!|\{\neg u : +r\}|\!)$ cannot be successfully executed either, because the belief base is $\mathcal{B}_1 = \{p, u\}$ and the context condition of plan-body $+r$ requires $\neg u$ to hold. Consequently, the top-level event-goal program $!e$ fails.

Let us now consider the execution of program $\mathsf{Plan}(!e)$. In the first step of the execution, rule Plan will be applicable, resulting in the set of relevant plans being selected as before:

$$\frac{\dfrac{\overline{\Delta = \{(p : +u; !e_2), (\neg u : +r)\}}}{\langle \mathcal{B}_0, \mathcal{A}, !e_1 \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}_0, \mathcal{A}, (\!|\Delta|\!) \rangle} \; Event}{\langle \mathcal{B}_0, \mathcal{A}, \mathsf{Plan}(!e_1) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_0, \mathcal{A}, \mathsf{Plan}((\!|\Delta|\!)) \rangle} \; \mathsf{Plan}$$

In the next step, rule Plan will be applicable again. However, unlike before, from the relevant set of plans Δ, plan-body $+u; !e_2$ will not be selected, because it

is not possible to successfully execute (decompose) this plan-body, for the reasons shown before. More specifically, condition $\langle \mathcal{B}_0, \mathcal{A}, (+u; !e_2) \rhd (\!|\{\neg u : +r\}|\!) \rangle \xrightarrow{\textsf{plan}_*} \langle \mathcal{B}', \mathcal{A}', nil \rangle$ required by the antecedent of rule Plan is not met. However, condition $\langle \mathcal{B}_0, \mathcal{A}, +r \rhd (\!|\{p : +u; !e_2\}|\!) \rangle \xrightarrow{\textsf{plan}_*} \langle \mathcal{B}', \mathcal{A}', nil \rangle$ is indeed met, resulting in plan-body $+r$ being selected for execution, and program Plan($!e$) succeeding. It is important to note that, of course, the first execution step of program Plan($!e$) would not have happened at all, if the successful execution of the Plan($!e$) program were not possible.

The full set of CANPlan rules is shown in Figure 3.3.

3.2.4 Properties of the Plan Construct

So far, we have provided a framework for planning from within the CAN BDI language, namely, the Plan(P) program construct. Next, we discuss properties of the Plan(P) construct. In particular, we show that Plan(P) does indeed amount to HTN planning on program P.

It was shown in the previous example that an agent will not make a BDI step on a program Plan(P) unless that step leads to a successful execution of P. It can then be expected that, if there is a successful HTN execution for a program P, then there is also a successful BDI execution for the program, provided there is no intervention from the outside environment.

Lemma 1. *For every belief base $\mathcal{B}$ sequence of actions $\mathcal{A}$ and program P, if $\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\textsf{plan}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$, then $\langle \mathcal{B}, \mathcal{A}, \textsf{Plan}(P) \rangle \xrightarrow{\textsf{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$.*

Proof. See Appendix A.1. □

Similarly, but more importantly, if an agent contains Plan(P) as its only intention, and the intention is able to start executing, then there is at least one successful BDI execution for the intention, provided there is no intervention from the outside environment.

Theorem 1. *Let $C = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \{\textsf{Plan}(P)\} \rangle$ such that $\langle \mathcal{B}, \mathcal{A}, \textsf{Plan}(P) \rangle \xrightarrow{\textsf{bdi}}$. For any environment-free agent execution E of C, intention Plan(P) is either executing or has been successfully executed in E. Moreover, there is an execution E^s of C in which intention Plan(P) has been successfully executed in E^s.*

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}', \mathcal{A}', (\Gamma \setminus \{P\}) \cup \{P'\}\rangle}\ A_{step}$$

$$\frac{e \text{ is a new external event}}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{!e\}\rangle}\ A_{event}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \overset{\mathsf{bdi}}{\not\longrightarrow}}{\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma\rangle \Longrightarrow \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\}\rangle}\ A_{clean}$$

$$\frac{\Delta = \{\psi_i\theta : P_i\theta \mid e' : \psi_i \leftarrow P_i \in \Pi \wedge \theta = \mathsf{mgu}(e, e')\}}{\langle \mathcal{B}, \mathcal{A}, !e\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle}\ Event$$

$$\frac{\psi_i : P_i \in \Delta \quad \mathcal{B} \models \psi_i\theta}{\langle \mathcal{B}, \mathcal{A}, (\!|\Delta|\!)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P_i\theta \blacktriangleright (\!|\Delta \setminus \{\psi_i : P_i\}|\!)\rangle}\ Sel$$

$$\frac{\mathcal{B} \models \phi\theta}{\langle \mathcal{B}, \mathcal{A}, ?\phi\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle}\ ?$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, +b\rangle \longrightarrow \langle \mathcal{B} \cup \{b\}, \mathcal{A}, nil\rangle}\ +b \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, -b\rangle \longrightarrow \langle \mathcal{B} \setminus \{b\}, \mathcal{A}, nil\rangle}\ -b$$

$$\frac{act' : \psi \leftarrow \Phi^-; \Phi^+ \in \Lambda \quad act'\theta = act \quad \mathcal{B} \models \psi\theta}{\langle \mathcal{B}, \mathcal{A}, act\rangle \longrightarrow \langle (\mathcal{B} \setminus \Phi^-\theta) \cup \Phi^+\theta, \mathcal{A} \cdot act, nil\rangle}\ act$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil\,; P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle}\ Seq_t \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1; P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P'; P_2)\rangle}\ Seq$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \rhd P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P' \rhd P_2)\rangle}\ \rhd \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, (nil \rhd P')\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle}\ \rhd_t$$

$$\frac{P_1 \neq nil \quad \langle \mathcal{B}, \mathcal{A}, P_1\rangle \overset{\mathsf{bdi}}{\not\longrightarrow}}{\langle \mathcal{B}, \mathcal{A}, (P_1 \rhd P_2)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}, \mathcal{A}, P_2\rangle}\ \rhd_f^{bdi}$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, (nil \parallel P)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle}\ \parallel_{t_1} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, (P \parallel nil)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, P\rangle}\ \parallel_{t_2}$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \parallel P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P' \parallel P_2)\rangle}\ \parallel_1 \qquad \frac{\langle \mathcal{B}, \mathcal{A}, P_2\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, (P_1 \parallel P_2)\rangle \longrightarrow \langle \mathcal{B}', \mathcal{A}', (P_1 \parallel P')\rangle}\ \parallel_2$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle \quad \langle \mathcal{B}', \mathcal{A}', P'\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle}\ \mathsf{Plan}$$

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(nil)\rangle \longrightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle}\ \mathsf{Plan}_t$$

$$\frac{\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle}{\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle}\ \mathsf{Plan}_{\mathsf{P}}$$

Figure 3.3: CANPlan's complete set of rules

Proof. Suppose the contrary, i.e., that $\mathsf{Plan}(P)$ is neither executing nor has been successfully executed in the environment-free execution E. Observe from Definition 5 that this can only be true if E is of the form $C_0 = C \cdot \ldots \cdot C_k$ such that $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle \overset{\mathsf{bdi}}{\not\longrightarrow}$, where $\mathsf{Plan}(P_k) = C_k|_P$. Observe, then, that $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \overset{\mathsf{bdi}_k}{\longrightarrow} \langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle$. By the antecedent of rule Plan and from the fact that E is an environment-free execution, we know that $\langle \mathcal{B}, \mathcal{A}, P\rangle \overset{\mathsf{plan}_k}{\longrightarrow} \langle \mathcal{B}_k, \mathcal{A}_k, P_k\rangle \overset{\mathsf{plan}_*}{\longrightarrow} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$, and therefore, that $\langle \mathcal{B}_k, \mathcal{A}_k, P_k\rangle \overset{\mathsf{plan}_*}{\longrightarrow} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$ holds. Then, from Lemma 1, we get that $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle \overset{\mathsf{bdi}_*}{\longrightarrow} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$. Consequently, there exists $\mathcal{B}'', \mathcal{A}'', P''$ such that $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle \overset{\mathsf{bdi}}{\longrightarrow} \langle \mathcal{B}'', \mathcal{A}'', P''\rangle \overset{\mathsf{bdi}_*}{\longrightarrow} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$. Therefore, it cannot be the case that $\langle \mathcal{B}_k, \mathcal{A}_k, \mathsf{Plan}(P_k)\rangle \overset{\mathsf{bdi}}{\not\longrightarrow}$ holds.

The second part follows easily from the fact that $\overset{\mathsf{plan}_*}{\longrightarrow}$ stands for a *finite* chain of transitions: if the agent follows those exact transitions, P will eventually terminate successfully. □

The above theorem is important, as it implies that the programmer can use the new look-ahead construct $\mathsf{Plan}(P)$ in order to avoid – to some extent – failing executions of program P. This is in contrast to the standard BDI execution of P, which may lead to the failure of P due to wrong decisions at choice points. It is important to note, however, that the deliberation construct $\mathsf{Plan}(P)$ is only local in the sense that it does not take into account the potential interactions of P with other concurrent intentions and the environment.

Next, we make clear the relationship between existing HTN semantics and our Plan construct. To do this, we first require that our agents be *bounded*. We say that a CANPlan agent is *bounded* if *(i)* $+b$ and $-b$ programs do not occur in the agent, i.e., the agent only changes its belief base via primitive actions; and *(ii)* all entities belonging to the agent conform to the same constraints and logical language as those of the corresponding entities belonging to the HTN domain. Note that, although $+b$ and $-b$ programs cannot occur in a CANPlan agent, this restriction does not in any way decrease the expressive power of the agent, as $+b$ and $-b$ operations can always be represented using special actions.

Using the definition of a bounded agent, we can now state, formally, the link between the Plan construct of CANPlan and HTN planning of (Erol et al., 1996). First, we show that $\mathsf{Plan}(P)$ does indeed amount to HTN planning on program P. Second, we show that executions of a program $\mathsf{Plan}(P)$ encode HTN plan solutions. Finally, we show that any HTN plan solution can be successfully executed by the BDI execution cycle.

Theorem 2. *For any bounded agent,*

1. $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}}$ *iff* $sol(P, \mathcal{B}, \Pi \cup \Lambda) \neq \emptyset$.
2. $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}_*} \langle \mathcal{B}', \mathcal{A} \cdot act_1 \cdot \ldots \cdot act_k, \mathsf{Plan}(P')\rangle$ *with* $k \geq 1$ *iff there exists a plan* $\sigma \in sol(P, \mathcal{B}, \Pi \cup \Lambda)$, *such that* $\sigma = act_1 \cdot \ldots \cdot act_k \cdot \ldots \cdot act_n$, *for some* $n \geq k$.
3. *If there exists a plan* $\sigma = act_1 \cdot \ldots \cdot act_n \in sol(P, \mathcal{B}, \Pi \cup \Lambda)$, *then* $\langle \mathcal{B}, \mathcal{A}, (act_1; \ldots; act_n)\rangle \xrightarrow{\mathsf{bdi}_*} \langle \mathcal{B}', \mathcal{A} \cdot \sigma, nil\rangle$.

Proof. This is an involved proof showing that plan-type transitions perform no more than the task decomposition done by HTN planners. The proof is based on the translation between BDI domain knowledge (i.e., libraries Π and Λ) and HTN domain knowledge (i.e., the method-library and operator-library), as discussed in Section 3.1. We refer the reader to (Sardina and Padgham, 2010) for the complete proof. □

Therefore, provided we conform to the language of HTN, our deliberator construct Plan provides a built-in HTN planner within the BDI framework. The above theorem is an important practical result as it provides a rationale for using existing HTN systems, such as UMCP, SHOP and SHOP2, within existing BDI implementations such as Jason, Jade and JACK.

Planning versus BDI execution

We conclude this chapter by exploring the differences between the execution of a planning program $\mathsf{Plan}(P)$ and the standard BDI execution of P. In particular, we present two results: first, we show that if the expressivity of a BDI agent is limited in a certain way, then $\mathsf{Plan}(P)$ is nothing more than look-ahead on the standard BDI execution of P; and second, we show that in some BDI program structures, the standard BDI execution of P can find solutions that $\mathsf{Plan}(P)$ cannot find.

To show the first result, we define a CANPlan^{-} agent as a CANPlan agent whose plan language does not include the $\|$ construct. This restriction corresponds to classical BDI agent programming languages such as AgentSpeak, and to *total-order* HTN planners such as SHOP– neither system includes concurrency. Under such restricted, AgentSpeak-like CANPlan agents, any step that can be taken using the planning construct can also be taken using standard BDI execution. This is stated by the following lemma.

Lemma 2. *For every CANPlan^{-} agent, if* $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle$, *then* $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$.

Proof. The only possible rule that could have been used for transition $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P)\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P')\rangle$ is rule Plan. Moreover, from the antecedent of the Plan rule, we know that the following two conditions must hold: *(a)* $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$; and *(b)* $\langle \mathcal{B}', \mathcal{A}', P'\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil\rangle$. Let $L(n)$ be the statement $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$. Using *(a)* and *(b)*, we will prove by induction on the number n of plan-type derivation rules involved in *(a)*, that $L(n)$ holds.

[Base Case: $n = 1$.] If only one derivation rule is involved in *(a)*, then one of the following cases must hold:

1. $P = act \mid ?\phi \mid +b \mid -b$. In this case, $P' = nil$, and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ follows trivially.
2. $P = (nil; P^a)$. In this case, $P' = P^a$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P^a\rangle$ follows trivially.
3. $P = (nil \rhd P^a)$. In this case, $P' = nil$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ follows trivially.
4. $P = !e$. In this case, $P' = (\!|\Delta|\!)$, and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', (\!|\Delta|\!)\rangle$ follows trivially.
5. $P = (\!|\Delta|\!)$. In this case, $P' = P_i\theta \rhd (\!|\Delta \setminus P_i|\!)$ due to rule *Sel*, and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P_i\theta \rhd (\!|\Delta \setminus P_i|\!)\rangle$ also holds.
6. $P = \mathsf{Plan}(nil)$. In this case, $P' = nil$ and $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', nil\rangle$ follow directly from derivation rule Plan_t.

[Induction Hypothesis] Assume that $L(n)$ holds for $n < k$.

[Inductive Step] Suppose $n = k$, that is, k derivation rules are involved in *(a)*. Then, one of the following cases must hold:

1. $P = (P_1; P_2)$. In this case, $P \neq nil$ and $P' = (P_1'; P_2)$, where $\langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P_1'\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$. By the induction hypothesis, we know that $\langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P_1'\rangle$ holds; therefore, $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$ must also hold due to rule *Seq*.
2. $P = (P_1 \rhd P_2)$. In this case, $P \neq nil$ and $P' = (P_1' \rhd P_2)$, where $\langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P_1'\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$. By the induction hypothesis, we know that $\langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P_1'\rangle$ holds; therefore, $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$ must also hold due to rule $\rhd$.
3. $P = \mathsf{Plan}(P_1)$. In this case, $P' = \mathsf{Plan}(P_1')$, where $\langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P_1'\rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$. By the induction hypothesis, we know that $\langle \mathcal{B}, \mathcal{A}, P_1\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}', \mathcal{A}', P_1'\rangle$ holds; therefore, $\langle \mathcal{B}, \mathcal{A}, P\rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', P'\rangle$ must also hold due to rule Plan.

□

Then, it follows that the planning construct is no more than a look-ahead mechanism on top of the BDI execution cycle.

Theorem 3. *Program P simulates program* $\mathsf{Plan}(P)$ *in every CANPlan*$^-$ *agent.*

Proof. Follows directly from Lemma 2. □

On the other hand, when concurrency is allowed in the language, performing planning on a program P may result in more executions than the standard BDI execution of P. It can be shown that executing $\mathsf{Plan}(\mathsf{Plan}(P_1)\|P_2)$, which is equivalent to executing $\mathsf{Plan}(P_1\|P_2)$, is very different from executing $(\mathsf{Plan}(P_1)\|P_2)$. The reason, informally, is that a Plan construct is ignored within the context of another Plan construct—there is no notion of planning within planning.

To understand why a program P does not necessarily simulate program $\mathsf{Plan}(P)$ when concurrency is involved, consider the following example.

Suppose program $P = \mathsf{Plan}(P_1)\|P_2$, where plan-body $P_1 = +r; ?p$ and $P_2 = +p$, and the initial belief base is $\mathcal{B} = \{q\}$. Let us now consider the execution of program P. In the first step, the only applicable rule $\|_2$ (see Figure 3.3) will be used to execute program $+p$. Observe that rule $\|_1$ cannot be used because program $\mathsf{Plan}(P_1)$, i.e., $\mathsf{Plan}(+r; ?p)$, does not have a full HTN decomposition – test condition $?p$ will not succeed in belief base $\mathcal{B} = \{q\}$. Nonetheless, after belief atom p is added to $\mathcal{B}$ in the first step, program $\mathsf{Plan}(+r; ?p)\|nil$ can be successfully executed in the next few execution steps, resulting in program P succeeding.

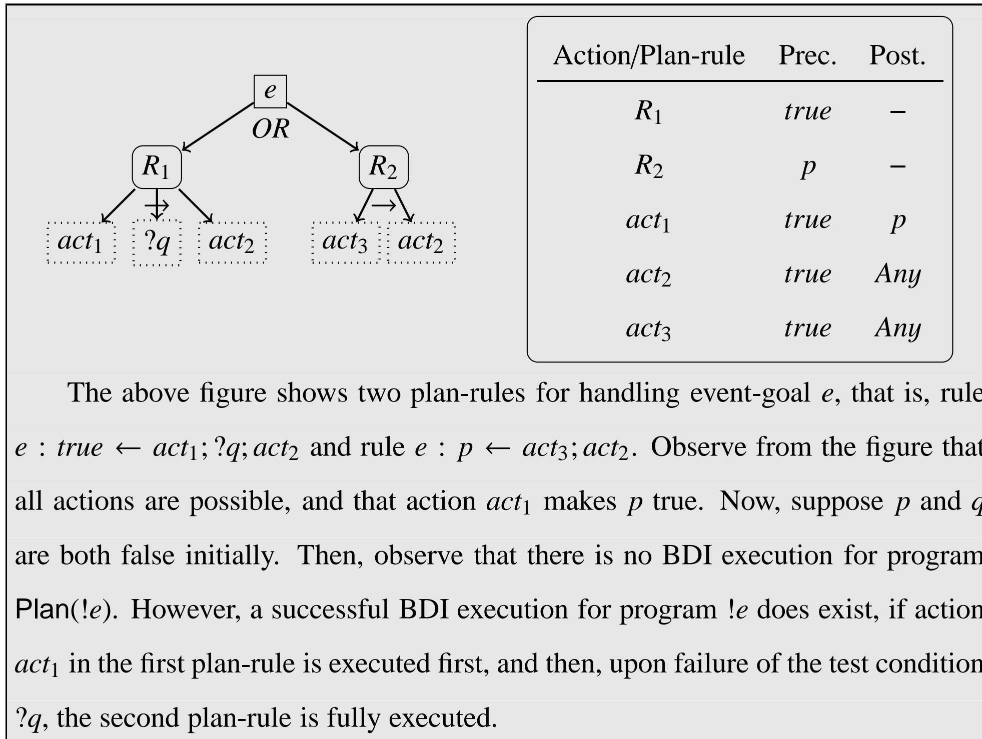
Let us now consider the execution of program $\mathsf{Plan}(P)$, i.e., program $\mathsf{Plan}(\mathsf{Plan}(P_1)\|P_2)$. Unlike we saw earlier, in the first execution step here, both concurrency rules $\|_1$ and $\|_2$ are applicable. Now, suppose rule $\|_1$ is applied first. This will result in program $\mathsf{Plan}(\mathsf{Plan}(nil; ?p)\| + p)$ and belief base $\mathcal{B}_1 = \{q, r\}$. Suppose rule $\|_2$ is applied two steps later (i.e., after the removal of program nil). This will result in program $\mathsf{Plan}(\mathsf{Plan}(?p)\|nil)$ and belief base $\mathcal{B}_2 = \{p, q, r\}$. As one can observe, this will eventually lead to the top-level program $\mathsf{Plan}(P)$ succeeding, because test condition $?p$ within P_1 will succeed in the new belief base $\mathcal{B}_2 = \{p, q, r\}$. Observe, then, that it is not possible to execute program P such that

the resulting evolution of the belief base is identical to the evolution of the belief base resulting from the above execution of program $\mathsf{Plan}(P)$. This is because the first belief addition to $\mathcal{B}$ by the execution of program P is the belief atom p, whereas the first belief addition to $\mathcal{B}$ by the above execution of program $\mathsf{Plan}(P)$ is the belief atom q. Consequently, although both P and $\mathsf{Plan}(P)$ execute successfully, there is a successful execution of $\mathsf{Plan}(P)$ that cannot be obtained by executing P.

Lemma 3. *There exists a program P such that P does not simulate program* $\mathsf{Plan}(P)$.

Proof. Refer to the above example. □

Finally, we show how the BDI execution engine may find successful executions that the planner cannot find. To see why this is the case, let us consider the following example.



Action/Plan-rule	Prec.	Post.
R_1	*true*	–
R_2	p	–
act_1	*true*	p
act_2	*true*	*Any*
act_3	*true*	*Any*

The above figure shows two plan-rules for handling event-goal e, that is, rule $e : true \leftarrow act_1; ?q; act_2$ and rule $e : p \leftarrow act_3; act_2$. Observe from the figure that all actions are possible, and that action act_1 makes p true. Now, suppose p and q are both false initially. Then, observe that there is no BDI execution for program $\mathsf{Plan}(!e)$. However, a successful BDI execution for program $!e$ does exist, if action act_1 in the first plan-rule is executed first, and then, upon failure of the test condition $?q$, the second plan-rule is fully executed.

The following theorem states this result formally.

Theorem 4. *There exists an agent configuration C of the form $\langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{P\}\rangle$ for which there is an execution where P is successfully executed, but such that no execution of $C' = \langle \mathcal{N}, \Lambda, \Pi, \mathcal{B}, \mathcal{A}, \Gamma \cup \{\mathsf{Plan}(P)\}\rangle$ can successfully execute* $\mathsf{Plan}(P)$.

Proof. Refer to the above example. □

As one can observe, the proof relies on the failure handling mechanism built into the BDI execution cycle, as well as on the programmer specifying only a partial BDI plan-library. In fact, if the plan-library in the above example had included the additional plan-rule $e : true \leftarrow act_1; ?p; act_3; act_2$, then the planner would also have found a successful execution. However, since agent programs—that is, plan-libraries—are often developed incrementally and in modules, the above situation could well arise.

It can then be seen that the combined framework, which includes both standard BDI execution as well as local HTN planning, is strictly more general than local HTN planning alone. Moreover, as discussed after Theorem 1, by using the new local planning module, the programmer can, to a certain extent, rule out BDI executions that are bound to fail.

Chapter 4

A First Principles Planning Framework for BDI Systems[†]

In the previous chapter, we provided a principled approach to incorporating HTN planning into the BDI framework, using the domain knowledge contained in the BDI plan-library. However, this approach only allows look-ahead planning to assist in choosing appropriate plans in a given context. It is not possible to construct new BDI plan structures using this approach. In this chapter, we provide an approach which uses first principles planning to find BDI plans that do not currently exist in the plan-library. Such an approach is useful when, for instance, none of the existing plans are applicable for an event-goal, or all applicable plans have failed during execution.

In earlier work on adding first principles planning to BDI-like systems, the focus has been on constructing plans composed of low level steps (e.g., (Despouys and Ingrand, 1999)) or primitive actions (e.g., (Wilkins et al., 1995; Meneguzzi et al., 2004b; Claßen et al., 2007)). In contrast, we are concerned here with the problem of constructing plans composed of hierarchical structures capturing the typical "standard operational procedures" of a BDI agent. We call such plans *abstract-plans*. In more precise terms, abstract-plans are plans in which actions are not primitive but abstract, representing BDI event-goals.

Abstract-plans are particularly appealing in the context of BDI systems because they respect and re-use the procedural domain knowledge that is already available in the system. According to (Kambhampati et al., 1998), the primitive plans that abstract-plans produce preserve a property

[†]Part of the work presented in this chapter has been previously published in (de Silva et al., 2009).

called *user intent*, which according to (Kambhampati et al., 1998) is the property where a primitive plan can be "parsed" in terms of event-goals whose primary effects support the goal state. Another feature of abstract-plans is that they are, like typical BDI plans, flexible and robust: if a primitive step of an abstract-plan happens to fail, another option may be tried to achieve the step.

In order to include event-goals as (abstract) actions in our planning domain, we need to axiomatise event-goals with precondition and postcondition information. We obtain this information offline using an adaptation and extension of the "summary" algorithms of (Clement et al., 2007). In particular, we take as the precondition of an event-goal the disjunction of the context conditions of the associated plans in the plan-library, and we compute the postcondition of an event-goal based on the structure of the event-goal's hierarchy, combined with existing knowledge about the effects of primitive actions. After abstract actions are obtained, we use them at runtime as input for a first principles planner, and we validate abstract-plans found by checking if there is a viable decomposition, which may involve calling the HTN planner. This validation step is necessary due to the representation we use when transforming event-goals into abstract actions. Our overall framework for first principles planning is depicted in Figure 4.1.

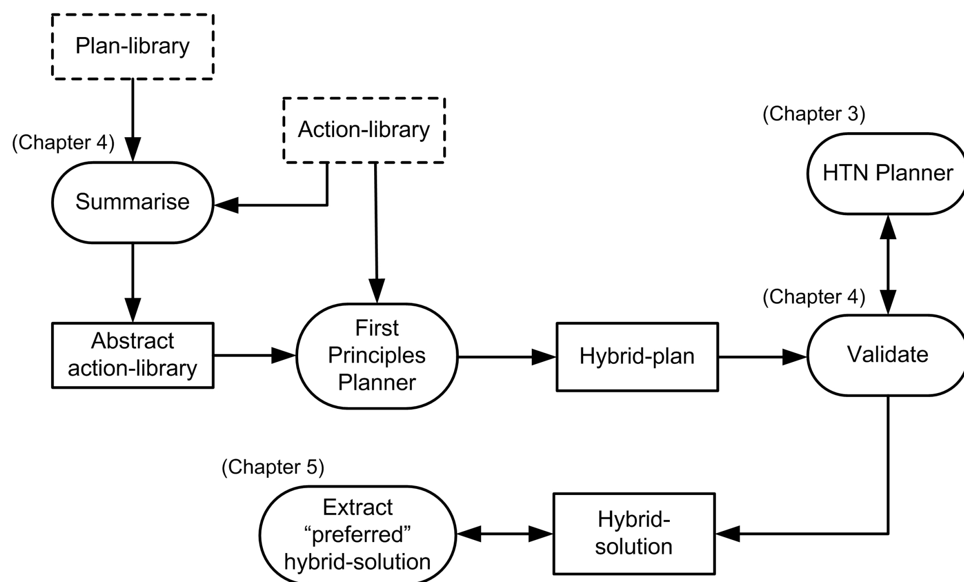


Figure 4.1: The overall framework for first principles planning in BDI systems

As one example of the value of first principles planning in BDI agents, consider Figure 4.2, which shows a Mars Rover agent. The top-level event-goal of the agent is to explore a soil location, and one way of doing this is to navigate to the location

and perform a soil experiment at the location. Observe the following:

1. navigating involves the primitive steps *CalibrateViaGPS* and *Move*;
2. performing a soil experiment involves the non-primitive steps *ObtainSoilResults* and *TransmitSoilResults*;
3. transmitting results to the lander involves the primitive step *SendResults*;
4. obtaining soil results involves the following primitive and non-primitive steps: *PickSoilSample* (primitive), *AnalyseSoil* (non-primitive), and *DropSoilSample* (primitive); and
5. analysing soil involves the primitive steps *GetMoistureContent* and *GetSurfaceImage*.

Note that in order to get an image of the surface of the soil at some location (*GetSurfaceImage*), the rover needs to be at the location. Next, suppose that *PickSoilSample*(dst) places a soil sample in the soil compartment, and that the action's precondition is met only if there is no soil in the compartment. Moreover, suppose the following:

1. R_0 is applicable only if the rover is at *src* and the soil compartment is empty;
2. R_1 is applicable only if the rover is at *src*;
3. R_2 and R_3 are applicable only if the rover is at dst and the soil compartment is empty;
4. R_4 is applicable only if the rover is at dst and there is a soil sample from dst in the soil compartment;
5. R_5 is applicable only if the soil results (i.e., moisture content and surface image) for dst are available; and finally,
6. R_6 is applicable only if the rover is not at *src* and if it has not transmitted results for dst, and R_6 involves calling a first principles planner in order to find a solution that can take the agent from its current state, to the goal state where soil results for dst have been transmitted.

Hence, note that R_6 is used for dealing with failed explorations in which the rover

has successfully navigated from *src*, but has not managed to, for whatever reason, transmit soil results for *dst*.

Now, suppose that the agent starts exploring some soil location *Waypoint*2 from its initial location *Waypoint*1, and that soon after navigating and picking a soil sample from *Waypoint*2, the rover's location (unexpectedly) changes to some other location *Waypoint*3. (This could happen, for instance, if *Waypoint*2 is at the top of a sandy slope; the rover successfully navigates to *Waypoint*2; and then soon after picking a soil sample the rover slips, perhaps due to strong wind, and moves down the slope to *Waypoint*3.) Observe, then, that the second step of R_3 will also fail, as this step requires the rover to be at *Waypoint*2. Consequently, R_0 will fail, and the agent will select the alternative plan-rule R_6, which will call a first principles planner in order to find an abstract-plan for reaching the goal state from the current location *Waypoint*3. One such abstract-plan is shown below.

1. *Navigate*(*Waypoint*3, *Waypoint*2)
2. *AnalyseSoil*(*Waypoint*2)
3. *TransmitSoilResults*(*Waypoint*2)

Observe that this abstract-plan does not involve picking a soil sample after navigating to *Waypoint*2 as the soil compartment already contains a soil sample from *Waypoint*2.

It is important to note that, while in this chapter we provide a framework for planning from first principles using BDI knowledge, we do not, unlike the previous chapter, integrate the framework into a BDI agent-programming language itself, i.e., we do not provide an operational semantics for first principles planning in a BDI agent-programming language. Such a semantics may require the inclusion of new operational rules into the language, along with a new construct such as PlanFP(ϕ), where ϕ is a goal state to achieve. We briefly discuss such a semantics in Chapter 7.

This chapter is organised as follows. First, in Section 4.1, we introduce some preliminary notions, in particular, the notions *hybrid planning problem* and *hybrid solution*. Second, in Section 4.2, we *(i)* define precisely the precondition and postcondition information we want to extract from BDI event-goals for use in first principles planning; *(ii)* provide algorithms for extracting

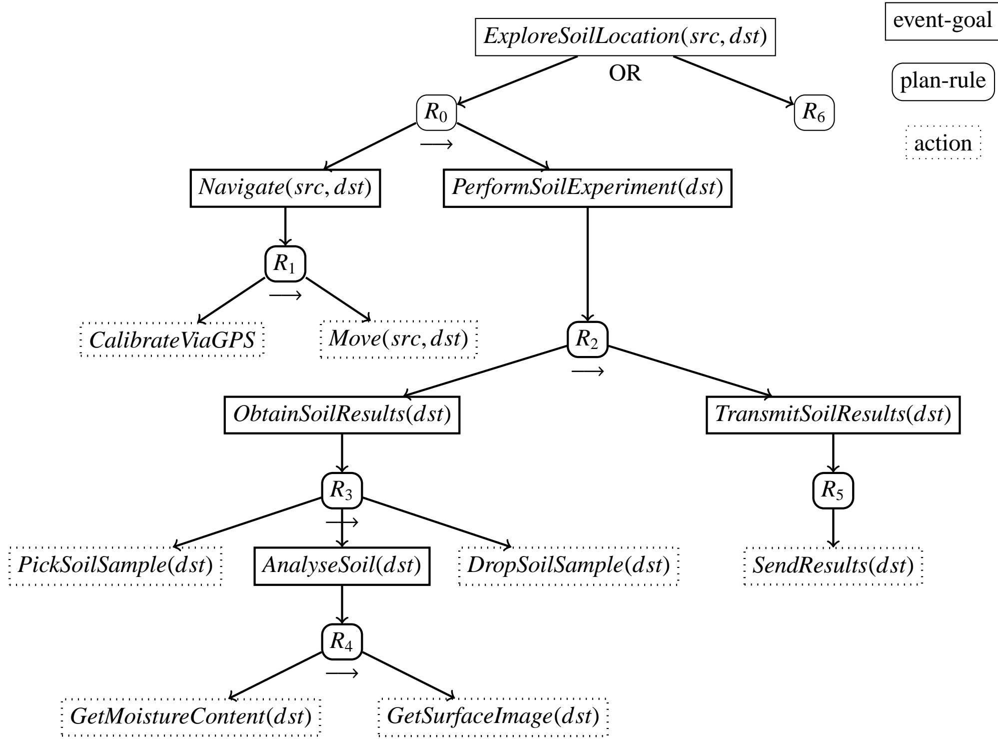


Figure 4.2: A Mars Rover agent. An arrow below a plan-rule indicates that its steps are ordered from left to right.

this information; and *(iii)* discuss the properties of the proposed algorithms. Third, in Section 4.3, we describe how to create abstract actions from the information gathered, and how to obtain an abstract-plan by using these actions for first principles planning. Finally, in Section 4.4, we provide algorithms for validating an abstract-plan found.

4.1 Hybrid Planning

In this section, we introduce some preliminary notions related to *hybrid-plans*, which are more generic than abstract-plans in that they allow the inclusion of primitive actions where necessary. The main notions we introduce are *hybrid planning problem* and *hybrid-solution*.

As mentioned earlier, using event-goals as abstract operators intuitively ensures that resulting hybrid-plans preserve the user intent property (Kambhampati et al., 1998), which is the property where a primitive plan can be "parsed" in terms of event-goals whose *primary* (or intended) effects support the goal state. The primary effects of event-goals (in (Kambhampati et al., 1998)) are

supplied by the programmer. We will start by illustrating the user intent property with an example.

Consider the Mars Rover agent of Figure 4.2. Suppose that the agent, for some reason, invokes the planner from location *Waypoint2*, in order to find a solution for the goal state where the soil compartment is empty, and results from *Waypoint2* for moisture content and surface image are available. Furthermore, suppose that the planner returns the following primitive solution:

1. *PickSoilSample*(*Waypoint*2)
2. *GetMoistureContent*(*Waypoint*2)
3. *GetSurfaceImage*(*Waypoint*2)
4. *DropSoilSample*(*Waypoint*2)

Observe that the action sequence *GetMoistureContent*(*Waypoint*2) · *GetSurfaceImage*(*Waypoint*2) can be parsed by plan-rule R_4, and hence by event-goal *AnalyseSoil*(*Waypoint*2). Observe, further, that the resulting sequence *PickSoilSample*(*Waypoint*2) · *AnalyseSoil*(*Waypoint*2) · *DropSoilSample*(*Waypoint*2) can be parsed by plan-rule R_3, and hence by event-goal *ObtainSoilResults*(*Waypoint*2), whose primary effect — i.e., to have soil results from *Waypoint*2 — supports the goal. Therefore, the primitive solution preserves user intent, as all of its actions can be parsed in terms of event-goals whose primary effects support the goal state. On the contrary, suppose the planner finds the following primitive solution instead, for the same initial state and goal state:

1. *PickSoilSample*(*Waypoint*2)
2. *GetSurfaceImage*(*Waypoint*2)
3. *GetMoistureContent*(*Waypoint*2)
4. *DropSoilSample*(*Waypoint*2)

Notice that, compared with the previous primitive solution, the second and third (independent) actions are in a different order. In this case, action sequence *GetSurfaceImage*(*Waypoint*2) · *GetMoistureContent*(*Waypoint*2) cannot be parsed

by R_4 (and hence event-goal *AnalyseSoil*(*Waypoint*2)) because the ordering of the two actions in the solution does not conform to the preferred ordering of those actions, i.e., the ordering imposed by R_4.[1] Consequently, the primitive solution does not preserve user intent.

Next, we move on to the definitions. Since we have already shown in the previous chapter the relationship between CAN entities and HTN entities, from now on, we will use CAN entities and HTN entities interchangeably for convenience (e.g., we will sometimes refer to event-goals as compound tasks, to plan-rules as methods, and to plan-bodies as task networks).

Intuitively, a hybrid-plan is a collection of operators, where each operator has a precondition and postcondition, but if the operator is abstract, then it is also associated with one or more plan-rules. More specifically, a hybrid-plan is a partially ordered set of primitive tasks and compound tasks; the partial ordering allows tasks in the hybrid-plan to be performed in parallel with other tasks in the plan. Thus, hybrid-plans are what is often referred to as *partially-ordered* plans (Minton et al., 1994). Formally, a *hybrid-plan* $h = [s, \phi]$ is a task network, where ϕ stands for a conjunction of HTN ordering constraints.

In this chapter, we investigate what we refer to as *hybrid planning*, which deals with formulating hybrid-plans that can bring about a certain state of affairs (as in classical planning) by making use of available domain knowledge (as in HTN planning). Hybrid planning is used to find solutions for *hybrid planning problems*. Formally, a *hybrid planning problem* is a tuple $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$, where $\mathcal{I}$ is the initial state, $\mathcal{G}$ is the goal state, and $\mathcal{D}$ is a HTN domain. Hybrid-plans that solve hybrid planning problems are called *hybrid-solutions*. More specifically, a hybrid-solution is a hybrid-plan that can be decomposed using the domain knowledge into a primitive plan that brings about the goal state.

Definition 8. (Hybrid-Solution) A hybrid-plan h is a *hybrid-solution* for a hybrid planning problem $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ iff $sol(h, \mathcal{I}, \mathcal{D}) \cap sol(\mathcal{I}, \mathcal{G}, Op) \neq \emptyset$, that is, if there is a HTN solution for h—a primitive plan—that achieves the goal. A hybrid-plan h is a *strong hybrid-solution* for hybrid planning problem $\mathcal{H}$ iff $sol(h, \mathcal{I}, \mathcal{D}) \subseteq sol(\mathcal{I}, \mathcal{G}, Op)$, that is, if all HTN solutions for h achieves the goal. ■

In this thesis, we will only deal with (weak) hybrid-solutions.

[1] The preference for obtaining moisture content before an image of the soil surface could, for instance, be for mechanical reasons.

Let us illustrate the notions hybrid-plan and hybrid-solution with an example. Consider the Mars Rover agent of Figure 4.2. Suppose that the agent is initially at location *Waypoint*3 with an empty soil container and no soil results, and that the agent wants to find a solution for the following goal: *(i)* results have been transmitted for *Waypoint*3, and *(ii)* the rover is at location *Waypoint*1. Now, consider the hybrid-plan $[s, \phi]$, where:

$$
\begin{aligned}
s &= \{(1 : \mathit{ObtainSoilResults}(\mathit{Waypoint}3)), (2 : \mathit{Navigate}(\mathit{Waypoint}3, \mathit{Waypoint}1)), \\
&\quad (3 : \mathit{TransmitSoilResults}(\mathit{Waypoint}3))\}; \\
\phi &= 1 \prec 2 \wedge 1 \prec 3.
\end{aligned}
$$

Observe that this hybrid-plan is a hybrid-solution, because, according to Figure 4.2, the hybrid-plan can be decomposed into a primitive plan that achieves the goal at hand. Observe, further, that the constraint formula ϕ requires soil results for *Waypoint*3 to be obtained first, and that it does not enforce an ordering between navigating and transmitting results. Therefore, these two tasks can be performed in parallel by executing primitive actions in the following order: *(i) CalibrateViaGPS*, *(ii) SendResults*(*Waypoint*3), and *(iii) Move*(*Waypoint*3, *Waypoint*1).

To obtain hybrid-solutions, we need to first transform event-goals in the BDI domain into a format that is understood by classical planners, that is, into (abstract) planning operators. In the next section, we show how the precondition and postcondition information that is required to create abstract operators is extracted from event-goals.

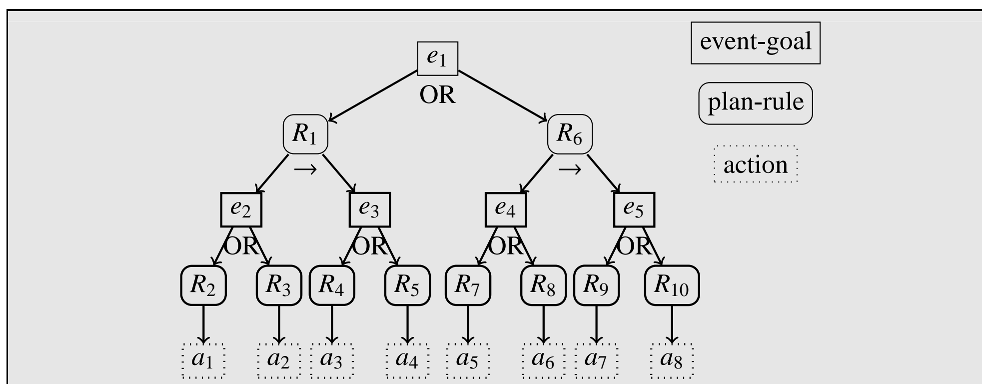
4.2 Creating Abstract Planning Operators

In this section, we define precisely the meaning of the precondition and postcondition of an event-goal, we present algorithms for computing such information, and finally, we discuss the properties of the algorithms presented.

Intuitively, the precondition of an event-goal encodes the conditions under which the event-goal will successfully execute. Since, typically, preconditions are specified on plan-rules as context conditions, obtaining the precondition of an event-goal involves taking the disjunction of the

context conditions of the associated plan-rules. Although postconditions can be manually specified for plan-rules in some BDI systems such as (Despouys and Ingrand, 1999), in most BDI systems and BDI agent programming languages (e.g., (Bordini et al., 2007; Busetta et al., 1999; Hindriks et al., 1999; Rao, 1996; Winikoff et al., 2002)), postconditions cannot be specified for plan-rules or event-goals. More importantly, manually calculating postconditions of event-goals can be troublesome for the programmer, and could lead to erroneous postconditions, which may eventually lead to the planner finding incorrect hybrid-plans. Consequently, we automatically derive the postcondition of an event-goal based on the structure of its hierarchy, combined with knowledge about the effects of primitive actions. To this end, we adapt and extend the "summary" algorithms of (Clement et al., 2007), to allow for the specification of a wider range of BDI plan-libraries, and to allow for variables in literals, event-goals and actions.

Specifically, what we derive from an event-goal's hierarchy are its *definite effects* and *possible effects*. Intuitively, definite effects are those things that are always true after successfully executing any decomposition of plan-rules to achieve the event-goal, and possible effects are those things that are possibly true after executing some decomposition of plan-rules to achieve the event-goal. However, for use as postconditions of abstract operators, we only use the definite effects of the corresponding event-goals. The reason for this is twofold. First, we want an abstract operator to encode, to the extent possible, only the primary effects of the corresponding event-goal, which are supplied by the programmer in the work of (Kambhampati et al., 1998). Our definite effects include all such primary effects (provided the programmer has specified primary effects with care), but they will also include any necessary side effects. Second, we want to avoid an exponential blow-up (with respect to the height of the given plan-library) in the number of abstract operators created. Such a blow-up could happen because possible effects of an event-goal correspond to the different ways in which the event-goal could be decomposed, as shown in the following example.



Consider the BDI hierarchy above. Suppose we want to construct abstract operators from the top-level event-goal e_1, using the possible effects of e_1. Observe, then, that we need to create an abstract operator for the eight possible decompositions of e_1, that is: one operator containing the effects brought about by selecting plan-rules R_1, R_2 and R_4; another containing the effects brought about by selecting plan-rules R_1, R_2 and R_5; another containing the effects brought about by selecting plan-rules R_1, R_3 and R_4; and so on.

It is worth noting that, although it would be possible to create a single operator for e_1 with a disjunctive precondition (i.e., $\phi_{R_1} \wedge (\phi_{R_2} \vee \phi_{R_3}) \wedge (\phi_{R_4} \vee \phi_{R_5}) \vee \phi_{R_6} \wedge \ldots$, where each ϕ_{R_i} is the context condition of plan-rule R_i), or a single operator for e_1 with conditional effects (i.e., where subsets of the postcondition are associated with separate preconditions), such an operator would still be "syntactic sugar," and it would eventually be compiled away into separate operators (Nebel, 2000; Ghallab et al., 2004, pp. 64, 101) as described above.

On the other hand, by taking into account only definite effects, we only need two operators for e_1 (or a single operator with a precondition containing two disjuncts); the postcondition of both operators is the set of definite effects of e_1, and the preconditions of the operators are the context conditions of R_1 and R_6.

Although we do not include the possible effects of event-goals in their corresponding postconditions, we do need to compute them in order to compute the definite effects of event-goals. Moreover, we use the possible effects of event-goals in order to validate hybrid-plans obtained via classical planning, that is, to determine whether a viable decomposition of the plan exists. This validation step is necessary because we only take the definite effects of event-goals as their postconditions, which can potentially lead to situations in which conflicts occur in a hybrid-plan between preconditions of event-goals (abstract operators) and possible effects brought about by decompositions of other event-goals.

4.2.1 Assumptions and Preliminary Definitions

Before we move on to the technical sections, which define precisely the meaning of preconditions, definite effects and possible effects of event-goals, we will present in this section some preliminary notions and state the assumptions we make in this chapter.

As usual, we use $\vec{x}, \vec{y}$, and $\vec{z}$ to denote vectors of distinct variables. Moreover, we use $\vec{t}$ and $\vec{c}$ to denote vectors of (not necessarily distinct) terms and constants, respectively. A subscript n on a vector (e.g., $\vec{x_n}$) denotes a vector of n elements. In this chapter, we assume that, like in AgentSpeak(L) (Rao, 1996; Moreira and Bordini, 2002), the plan-library does not mention any parallel programs $P_1 \parallel P_2$. Since, without parallelism, we would not need to avoid ambiguity in plan-body programs with the use of parenthesis — e.g., given program $P_1; (P_2; P_3)$, program $(P_1; P_2); P_3$, and program $P_1; P_2; P_3$, the BDI execution engine will execute P_1 first, P_2 second and P_3 third in all cases — we assume that parenthesis are not mentioned in plan-bodies.[2]

Second, we assume that the plan-library does not have recursion. Although this may seem limiting, we can overcome this restriction to a certain extent by using first principles planning to emulate recursion, as we will show in Section 6.3. To be more precise regarding our assumption, we define the two notions: *children* and *ranking*. Intuitively, the children of an event-goal e are event-goals mentioned in the plan-rules associated with e.

Definition 9. (Children) The *children* of an event-goal $\hat{e}$ relative to a plan-library Π, denoted $children(\hat{e}, \Pi)$, is defined as follows:[3]

$children(\hat{e}, \Pi) = \{e \mid e' : \psi \leftarrow P \in \Pi, \hat{e}$ and e' have the same type, and $!e$ is mentioned in $P\}$. ■

Intuitively, the ranking of an event-goal is the height of the event-goal in the given hierarchy (plan-library).

Definition 10. (Ranking) A *ranking* for a plan-library Π is a function $\mathcal{R}_\Pi : E_\Pi \mapsto \mathbb{N}_0$ from event-goal types mentioned in Π to natural numbers, such that the following condition holds: for each event-goals $e_1, e_2 \in E_\Pi$ such that e_2 is the same type as some event-goal $e_3 \in children(e_1, \Pi)$, it is the case that $\mathcal{R}_\Pi(e_1) > \mathcal{R}_\Pi(e_2)$. ■

Then, we assume that a ranking exists for the plan-library. We say that $\mathcal{R}_\Pi(e)$ is the *rank* of e in Π, and moreover, given any event-goal $e(\vec{t})$ mentioned in Π, we define $\mathcal{R}_\Pi(e(\vec{t})) = \mathcal{R}_\Pi(e(\vec{x}))$, that is, the rank of an event-goal is equivalent to the rank of its type.

For example, consider the plan-library in Figure 4.2. Possible rank functions of

[2] On the other hand, observe that due to the use of parenthesis in programs $P_1; (P_2 \parallel P_3)$ and $(P_1; P_2) \parallel P_3$, there are executions of the latter program that cannot be obtained by executing the former, because the former program requires P_1 to be (completely) executed first.

[3] Two event-goals e and e' have the same *type* if they have the same predicate symbol and arity.

event-goals mentioned in this plan-library are shown below:

$$
\begin{aligned}
\mathcal{R}_{\Pi}(Navigate(src, dst)) &= 1 \\
\mathcal{R}_{\Pi}(AnalyseSoilSample(dst)) &= 1 \\
\mathcal{R}_{\Pi}(TransmitSoilResults(dst)) &= 1 \\
\mathcal{R}_{\Pi}(ObtainSoilResults(dst)) &= 2 \\
\mathcal{R}_{\Pi}(PerformSoilExperiment(dst)) &= 3 \\
\mathcal{R}_{\Pi}(ExploreSoilLocation(dst)) &= 4
\end{aligned}
$$

Our third assumption is that plan-libraries are *safe*, i.e., all plan-rules in them are written so that whenever the context condition of a plan-rule is met in some belief base, there is at least one successful HTN execution of the corresponding plan-body. Note that this does not imply that the plan-body should be successfully executed by the BDI execution cycle – the BDI execution of the plan-body may still fail if the agent makes a wrong choice.

We define a *successful HTN execution* of a program P (relative to a plan-library and an action-library) as a finite sequence of configurations of the form $C_1 = \langle \mathcal{B}_1, \mathcal{A}_1, P\rangle \cdot \ldots \cdot C_n = \langle \mathcal{B}_n, \mathcal{A}_n, nil\rangle$, such that $C_i \xrightarrow{\text{plan}} C_{i+1}$, for every $i \in \{1, \ldots, n-1\}$. We say that a plan-library Π is *safe* (relative to an action-library) if for all plan-rules $e : \psi \leftarrow P \in \Pi$, ground instances $e\theta$ of e, and belief bases $\mathcal{B}_1$, if $\mathcal{B}_1 \models \psi\theta\theta'$, then there exists a successful HTN execution $C_1 \cdot \ldots \cdot C_n$ of $P\theta\theta'$, where $C_1|_{\mathcal{B}} = \mathcal{B}_1$ (recall $C|_{\mathcal{B}}$ denotes the projection of the belief base in configuration C). This definition is in terms of HTN executions rather than BDI executions because we are not interested in solutions that include BDI-style failure and recovery.

Similarly, we assume that belief operations in plan-libraries are written with appropriate care. For example, a situation should never be reached in which an unground atom is added to the belief base. Specifically, for any finite sequence of configurations of the form $C_1 = \langle \mathcal{B}_1, \mathcal{A}_1, P\rangle \cdot \ldots \cdot C_n = \langle \mathcal{B}_n, \mathcal{A}_n, P'\rangle$, where P, P' are programs and $C_i \longrightarrow C_{i+1}$ for every $i \in \{1, \ldots, n-1\}$, we require that for each $i \in \{1, \ldots, n\}$, $\mathcal{B}_i$ is a set of ground atoms (and hence consistent).

Recall from the *Event* rule of the CAN operational semantics in Figure 3.2 (p. 63) that in order to select a plan-rule $e'(\vec{t'}) : \psi \leftarrow P$ to achieve a given event-goal program $!e(\vec{t})$, the plan-rule must be *relevant* for $!e(\vec{t})$, that is, $e(\vec{t})$ must unify with $e'(\vec{t'})$. This entails, for instance, that if the argument (term) at some index i in $\vec{t'}$ and the argument at the same index in $\vec{t}$ are constants,

then the two arguments must be equivalent. We assume that such requirements on the bindings of arguments in event-goals are encoded in context conditions of associated plan-rules. More precisely, we assume, without loss of generality, that all plan-rules $e(\vec{t}) : \psi \leftarrow P$ in a given plan-library Π are such that $\vec{t}$ is a vector of distinct variables.

Let us illustrate this assumption with an example. Suppose we have the following three plan-rules for travelling to three different destinations from Melbourne:

$Travel(Melb, Syd) : \psi_1 \leftarrow P_1;$
$Travel(Melb, Perth) : \psi_2 \leftarrow P_2;$
$Travel(src, src) : \psi_3 \leftarrow P_3.$

Observe that the third plan-rule handles the situation where the agent is already at the destination (hence, P_3 may be the empty plan-body). Our assumption requires that the above three plan-rules be encoded as follows:

$Travel(x1, y1) : \psi_1 \wedge =(x1, Melb) \wedge =(y1, Syd) \leftarrow P_1;$
$Travel(x2, y2) : \psi_2 \wedge =(x2, Melb) \wedge =(y2, Perth) \leftarrow P_2;$
$Travel(x3, y3) : \psi_3 \wedge =(x3, y3) \leftarrow P_3.$

Observe that the event-goals no longer mention constants, and that the context condition of each plan-rule includes equality predicates. Such predicates within a plan-rule encode the conditions under which arguments of the original associated event-goal (e.g., $Travel(Melb, Syd)$) will unify successfully with those of a given event-goal program for travelling (i.e., an event-goal program with symbol *Travel* and two arguments).

Observe that the encoding of such binding details into the context conditions of plan-rules can be easily automated.

4.2.2 Preconditions and Postconditions

Next, we present the main notions of this chapter, namely, the precondition, *must literals* (definite effects), and *mentioned literals* (possible effects) of a program. We start by defining some basic notions, namely, *atomic program*, *primitive program* and *postcondition* of a primitive program.

Formally, given a program P, we say that P is an atomic program if $P = !e \mid act \mid +b \mid -b \mid ?\phi$, and that P is a primitive program if P is an atomic program that is not an event-goal program. Like the postcondition of a STRIPS action, the *postcondition* of a primitive program consists of the atoms added to and removed from the belief base due to the program's execution. Formally, the postcondition of a primitive program P relative to an action-library Λ, denoted $post(P, \Lambda)$, is the set of literals defined as follows:[4]

$$post(P,\Lambda) = \begin{cases} \emptyset & \text{if } P = ?\phi; \\ \{b\} & \text{if } P = +b; \\ \{\neg b\} & \text{if } P = -b; \\ \Phi^+\theta \cup \{\neg b \mid b \in \Phi^-\theta\} & \text{if } P = act \text{ and } act' : \psi \leftarrow \Phi^+; \Phi^- \in \Lambda \text{ s.t. } act = act'\theta. \end{cases}$$

Observe that test conditions have the empty postcondition since executing them does not result in an update to the belief base. The last condition in the definition states that the postcondition of an action program is the combination of the add list and delete list of the associated action-rule, after applying the appropriate substitution. Note that θ may not be a ground substitution – it may simply be a variable renaming substitution.

We will now move on to the notions precondition, must literal and mentioned literal. Intuitively, the precondition of an event-goal program encodes the conditions under which the program will execute successfully. More specifically, the precondition of an event-goal program is a formula, such that the formula is met in some state if and only if there is at least one successful HTN execution of the program from that state.

Definition 11. (Precondition) A formula ϕ is a precondition of an event-goal program $!e$ (relative to a plan-library and an action-library) if for all ground instances $!e\theta$ of $!e$ and belief bases $\mathcal{B}_1$, it is the case that $\mathcal{B}_1 \models \phi\theta$ holds if and only if there exists a successful HTN execution $C_1 \cdot \ldots \cdot C_n$

[4]Recall that any action program mentioned in a plan-library has exactly one corresponding action-rule in the action-library, and that an action-rule $act : \psi \leftarrow \Phi^+; \Phi^-$ is such that *(i)* act is a symbol followed by a vector of distinct variables, and *(ii)* all variables free in ψ, Φ^+ and Φ^- are also free in act.

of $!e\theta$ such that $C_1|_{\mathcal{B}} = \mathcal{B}_1$. ■

In the above definition and in those that follow, we blur the distinction between event-goals and event-goal *programs* – that is, the definitions in this chapter that apply to event-goal programs also apply to event-goals. Next, we define a must literal of a program as a literal that holds at the end of every successful HTN execution of the program.

Definition 12. (Must Literal) A literal l is a *must literal* of a program P (relative to a plan-library and an action-library) if and only if *(i)* free variables in l are also free in P; and *(ii)* for all ground instances $P\theta$ of P and successful HTN executions $C_1 \cdot \ldots \cdot C_n$ of $P\theta$, it is the case that $C_n|_{\mathcal{B}} \models l\theta$. ■

Note that we require free variables in l to also be free in P so that we can know, given some ground instance of P, exactly which ground instance of l holds in $C_n|_{\mathcal{B}}$. Recall from Definition 7 (p. 70) that all action-rules in any action-library Λ must be consistent. Similarly, because we eventually convert event-goals into abstract planning operators, we need to show that the operators we create will be *consistent*, i.e., that whenever the precondition of an event-goal holds, the set of must literals of the event-goal does not contain conflicting literals. This is stated in the following theorem.

Theorem 5. *Let e be an event-goal, let ϕ be the precondition of e (relative to a plan-library Π and an action-library Λ), and let L^{mu} be a set of must literals of e (relative to Π and Λ). Then, for all ground instances $e\theta$ of e and belief bases $\mathcal{B}$, if $\mathcal{B} \models \phi\theta$, then $L^{mu}\theta$ is consistent.*

Proof. Since $L^{mu}\theta$ is a set of ground literals, observe that if $L^{mu}\theta$ is consistent, then for all literals $l, l' \in L^{mu}$ it is the case that $l\theta \neq \overline{l'}\theta$ (i.e., $l\theta$ is not the complement of $l'\theta$).

We prove the theorem by contradiction. Suppose the theorem does not hold. Then the following conditions must hold: there exists a belief base $\mathcal{B}_1$ and a ground instance $e\theta$ of e, such that $\mathcal{B}_1 \models \phi\theta$ but such that $l\theta = \overline{l'}\theta$ for some $l, l' \in L^{mu}$.

First, notice from the first condition of Definition 12 (Must Literal) that both $l\theta$ and $l'\theta$ are ground. Next, observe from Definition 11 (Precondition) that since $\mathcal{B}_1 \models \phi\theta$ holds, it must also be the case that there exists a successful HTN execution $C_1 \cdot \ldots \cdot C_n$ of $e\theta$ such that $C_1|_{\mathcal{B}} = \mathcal{B}_1$. Therefore, since l is a must literal of e, we know from Definition 12 (Must Literal) that $C_n|_{\mathcal{B}} \models l\theta$. Moreover, since l' is also a must literal of e, it must also be the case that $C_n|_{\mathcal{B}} \models l'\theta$. However,

since we know from our assumption that $l\theta = \overline{l'}\theta$ (i.e., that $l\theta$ is the complement of $l'\theta$), both $C_n|_{\mathcal{B}} \models l\theta$ and $C_n|_{\mathcal{B}} \models \overline{l'}\theta$ cannot hold (recall that $C_n|_{\mathcal{B}}$ is consistent according to our assumption in Section 4.2.1). Therefore, our assumption does not hold. □

Intuitively, a mentioned literal is a literal that is mentioned in the program or in one of its decompositions. Formally, the set of *mentioned literals* of a program P (relative to a plan-library Π and an action-library Λ) is defined as follows:

$$mnt(P) = \begin{cases} mnt(P_1) \cup mnt(P_2) & \text{if } P = P_1; P_2, \\ \{l \mid l \in mnt(P'), e' : \psi \leftarrow P' \in \Pi \text{ and } e = e'\theta\} & \text{if } P = !e, \\ post(P, \Lambda) & \text{if } P = +b \mid -b \mid act \mid ?\phi. \end{cases}$$

Although our notion of a mentioned literal is based on that of a *may summary condition* in (Clement et al., 2007), the latter notion is stronger in that it corresponds to a literal that is met at the end of at least one successful HTN execution of the program in question. Our rationale for using a weaker notion is explained next.

In (Clement et al., 2007), there is a requirement that all possible traces through a goal-plan tree resulting from a plan-body are able to successfully execute. If this is not the case, then the plan-body is said to be inconsistent. However, this requirement is too strong, since it is natural for an event-goal to be used in a plan-body with the expectation that only certain plan-rules of that event-goal will be applicable. This is particularly true if event-goals, and their associated plan-rules, are to be re-usable components. In (Clement et al., 2007), if an event-goal (say e_1) in a plan-body (say P_1) has some plan-rule whose precondition could be clobbered by a plan-rule of some earlier event-goal in P_1, then P_1 is said to be inconsistent, even though there may always be other suitable plan-rules for handling e_1. Thus for a plan-body to be consistent, according to (Clement et al., 2007) every event-goal mentioned in it must be handled only by plan-rules whose preconditions are not made false by effects brought about by plan-rules of other event-goals in the plan-body in question.

For example, consider Figure 4.3, which shows a subset of the plan-library belonging to a simple personal assistant agent. The library shown is for going to work on Fridays, which involves travelling to work, doing work, having after-work drinks, and then travelling home from work. (Note that all details left out in the figure —

e.g., for travelling to work — are not important for this example.) Observe that the *TravelHome* event-goal is a separate "module" — it can be used from within any plan-body. Suppose that the postcondition of having drinks is *HadDrinks*. Suppose, further, that the context condition of the plan-rule for driving home is *HaveCar* (i.e., the car is at the same location as the person) and ¬*HadDrinks*; that the context condition of the plan-rule for travelling home by taxi is *HaveMoneyForTaxi*; and that the context condition of the plan-rule for travelling home by train is *HaveMoneyForTicket*.

Then, observe that plan-rule *GoToWorkFridaysPlan* is inconsistent (according to (Clement et al., 2007)), because literal ¬*HadDrinks* in the context condition of plan-rule *DriveHmPlan* is clobbered by literal *HadDrinks* brought about by action *HaveDrinks*. This is despite the fact that plan-rules *TravelHmByTaxiPlan* and *TravelHmByTrainPlan* may be applicable for event-goal *TravelHome*.

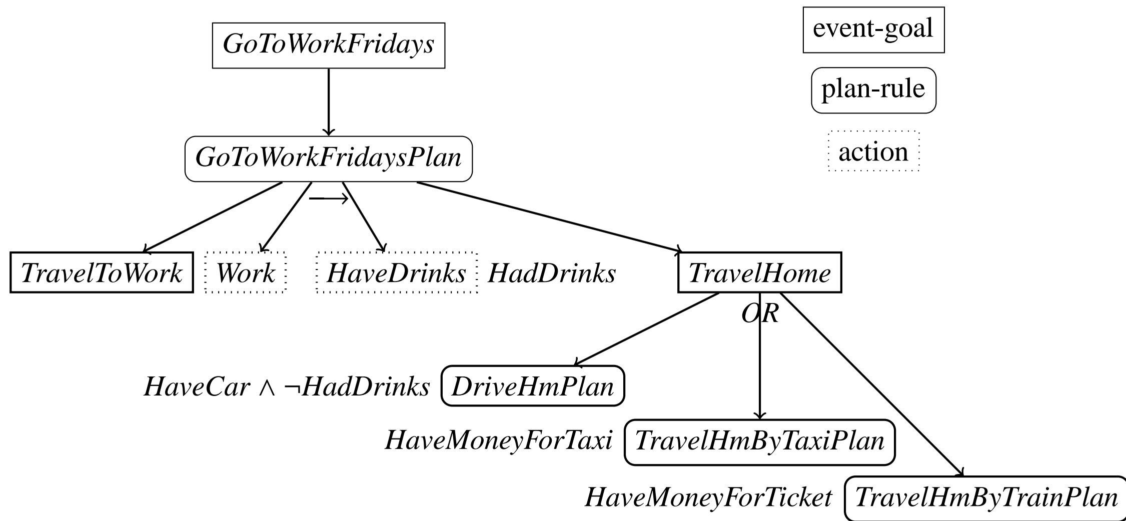


Figure 4.3: An inconsistent plan-rule *GoToWorkFridaysPlan*

We avoid constraining our plan-bodies in this way, although this leads to a weaker notion – *mentioned literals* – than the corresponding definition of a may summary condition in (Clement et al., 2007). In our definition, there can be literals which are mentioned in some plan-body but in fact can never be asserted, due to interactions which ensure that the particular plan-body which asserts that literal can never be applied. For example, while the postconditions of actions in the

plan-body for driving home are mentioned literals of the plan-body, these literals will never be asserted, as the plan-rule for driving home will never be applicable (with respect to the hierarchy shown). We do not define a notion of possible effects that does not take into account literals that will never be asserted because, in practice, recognising such a literal requires reasoning about whether a context condition is definitely clobbered by another plan-rule(s), which, in turn, requires propagating the effects of appropriate plan-rules as a formula (in a similar manner to how we propagate must literals in the algorithms), and then determining whether the context condition is met with respect to the formula. Since this last step amounts to first order entailment, the problem is semi-decidable (Gabbay et al., 1994).

In the next section, we will provide algorithms to obtain preconditions, must literals and mentioned literals of event-goals, given a plan-library and an action-library.

4.2.3 Algorithms

For use in the algorithms that follow, we define a *summary information* of a program as follows.

Definition 13. (Summary Information) A *summary information* of a program P (relative to a plan-library and an action-library) is a tuple $\langle P, \phi, L^{mt}, L^{mn} \rangle$, where ϕ is a precondition of P if P is an event-goal program, and $\phi = \epsilon$ otherwise; L^{mt} is a set of must literals of P; and L^{mn} is a set of mentioned literals of P. ■

In order to compute the must literals of a program, we need to take into account the possibility of literals brought about by the program's execution *conflicting* with other literals brought about by the execution. More specifically, we need to take into account situations in which literals are *definitely undone* (or *must undone*) and *possibly undone* (or *may undone*) within a program. Since, unlike the work of (Clement et al., 2007), we do allow variables in literals, event-goals and actions, finding such conflicts involves reasoning about values assigned at runtime to variables in literals.

For example, take the following plan-body:

$$
\begin{aligned}
&\ldots;\\
&+Colour(Block1, Blue);\\
&?(Block(b) \wedge Colour(b, Blue));\\
&-Colour(b, Blue);\\
&+Colour(b, Red).
\end{aligned}
$$

> This plan adds a belief that $Block1$ is blue, then binds the variable b to some blue block (possibly $Block1$), removes the belief that b is blue and adds the belief that b is red. The literals $Colour(Block1, Blue)$ and $Colour(b, Red)$ are both asserted in the body of this plan. However, only $Colour(b, Red)$ can be considered a definite effect, as $Colour(Block1, Blue)$ will be true only if b is not bound to $Block1$. Therefore, $Colour(Block1, Blue)$ is only a possible effect.

We say that a literal l is *must undone* in some program P if the negation of the literal is a must literal of some atomic program mentioned in P. Note that, although l can be *any* literal and P can be *any* program (i.e., sequence of atomic programs), we will only need to use this definition to determine whether a literal belonging to some atomic program P' in a plan-body is must undone in the sequence of atomic programs *after* P'. Formally, given a program P and the set Δ of summary information of all atomic programs mentioned in P, a literal l is *must undone* in P relative to Δ, denoted Must-Undone(l, P, Δ), if there exists an atomic program P' mentioned in P and a literal $l' \in L^{mt}$, with $\langle P', \phi, L^{mt}, L^{mn} \rangle \in \Delta$, such that $l = \overline{l'}$, i.e., l is the complement of l'.[5] Similarly, we say that a literal l is *may undone* in a program P if there is a literal l' that is a mentioned (or must) literal of some atomic program in P such that l' may become the negation of l due to variable substitutions at runtime. More precisely, given a program P and the set Δ of summary information of all atomic programs mentioned in P, a literal l is *may undone* in P relative to Δ, denoted May-Undone(l, P, Δ), if there exists an atomic program P' mentioned in P, substitutions θ, θ', and a literal $l' \in L^{mn}$, with $\langle P', \phi, L^{mt}, L^{mn} \rangle \in \Delta$, such that $l\theta = \overline{l'}\theta'$.

Next, we move on to the main algorithms for computing the summary information of programs, that is, algorithms 4.1, 4.2 and 4.3. We will use Figure 4.4 and Table 4.1 as a running example.

Algorithm 4.1: Given a plan-library and an action-library, Algorithm 4.1 computes the summary information of each event-goal mentioned in the plan-library. In a nutshell, the algorithm works bottom up, by summarising first the leaf-level entities of the plan-library, that is, primitive programs (line 1), and then repetitively summarising plan-bodies using Algorithm 4.2, and event-goals using Algorithm 4.3, in increasing order of their levels of abstraction (lines 3-8). The algorithm terminates after all top level event-goals (i.e., those with the highest rank) have been

[5]The *complement* of a literal $l \in \{a, \neg a\}$ is a if $l = \neg a$, and $\neg a$ otherwise.

Algorithm 4.1 Summarise(Π, Λ)

Input: Plan-library Π and action-library Λ, where a ranking exists for Π.
Output: Set Δ of summary information of event types mentioned in Π.

1: $\Delta \Leftarrow \{\langle P, \epsilon, post(P, \Lambda), post(P, \Lambda)\rangle \mid P$ is a primitive program mentioned in $\Pi\}$
// Summarising primitive programs; recall $post(P, \Lambda)$ is a set of literals
2: $E \Leftarrow \{e(\vec{x}) \mid e$ is an event-goal mentioned in $\Pi\}$ // Construct the set of event types in Π
3: **for** $i \Leftarrow min(\{\mathcal{R}_\Pi(e) \mid e \in E\})$ **to** $max(\{\mathcal{R}_\Pi(e) \mid e \in E\})$ **do** // Recall $\mathcal{R}_\Pi(e)$ is the rank of e
4: **for** each $e \in E$ such that $\mathcal{R}_\Pi(e) = i$ **do**
5: $\Delta \Leftarrow \Delta \cup \{$Summarise-Plan-Body$(P, \Pi, \Lambda, \Delta) \mid e' : \psi \leftarrow P \in \Pi, e' = e\theta\}$
// Summary information of event-goals mentioned in P is available due to ranking
6: $\Delta \Leftarrow \Delta \cup \{$Summarise-Event$(e, \Pi, \Delta)\}$
7: **end for**
8: **end for**
9: **return** $\Delta \setminus \{u \mid u \in \Delta, u$ is not the summary information of an event-goal$\}$

summarised.

In lines 2-8, all event-goal types mentioned in the plan-library are obtained and then summarised in increasing order of their rank. This way, there is a guarantee that whenever the summary information of an event-goal or plan-body needs to be computed, all the summary information of associated less abstract entities has already been computed. Finally, before returning the computed set Δ in line 9, we remove all the summary information tuples of entities other than event-goals, since we are only interested in the summary information of event-goals.

Observe that, although according to Definition 12 (Must Literal), *any* literal that holds at the end of all successful executions of a program is considered a must literal of the program (i.e., even if a literal holds at the end of such an execution only due to preconditions that require it to hold), the algorithm only classifies as must literals those that are actually *brought about* during the program's execution, i.e., literals in postconditions of primitive programs. This is because our aim is to create operators from event-goals, and consequently, literals that are required to hold due to preconditions do not need to be added to postconditions of operators.

For example, consider an event-goal e that is handled by one plan-rule $e : p \leftarrow act$, where the operator associated with action act does not mention proposition p. Observe that p is a must literal of e because it holds at the end of all successful executions of e. However, since p is not actually brought about by the execution of e — i.e., p is only required by some precondition — we do not include p in the postcondition of the operator corresponding to e.

Algorithm 4.2 Summarise-Plan-Body($P, \Pi, \Lambda, \Delta_{in}$)

Input: Plan-body P; plan-library Π, where a ranking exists for Π; action-library Λ; and the set Δ_{in} of summary information of *(i)* primitive programs mentioned in P, and *(ii)* event types mentioned in P.

Output: The summary information of P.

1: $\Delta \Leftarrow \Delta_{in} \cup \{\langle !e(\vec{x}), \phi, L^{mt}, L^{mn}\rangle\theta \mid !e(\vec{t}) \text{ occurs in } P, \langle e(\vec{x}), \phi, L^{mt}, L^{mn}\rangle \in \Delta_{in}, e(\vec{t}) = e(\vec{x})\theta\}$

// Variables in L^{mn} must be renamed appropriately

2: Suppose $P = P_1; P_2; \ldots; P_n$

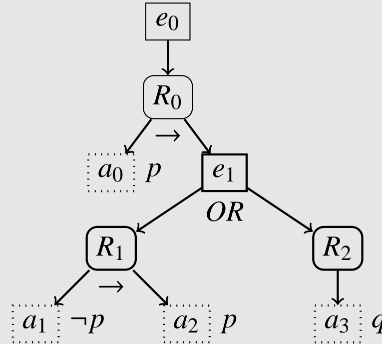
3: $L_P^{mt} \Leftarrow \{l \mid l \in L^{mt}, \langle P_i, \phi, L^{mt}, L^{mn}\rangle \in \Delta, i \in \{1, \ldots, n\}, \neg\text{May-Undone}(l, P_{i+1}; \ldots; P_n, \Delta)\}$

4: $L_P^{mn} \Leftarrow$
$\{l \mid l \in L^{mt} \cup L^{mn}, \langle P_i, \phi, L^{mt}, L^{mn}\rangle \in \Delta, i \in \{1, \ldots, n\}, \neg\text{Must-Undone}(l, P_{i+1}; \ldots; P_n, \Delta)\}$

5: **return** $\langle P, \epsilon, L_P^{mt}, L_P^{mn}\rangle$

Algorithm 4.2: This algorithm summarises the given plan-body with respect to the given plan-library, action-library, and set Δ_{in} of summary information. The algorithm first obtains the summary information of each event-goal program mentioned in the plan-body, from the already available summary information in Δ_{in} of the corresponding event-goal types (line 1). Next, the algorithm computes the set of must literals (L_P^{mt}) and the set of mentioned literals (L_P^{mn}) of the given plan-body P, by determining, from the must and mentioned literals of atomic programs mentioned in P, which literals will definitely be met and which literals will possibly be met on the successful executions of P (lines 3 and 4). More specifically, a must literal l of an atomic program P_i mentioned in $P = P_1; \ldots; P_n$ is considered a must literal of P only if l is not may (or must) undone in $P_{i+1}; \ldots; P_n$ (line 3). If this is not the case, then l is considered a mentioned literal of P, provided l is not must undone in $P_{i+1}; \ldots; P_n$ (line 4). The reason we do not summarise mentioned literals that are must undone is to avoid losing completeness. This is shown in the following example.

Suppose the algorithm does summarise mentioned literals that are must undone. Next, consider the plan-library below. Observe the following: the postcondition of a_0 and a_2 is p; the postcondition of a_1 is $\neg p$; and the postcondition of a_3 is q.



Furthermore, observe that:

1. p is a must literal of R_1;
2. q is a must literal of R_2;
3. e_1 has no must literals—there are no literals that are guaranteed to hold irrespective of the plan-rule selected to achieve e_1;
4. $\neg p$ (brought about by a_1) is a mentioned literal of R_1; and
5. $\neg p$ is also mentioned literal of e_1.

Observe, further, that, according to the algorithm, the literal p brought about by a_0 is not a must literal of R_0 because it may be undone by mentioned literal $\neg p$ of e_1. However, in reality, although R_1 does bring about literal $\neg p$, action a_2 of R_1 later adds p. This means that p is indeed a must literal of R_0. The algorithm will recognise this (i.e., it will be more complete) if *(i)* the algorithm recognises that mentioned literal $\neg p$ is must undone in R_1, and *(ii)* the algorithm excludes $\neg p$ from the set of mentioned literals of R_1.

Then, the literals added to set L_P^{mn} by the algorithm are not just mentioned literals, but what we call *may* literals, that is, mentioned literals of atomic programs occurring in the given plan-body that are not must undone later in the plan-body. It is important to note, however, that our may literals are still a weaker notion than the corresponding notion of a may summary condition in (Clement et al., 2007), because it is still possible that our may literals are never asserted, due to interactions which ensure that the particular plan-body which asserts a may literal can never be applied.

To illustrate how Algorithm 4.2 works, consider Figure 4.4 and Table 4.1. Observe that literals $HaveMoistureContent(dst)$ and $HaveParticleSize(dst)$ are must literals of plan-body P_5 because *(i)* they are must literals of primitive actions $GetMoistureContent(dst)$ and $GetSoilParticleSize(dst)$, and *(ii)* they are neither must undone nor may undone in P_5.

Next, consider plan-body P_4. Observe that, although literal $HaveSoilSample(dst)$ is a must literal of primitive action $PickSoilSample(dst)$, the literal is must undone by the last primitive action $DropSoilSample(dst)$ of P_4.

Therefore, $HaveSoilSample(dst)$ is neither a may nor must literal of the plan-body. Literal $\neg HaveSoilSample(dst)$ is a must literal of P_4, along with the must literals $HaveMoistureContent(dst)$ and $HaveParticleSize(dst)$ belonging to event-goal $AnalyseSoilSample(dst)$.

Finally, consider plan-body P_0. Observe that literal *Calibrated* is a may literal of the plan-body because the literal is a may literal of event-goal $Navigate(src, dst)$, and the literal is not must undone in $PerformSoilExperiment(dst)$. On the other hand, observe that although literal $At(dst)$ (respectively $\neg At(src)$) is a must literal of event-goal $Navigate(src, dst)$, this literal is only a may literal of P_0, because $\neg At(dst)$ (respectively $At(ldr)$) is a may literal of event-goal $PerformSoilExperiment(dst)$, and consequently, $At(dst)$ (respectively $\neg At(src)$) is may undone in $PerformSoilExperiment(dst)$.

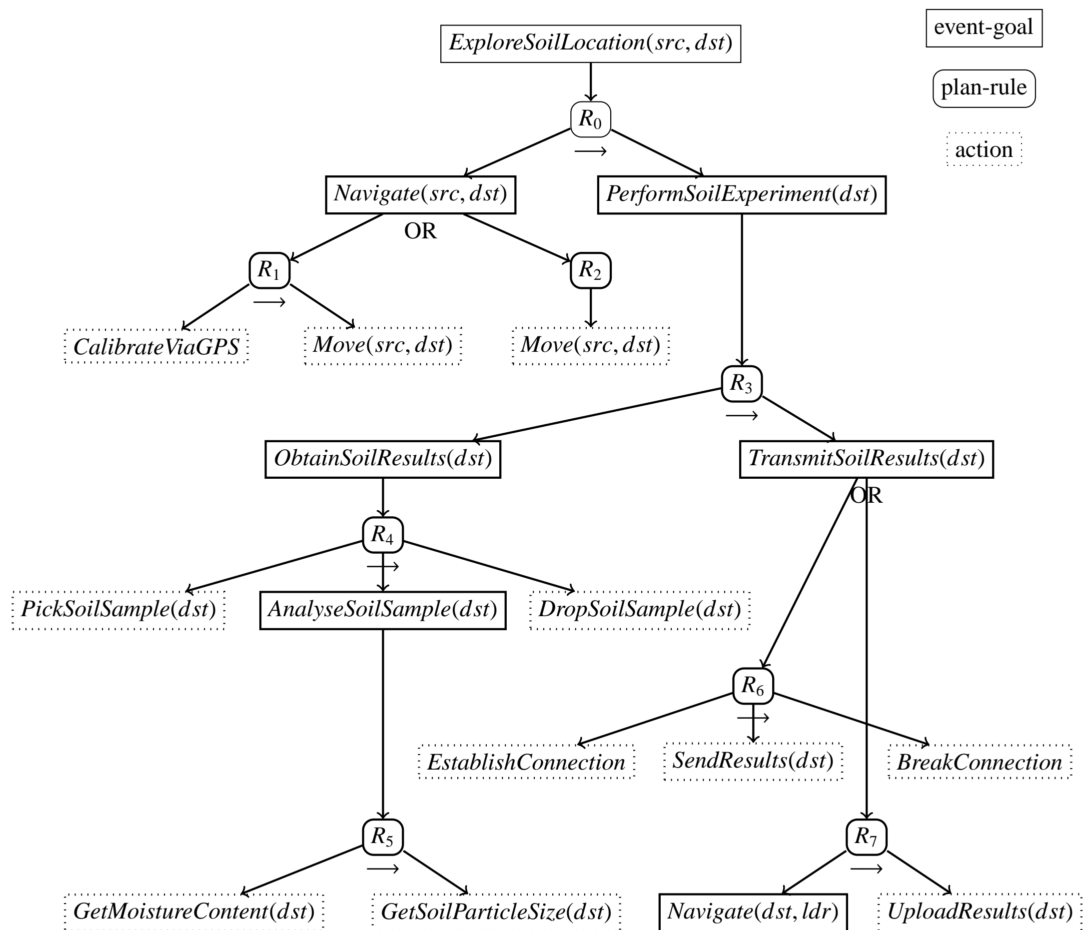


Figure 4.4: A slightly modified and extended version of the Mars Rover agent of Figure 4.2. This version has options for navigating and transmitting results, and if the lander is not within range, transmitting involves navigating to the lander and uploading results.

Program	Must Literals	May Literals
$CalibrateViaGPS$	CA	-
$Move(src, dst)$	$\neg At(src), At(dst)$	-
$PickSoilSample(dst)$	$HSS(dst)$	-
$DropSoilSample(dst)$	$\neg HSS(dst)$	-
$GetMoistureContent(dst)$	$HMC(dst)$	-
$GetSoilParticleSize(dst)$	$HPS(dst)$	-
$EstablishConnection$	CE	-
$SendResults(dst)$	$RT(dst)$	-
$BreakConnection$	$\neg CE$	-
$UploadResults(dst)$	$RT(dst)$	-
P_1	$\neg At(src), At(dst), CA$	-
P_2	$\neg At(src), At(dst)$	-
P_5	$HMC(dst), HPS(dst)$	-
P_4	$HMC(dst), HPS(dst), \neg HSS(dst)$	-
P_6	$RT(dst), \neg CE$	-
P_7	$\neg At(dst), At(ldr), RT(dst)$	CA
P_3	$RT(dst), HMC(dst), HPS(dst),$ $\neg HSS(dst)$	$\neg CE, \neg At(dst),$ $At(ldr), CA$
P_0	$RT(dst), HMC(dst), HPS(dst),$ $\neg HSS(dst)$	$\neg CE, At(dst), \neg At(dst),$ $At(ldr), CA, \neg At(src)$
$Navigate(src, dst)$	$\neg At(src), At(dst)$	CA
$AnalyseSoilSample(dst)$	Same as P_5	-
$ObtainSoilResults(dst)$	Same as P_4	-
$TransmitSoilResults(dst)$	$RT(dst)$	$\neg CE, \neg At(dst)$ $At(ldr), CA$
$PerformSoilExperiment(dst)$	Same as P_3	Same as P_3
$ExploreSoilLocation(src, dst)$	Same as P_0	Same as P_0

Table 4.1: Must literals and may literals of atomic programs and plan-bodies of Figure 4.4. Note that the rightmost column only shows the may literals that are not also must literals. Abbreviations used in the table are as follows: CA = *Calibrated*, HSS = *HaveSoilSample*, HMC = *HaveMoistureContent*, HPS = *HaveParticleSize*, CE = *ConnectionEstablished*, and RT = *ResultsTransmitted*. Each P_i is the plan-body corresponding to plan-rule R_i in the figure.

Algorithm 4.3: This algorithm summarises the given event-goal with respect to the given plan-library and set Δ of summary information. In a nutshell, in lines 3-8, the algorithm computes the mentioned literals of the event-goal, and the algorithm obtains the precondition of the event-goal as the disjunction of the context conditions of all associated plan-rules. Next, the algorithm obtains the must and may literals of the event-goal by respectively taking the intersection of the must literals of associated plan-rules (lines 6 and 10), and the union of the may literals of associated plan-rules (line 7).

More specifically, in line 5, the current plan-rule's context condition is added as a disjunction to the current value of the event-goal's precondition, after performing the appropriate variable renamings. In line 10, the must literals of the event-goal are taken as the must literals that are common across the plan-bodies of all plan-rules handling the event-goal, since such literals are guaranteed to be true after any successful execution of the event-goal, irrespective of the plan-rule chosen to achieve it.

Algorithm 4.3 Summarise-Event($e(\vec{x}), \Pi, \Delta$)

Input: Event-goal type $e(\vec{x})$; plan-library Π, where a ranking exists for Π; and the set Δ of summary information of plan-bodies of plan-rules $e' : \psi \leftarrow P \in \Pi$ such that $e' = e(\vec{x})\theta$.
Output: The summary information of $e(\vec{x})$.

1: $\phi \Leftarrow false$
2: $L^{mt}, L^{mn}, S \Leftarrow \emptyset$ // L^{mt}, L^{mn} are sets of literals and S is a set of sets of literals
3: **for** each $e(\vec{y}) : \psi \leftarrow P \in \Pi$ such that $e(\vec{x}) = e(\vec{y})\theta$ **do**
4: // Variables in ψ and tuple $\langle P, \epsilon, L_P^{mt}, L_P^{mn}\rangle \in \Delta$ need to be renamed appropriately
5: $\quad\phi \Leftarrow \phi \vee \psi\theta$
6: $\quad S \Leftarrow S \cup \{L_P^{mt}\theta\}$, where $\langle P, \epsilon, L_P^{mt}, L_P^{mn}\rangle \in \Delta$
7: $\quad L^{mn} \Leftarrow L^{mn} \cup L_P^{mn}\theta$
8: **end for**
9: **if** $S \neq \emptyset$ **then** // Obtain the must literals of $e(\vec{x})$
10: $\quad L^{mt} \Leftarrow \bigcap S$
11: $\quad L^{mt} \Leftarrow \{l \mid l \in L^{mt},$ variables occurring in l also occur in $e(\vec{x})\}$
12: **end if**
13: **return** $\langle e(\vec{x}), \phi, L^{mt}, L^{mn}\rangle$

For example, consider Figure 4.4 and Table 4.1. Observe that the only must literal in common between plan-bodies P_6 and P_7 is $RT(dst)$. This literal is also a must literal of event-goal $TransmitSoilResults(dst)$, because there is a guarantee that a ground instance of this literal will be true on the successful execution of $TransmitSoilResults(dst)$, irrespective of whether P_6 or P_7 is chosen to achieve the

event-goal. Observe that all other (must or may) literals of the two plan-bodies are may literals of the event-goal.

4.2.4 An Exploration of Soundness and Completeness

In this section, we will explore the properties of the algorithms presented in the previous section. In particular, we will show that *(i)* whenever Algorithm 4.1 (Summarise) classifies a literal as a must literal, this is indeed the case; *(ii)* the algorithm correctly computes the preconditions of event-goals; and that *(iii)* the algorithm terminates. Moreover, we will give some insight into the situations in which the algorithm is not complete. In what follows, we assume that any given plan-library Π is such that a ranking exists for Π.

Soundness and termination

The lemmas that follow rely on the following definition of what it means for a set of literals to *capture* a program. Intuitively, a set of literals captures a program if any literal resulting from any successful execution of the program is in the set.

Definition 14. (Capturing a Program) Let P be a program and L be a set of literals. Set L captures P if and only if for any ground instance P^g of P, successful HTN execution $C_1 \cdot \ldots \cdot C_n$ of P^g, and ground literal l such that $C_1|_{\mathcal{B}} \not\models l$ and $C_n|_{\mathcal{B}} \models l$, it is the case that there is a literal $l' \in L$ such that $l = l'\theta$, for some substitution θ. ■

Observe, then, that the (full) set of mentioned literals of a program captures the program. We start by showing that the computation in Algorithm 4.1 of the must literals of primitive programs is sound. The proofs for the lemmas in this section can be found in Appendix A.2.

Lemma 4. *Let P be a primitive program (i.e., $P = ?\phi \mid +b \mid -b \mid act$) mentioned in a plan-library Π, and let Λ be an action-library. Given Π and Λ as input for Algorithm 4.1, at the end of line 1 of the algorithm, there exists exactly one tuple $\langle P, \epsilon, L^{mt}, L^{mn} \rangle \in \Delta$ such that the tuple is the summary information of P, and L^{mn} captures P.*

The following lemma states that Algorithm 4.2 (Summarise-Plan-Body) is sound, that is, whenever it classifies a literal as a must literal this is indeed the case.

Lemma 5. *Let P be a plan-body mentioned in a plan-library Π, and let Λ be an action-library. Let Δ_{in} be a set of tuples such that:*

1. *for each primitive program P' mentioned in P, there is exactly one tuple $\langle P', \epsilon, L^{mt}, L^{mn}\rangle \in \Delta_{in}$ such that the tuple is the summary information of P', and L^{mn} captures P'; and*

2. *for each event-goal program $!e$ mentioned in P, there exists exactly one tuple $\langle e', \phi, L^{mt}, L^{mn}\rangle \in \Delta_{in}$ such that the tuple is the summary information of e', event-goal e' is the event type of e, and L^{mn} captures e'.*

Finally, let tuple $\langle P', \epsilon, L^{mt}, L^{mn}\rangle$ = Summarise-Plan-Body$(P, \Pi, \Lambda, \Delta_{in})$. Then, it is the case that the tuple is the summary information of P, and L^{mn} captures P.

Next, we move on to Algorithm 4.3 (Summarise-Event). The following two lemmas state that this algorithm is sound, that is, whenever it classifies a literal as a must literal this is indeed the case, and that the algorithm correctly computes summary preconditions of event-goals.

Lemma 6. *Let e be the event type of some event-goal mentioned in a plan-library Π. Let Δ be a set of tuples such that for each plan-rule $e' : \psi \leftarrow P \in \Pi$, where e and e' have the same type, there exists exactly one tuple $\langle P, \epsilon, L^{mt}, L^{mn}\rangle \in \Delta$ such that the tuple is the summary information of P, and L^{mn} captures P. Finally, let tuple $\langle e', \phi, L^{mt}, L^{mn}\rangle$ = Summarise-Event(e, Π, Δ). Then, it is the case that $e = e'$, L^{mt} is a set of must literals of e, and that L^{mn} captures e.*

Lemma 7. *Let e be the event type of some event-goal mentioned in a plan-library Π, and let $\langle e', \phi, L^{mt}, L^{mn}\rangle$ = Summarise-Event(e, Π, Δ), for some Δ. Then, ϕ is the precondition of e.*

Finally, the following two theorems state that the main algorithm – Algorithm 4.1 – is sound, and always terminating. They rely on the two lemmas given below.

Lemma 8. *Algorithm 4.2 always terminates.*

Lemma 9. *Algorithm 4.3 always terminates.*

Theorem 6. *Algorithm 4.1 always terminates.*

Proof. Follows trivially from the fact that, from Lemmas 9 and 8, lines 6 and 5 (respectively) always terminates. □

Theorem 7. *Let Π be a plan-library, Λ be an action-library, e be an event type mentioned in Π, and let Δ_{out} = Summarise(Π, Λ). Then, there is exactly one tuple $\langle e, \phi, L^{mt}, L^{mn}\rangle \in \Delta_{out}$ such that the tuple is the summary information of e, and L^{mn} captures e.*

Proof. We prove this by induction on the rank of e in Π.

[Base Case] Let e be an event of rank 0 in Π, that is, $\mathcal{R}_\Pi(e) = 0$. Observe from the definition of a ranking for a plan-library (Definition 10) that if $\mathcal{R}_\Pi(e) = 0$, then $children(e, \Pi) = \emptyset$. This entails that for all plan-rules $e' : \psi \leftarrow P \in \Pi$ such that $e = e'$, no event-goals are mentioned in P. If e has no associated plan-rules, then observe that the call to procedure Summarise-Event(e, Π, Δ) in line 6 of procedure Summarise(Π, Λ) returns tuple $\langle e, \mathit{false}, \emptyset, \emptyset\rangle$, which is indeed the summary information of $!e$, and $\emptyset$ captures $!e$ (see that $!e$ has no successful executions).

Consider the case where there are one or more plan-rules $e' : \psi \leftarrow P \in \Pi$ such that e and e' have the same type, but such that no event-goals are mentioned in the corresponding plan-bodies. Let P_{all} denote the (non-empty) set of plan-bodies corresponding to all such plan-rules. Then, we know from Lemma 4 that, due to line 1 in the algorithm, there is exactly one tuple $\langle P', \epsilon, L^{mt}_{P'}, L^{mn}_{P'}\rangle \in \Delta$ for each primitive program P' mentioned in each plan-body $P \in P_{all}$, such that the tuple is the summary information of P', and $L^{mn}_{P'}$ captures P'.

Next, observe that before reaching line 6 of procedure Summarise-Event(Π, Λ), procedure Summarise-Plan-Body(P, Π, Λ, Δ) is called in line 5 for each plan-body $P \in P_{all}$. Then, from Lemma 5, we know that, on the completion of line 5, there is exactly one tuple $\langle P, \epsilon, L^{mt}_P, L^{mn}_P\rangle \in \Delta$ for each plan-body $P \in P_{all}$ such that the tuple is the summary information of P, and L^{mn}_P captures P. Finally, from Lemmas 6 and 7, we can conclude that on the completion of line 6 of procedure Summarise-Event(Π, Λ) (i.e., after calling procedure Summarise-Event(e, Π, Δ)), there is exactly one tuple $\langle e, \phi_e, L^{mt}_e, L^{mn}_e\rangle \in \Delta$ such that the tuple is the summary information of e, and that L^{mn}_e captures e. Therefore, the theorem holds.

[Induction Hypothesis] Assume that the theorem holds if $\mathcal{R}_\Pi(e) \leq k$, for some $k \in \mathbb{N}_0$.

[Inductive Step] Suppose $\mathcal{R}_\Pi(e) = k+1$. Let $P_{all} = \{P \mid e' : \psi \leftarrow P \in \Pi, e \text{ and } e' \text{ have the same type}\}$. There are three cases to consider. First, there is no plan-body $P \in P_{all}$ such that there is an event-goal mentioned in P (i.e., all plan-bodies in P_{all} mention only primitive programs). Thus, $children(e, \Pi) = \emptyset$ (Definition 9). The proof for this case is the same as the proof for the Base Case above. The second case is that $P_{all} = \emptyset$ (i.e., there is no plan-rule $e' : \psi \leftarrow P \in \Pi$ such that e and e' have the same type). The proof for this case is also the same as the proof for the Base Case

above. The third case is that $P_{all} \neq \emptyset$ and there exists a plan-body $P \in P_{all}$ such that an event-goal is mentioned in P. This final case is discussed next.

Let E_{all} denote the (non-empty) set of event-goal types of all event-goals mentioned in all plan-bodies $P \in P_{all}$. From Definition 10 (Ranking), for all event-goals $e' \in E_{all}$, $\mathcal{R}_{\Pi}(e') < \mathcal{R}_{\Pi}(e) \leq k$. Then, from the induction hypothesis, for each $e' \in E_{all}$, there is exactly one tuple $\langle e', \phi_{e'}, L_{e'}^{mt}, L_{e'}^{mn} \rangle \in \Delta_{out}$ such that the tuple is the summary information of e', and $L_{e'}^{mn}$ captures e'. In particular, it is not difficult to see from procedure Summarise that all such tuples exist in Δ_{out} because the value returned by procedure Summarise-Event(e', Π, Δ) is added to set Δ in line 6, for each event-goal $e' \in E_{all}$. Moreover, since all event-goals in E_{all} have lower ranks than e, it is easy to see from procedure Summarise(Π, Λ) that procedure Summarise-Event(e, Π, Δ) is called only *after* procedure Summarise-Plan-Body(P, Π, Λ, Δ) is called for each plan-body $P \in P_{all}$, and in turn, that the latter procedure calls only take place after procedure Summarise-Event(e', Π, Δ) is called for each event-goal $e' \in E_{all}$.

Then, from Lemma 4, the induction hypothesis, and from Lemma 5, it follows that on the completion of the call to Summarise-Plan-Body(P, Π, Λ, Δ) in line 5 for each $P \in P_{all}$, there is exactly one tuple $\langle P, \epsilon, L_P^{mt}, L_P^{mn} \rangle \in \Delta$ such that the tuple is the summary information of P, and such that L_P^{mn} captures P. Finally, from Lemmas 6 and 7, we can conclude that after calling procedure Summarise-Event(e, Π, Δ) in line 6, there is exactly one tuple $\langle e, \phi_e, L_e^{mt}, L_e^{mn} \rangle \in \Delta$ such that the tuple is the summary information of e and L_e^{mn} captures e. Therefore, the theorem holds. □

Completeness

So far, we have shown that Algorithm 4.1 (Summarise) is sound, that is, whenever it determines that a literal is a must literal of some program, this is guaranteed to be the case. However, the algorithm is not complete, that is, there may exist a must literal of some program that the algorithm determines to be (only) a may literal of the program. Next, we will give some insight into the situations in which the algorithm is not complete, which, as discussed before, arise because the algorithm does not reason about context conditions of plan-rules.

It is important to note that, although the summary algorithm in the work of (Clement et al., 2007) is both sound and complete, they only deal with propositions, whereas we deal with first order atoms, and moreover, as discussed earlier, they make use of an assumption which requires plan-bodies to be consistent, whereas we do not have this assumption. In particular, because of

their assumption, one of the situations (shown below) in which our algorithm loses completeness is not handled by their algorithm.

Next, we give three scenarios in which our algorithm classifies a literal as only a may literal, while according to Definition 12 (Must Literal), the literal is also a must literal. First, given some plan-library Π, suppose there is an event-goal mentioned in Π that is associated with a single plan-rule having a context condition which entails $\neg{=}(b, Block1)$, and the plan-body shown below (from Section 4.2.3):

$$\begin{array}{l} \ldots; \\ +Colour(Block1, Blue); \\ ?(Block(b) \wedge Colour(b, Blue)); \\ -Colour(b, Blue); \\ +Colour(b, Red). \end{array}$$

Then, observe that, according to Definition 12, literal $Colour(Block1, Blue)$ is a must literal of the event-goal, since the literal will be true at the end of every successful execution of the event-goal, due to the context condition disallowing variable b from binding to $Block1$. However, since the algorithm does not reason about context conditions, according to the algorithm $Colour(Block1, Blue)$ is only a may literal of the event-goal, as the literal is may undone by belief operation $-Colour(b, Blue)$. Note that, although it is obvious from the given context condition that variable b will not bind to $Block1$, in practice, such a constraint can be enforced in various obscure ways. For example, there could be an event-goal program $!e(b)$ occurring immediately before step $+Colour(Block1, Blue)$ in the plan-body that is associated with a single plan-rule having test condition $?({=}(b, Block2))$.

Second, suppose that there is an event-goal mentioned in Π with two associated plan-rules, and that one of the plan-rules has context condition $Colour(Block1, c)$ along with the following plan-body:

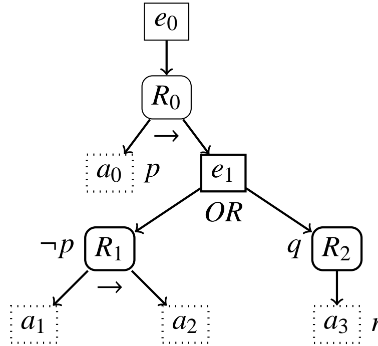
$$\begin{array}{l} \ldots; \\ -Colour(Block1, c); \\ +Colour(Block1, Blue). \end{array}$$

Furthermore, suppose that the second plan-rule has context condition $Colour(b, c) \wedge {=}(b, Block1)$,

along with the following plan-body:

$$\begin{aligned}&\ldots;\\&-Colour(b,c);\\&+Colour(b,Blue).\end{aligned}$$

Then, observe that, according to Definition 12, literal $Colour(Block1, Blue)$ is a must literal of the event-goal, because the literal will be true in the final state resulting from any successful execution of the event-goal (i.e., irrespective of the plan-rule chosen to achieve it). However, according to the algorithm, literal $Colour(Block1, Blue)$ is only a may literal of the event-goal, because the literal is not a must literal of all plan-bodies associated with the event-goal.



Finally, consider the plan-rule R_0 in the above figure. Suppose the following: the postcondition of action a_0 is p; the postcondition of action a_3 is r; proposition r is not mentioned anywhere else; the context condition of plan-rule R_1 is $\neg p$; the context condition of R_2 is q; and that all remaining context conditions and preconditions are *true*. Then, note that since literal $\neg p$ in the context condition of R_1 is clobbered by literal p brought about by a_0, plan-rule R_1 is never applicable. Consequently, according to Definition 12, literal r is a must literal of R_0 and e_0. However, since the algorithm will not realise that R_1 is never applicable, r is classified as only a may literal of R_0 and e_0.

4.3 Finding Hybrid-Plans

So far, we have shown how to compute summary information of event-goals mentioned in the plan-library. In this section, we show how the domain information for classical planning is constructed, in particular, how abstract actions are constructed using event-goals and their summary information. Moreover, we show how hybrid-plans are obtained using this domain information.

In what follows, we assume, without loss of generality, that no two event-goal types mentioned in a given plan-library Π have the same predicate symbol, and that no event-goal type mentioned in Π has the same symbol as an action mentioned in Λ.[6] In order to create the domain information for classical planning given a plan-library and an action-library, we first obtain using Algorithm 4.1 (Summarise) the set Δ of summary information of event-goal types mentioned in the plan-library. We then create an operator for each event-goal type in Δ as follows. First, we obtain the name of the operator by adding to the name of the event-goal variables occurring in its precondition. This is necessary because an operator name *act* needs to contain all free variables occurring in its precondition ψ (Ghallab et al., 2004, p. 28). Next, we take as the precondition of the operator the precondition associated with the event-goal. Finally, we take as the postcondition of the operator the set of must literals of the event-goal.[7]

Formally, given the set $\Delta = \mathsf{Summarise}(\Pi, \Lambda)$ for some plan-library Π and action-library Λ, we extract the set of abstract operators as follows:

$$\begin{aligned}\mathbb{A}(\Delta) \;=\; & \{e(\vec{x}, \vec{y}) : \psi \leftarrow \Phi^+; \Phi^- \mid \langle e(\vec{x}), \psi, L^{mt}, L^{mn}\rangle \in \Delta, \Phi^- = \{l \mid \neg l \in L^{mt}\},\\ & \Phi^+ = \{l \mid l \in L^{mt}, l \text{ is positive}\}, \vec{y} \text{ are the variables occurring in } \psi\\ & \text{but not in } \vec{x}\}.\end{aligned}$$

The domain information used as input for our classical planner is the set $\Lambda \cup \mathbb{A}(\Delta)$, that is, the set of newly created abstract operators together with the agent's existing action-library. We include the existing action-library in the domain information in order to not unnecessarily miss existing solutions.

At runtime, wherever it is desirable to apply classical planning to achieve some goal state $\mathcal{G}$ (e.g., at a programmer specified point in a plan-body), the domain information, the belief base $\mathcal{I}$ of the agent, and the goal state can be used with any classical planner to obtain a solution.[8] The only requirement on the classical planner is, of course, that it should be able to handle the expressivity of our operators. Specifically, the planner should be able to handle negative goals; preconditions and postconditions containing restricted first order atoms, in particular, atoms with variables and

[6] Recall that two event-goals e and e' have the same type if they have the same predicate symbol and arity. Moreover, as usual, given any event-goal $e(\vec{t})$, we use $e(\vec{x})$ to denote its type, where $|\vec{x}| = |\vec{t}|$ and $\vec{x}$ is a vector of distinct variables.

[7] Recall from Definition 12 that variables occurring in the must literals of an event-goal will also occur in the event-goal.

[8] We use Metric-FF(Hoffmann, 2003).

constants but no function symbols; and preconditions with disjunction, negation, and equality. From the classical planner's point of view, the domain information and solutions are composed entirely of primitive actions, despite the fact that some of them may represent event-goals.

Once a solution is found, abstract actions present in the solution (if any) need to be mapped back into their corresponding ground event-goals. More precisely, given a primitive solution $\sigma \in sol(\mathcal{I}, \mathcal{G}, \Lambda \cup \mathbb{A}(\Delta))$, we obtain the hybrid-plan (i.e., a partially-ordered set of primitive actions and event-goals) corresponding to σ with respect to Π, denoted by $\widehat{\sigma}_\Pi$, as follows:

$$\begin{aligned}
\widehat{\sigma}_\Pi &= [s, \phi], \text{ where} \\
s &= \{(i : act_i) \mid act_i \in \sigma, \text{ there is no event-goal mentioned in } \Pi \text{ having as its symbol} \\
&\quad \text{the symbol of } act_i\} \cup \{(i : e(t_1, \ldots, t_m)) \mid e(t_1, \ldots, t_n)_i \in \sigma, e(\vec{t_m}) \text{ is an event-goal} \\
&\quad \text{mentioned in } \Pi, m \leq n\}; \\
\phi &= \bigwedge\{i \prec j \mid i, j \in \{1, \ldots, |\sigma|\}, i < j\}.
\end{aligned}$$

For example, suppose $\sigma = act_1 \cdot act_2 \cdot act_3$, where act_1 and act_3 are primitive actions, and the symbol of act_2 matches the symbol of event-goal $e(x, y)$ mentioned in Π. Moreover, suppose $act_2 = e(P, Q, R)$. Then, $\widehat{\sigma}_\Pi = [s, \phi]$, where

$$\begin{aligned}
s &= \{(1 : act_1), (3 : act_3)\} \cup \{(2 : e(P, Q)\}, \text{ and} \\
\phi &= 1 \prec 2 \wedge 1 \prec 3 \wedge 2 \prec 3.
\end{aligned}$$

In particular, the third argument of abstract action $e(P, Q, R) \in \sigma$ is removed. The reason an extra third argument is included in the name of the abstract operator $e(x, y, z)$ is because, although variables (e.g., z) occurring in plan-rules associated with event-goal $e(x, y)$ do not have to also occur in the event-goal, by the definition of an operator (Ghallab et al., 2004, p. 28), any variable occurring in the precondition or postcondition of operator $e(x, y, z)$ must also occur in the operator name.

Unfortunately, hybrid-plans obtained in this manner are not necessarily correct, that is, there might not exist a viable decomposition of the hybrid-plan with respect to the planning problem. This is because of potential conflicts between effects brought about by event-goals and the preconditions of other event-goals in the hybrid-plan. In the next section, we will show why such

conflicts occur, and provide techniques for detecting such conflicts.

4.4 Validating Hybrid-Plans

Since abstract operators do not encode the may literals of their corresponding event-goals, this entails that, although the classical planner will find correct plans for the given planning problem with respect to the given encoding of the abstract and primitive operators, the corresponding hybrid-plan will not necessarily have a decomposition that solves the planning problem. This is because, when a hybrid-plan is decomposed, it is possible that a may literal brought about by the decomposition of an event-goal conflicts with a precondition encountered during the decomposition of some other event-goal, as shown in the following example.

Suppose hybrid-plan $[\{(1 : e_1), (2 : e_2)\}, 1 \prec 2]$ is obtained for initial state $\{p, r\}$ and goal state $\{s\}$ via classical planning as described in the previous section, where e_1 and e_2 have following summary information:

- the precondition of e_1 and e_2 is respectively p and $q \wedge r$;
- the set of must literals of e_1 and e_2 is respectively is $\{q\}$ and $\{s\}$; and
- the set of may literals of e_1 and e_2 is respectively $\{\neg r\}$ and $\emptyset$.

Now, observe that if the decomposition of e_1 brings about may literal $\neg r$, then the state resulting from the execution of e_1 is $\{p, q, \neg r\}$. Consequently, it is not possible to decompose e_2, because its precondition (i.e., all of its decompositions) require r to hold.

Because of this potential complication due to may literals, it is necessary to validate the hybrid-plan that is obtained, to ensure that it is viable. To this end, we perform two checks. We first perform a simple polynomial-time check to ascertain whether the hybrid-plan is potentially incorrect. If this is the case, we perform a second check using HTN planning to determine whether the hybrid-plan is actually incorrect.

The first check involves determining whether there is any literal mentioned in the precondition of an event-goal in the hybrid-plan such that this literal is possibly clobbered by a may literal of some other event-goal in the hybrid-plan. This process is shown next. For convenience, given a set of summary information Δ, with $\langle P, \phi, L^{mt}, L^{mn} \rangle \in \Delta$, we use function $pre[P, \Delta] = \phi$, function $must[P, \Delta] = L^{mt}$, and function $men[P, \Delta] = L^{mn}$. Moreover, given a totally-ordered hybrid-plan

$h = [s, \phi]$ and a goal state $\mathcal{G}$, we use $h^{\mathcal{G}}$ to denote a modified version of h that incorporates the goal state, that is, $h^{\mathcal{G}} = [s \cup \{(n_{\mathcal{G}} : act_{\mathcal{G}})\}, \phi \wedge (n_{last} \prec n_{\mathcal{G}})]$, where: *(i)* $act_{\mathcal{G}}$ is an action whose corresponding operator's precondition is the goal state, and whose corresponding operator's postcondition is the empty set; *(ii)* $n_{\mathcal{G}}$ is a task label not occurring in h; and *(iii)* n_{last} is the task label of the labelled task ordered to occur after all other labelled tasks in h. This modification to h ensures that the check for correctness takes into account the fact that, as highlighted in Definition 8, hybrid-plan h must bring about the given goal state.

Then, let $\mathcal{H}$ be a hybrid planning problem; let $\Delta_{in} = \textsf{Summarise}(\Pi, \Lambda)$; let $\sigma \in sol(\mathcal{I}, \mathcal{G}, \Lambda \cup \mathbb{A}(\Delta_{in}))$; let hybrid-plan $h = [s, \phi] = \widehat{\sigma}_{\Pi}$; and let $\Delta =$

$$\begin{aligned}
&\{\langle e, \phi, L^{mt}, L^{mn}\rangle\theta \;\mid\; \langle e, \phi, L^{mt}, L^{mn}\rangle \in \Delta_{in}, (n : e') \in s, e' = e\theta\} \;\cup \\
&\{\langle act, \psi, L, L\rangle\theta \;\mid\; (n : act') \in s, act : \psi \leftarrow \Phi^{+}; \Phi^{-} \in \Lambda, act' = act\theta, \\
&\qquad L = \Phi^{+} \cup \{\neg b \mid b \in \Phi^{-}\}\}.
\end{aligned}$$

We say that hybrid-plan h is <u>*correct*</u> with respect to $\mathcal{H}$ if for each $(n_1 : e_1), (n_2 : e_2) \in s^{\mathcal{G}}$, with $n_1 \neq n_2$ and $h^{\mathcal{G}} = [s^{\mathcal{G}}, \phi^{\mathcal{G}}]$, literal l_2 mentioned in $pre[e_2, \Delta]$, and literal $l_1 \in men[e_1, \Delta]$ such that *(i)* $l_2\theta_2 = \overline{l_1}\theta_1$, for some θ_1, θ_2, where $l_2\theta_2$ is ground; *(ii)* $l_1\theta_1, \overline{l_1}\theta_1 \notin must[e_1, \Delta]$; and *(iii)* $\phi^{\mathcal{G}} \models (n_1 \prec n_2)$: there exists $(n' : e') \in s^{\mathcal{G}}$ such that $\phi^{\mathcal{G}} \models (n_1 \prec n') \wedge (n' \prec n_2)$, and $\overline{l_1}\theta_1 \in must[e', \Delta]$ or $l_1\theta_1 \in must[e', \Delta]$. Otherwise, we say that h is <u>*potentially incorrect*</u> with respect to $\mathcal{H}$.

In words, for a hybrid-plan h to be considered correct, there should not be a literal l_2 in the precondition of some event-goal e_2 in h such that a may literal of some earlier event-goal e_1 in h can potentially become the negation of l_2, unless l_2 or its negation is also a must literal of an event-goal e' that occurs between e_1 and e_2.[9] Note that, although this condition is *sufficient* to determine whether a hybrid-plan is correct, the condition is not *necessary* to determine whether a hybrid-plan is correct. Therefore, given a correct hybrid-plan, the algorithm will not necessarily infer that it is correct; however, whenever the algorithm does infer that the hybrid-plan is correct, this is guaranteed to be the case.

Theorem 8. *If a hybrid-plan h is correct with respect to a hybrid planning problem $\mathcal{H}$, then h is*

[9]Note that because the negation of l_2 is a must literal of e', this *guaranteed* clobbering of l_2 by e' does not make h invalid — the planner has already accounted for this clobbering. Hence, any *potential* clobbering of l_2 by event-goals occurring *before* e' cannot make h invalid either.

a hybrid-solution for $\mathcal{H}$.

Proof. Let us assume the contrary, i.e., that h is correct, but that $sol(h^{\mathcal{G}}, \mathcal{I}, \mathcal{D}) = \emptyset$, where $h^{\mathcal{G}} = [s^{\mathcal{G}}, \phi^{\mathcal{G}}]$ is h modified to take the goal state into account. Suppose that $\vec{e} = e_1 \cdot \ldots \cdot e_m$ is the sequence of (ground) tasks corresponding to the (totally-ordered) hybrid-plan $h^{\mathcal{G}}$ — i.e., there is a permutation $(n_1 : e_1) \cdot \ldots \cdot (n_m : e_m)$ of $s^{\mathcal{G}}$ such that for each n_i, n_j, with $i < j$ and $i, j \in \{1, \ldots, m\}$, it is the case that $\phi^{\mathcal{G}} \models (n_i \prec n_j)$. Informally, since $sol(h^{\mathcal{G}}, \mathcal{I}, \mathcal{D}) = \emptyset$, there must be at least one task $e_i \in \vec{e}$ that cannot be successfully decomposed. Formally, the following condition must hold: there is a task $e_i \in \vec{e}$ such that for all primitive plan solutions $\sigma \in sol([\{(1 : e_1), \ldots, (i-1 : e_{i-1})\}, \bigwedge\{(j \prec k) \mid j, k \in \{1, \ldots, i-1\}, j < k\}], \mathcal{I}, \mathcal{D})$, it is the case that $sol([\{(1 : e_i)\}, true], \mathcal{I}', \mathcal{D}) = \emptyset$, where $\mathcal{I}' = Res^*(\sigma, \mathcal{I}, Op)$ is the result of applying σ in $\mathcal{I}$. Consequently, there does not exist a successful HTN execution $C_1 \cdot \ldots \cdot C_k$ of e_i with $C_1|_{\mathcal{B}} = \mathcal{I}'$ (Theorem 2, p. 76), and we can infer from the definition of a Precondition (Definition 11, p. 96) that $\mathcal{I}' \not\models pre[e_i, \Delta]$. Then, informally, it is not difficult to see that there must exist a literal l mentioned in $pre[e_i, \Delta]$ and a task e_x, with $1 \leq x < i$, such that some "hidden" mentioned literal of e_x conflicts with l, that is, *(i)* there is a successful HTN execution $C_1 \cdot \ldots \cdot C_k$ of e_x, with $C_k|_{\mathcal{B}} \models \bar{l}\theta$, $C_1|_{\mathcal{B}} \not\models \bar{l}\theta$, and $l\theta, \bar{l}\theta \notin must[e_x, \Delta]$; and *(ii)* there is no task e_y, with $x < y < i$, such that $l\theta \in must[e_y, \Delta]$ or $\bar{l}\theta \in must[e_y, \Delta]$. From condition *(i)*, it follows that literal $\bar{l}$ is in the set of literals captured by task e_x (Definition 14), and from Theorem 7, it follows that $\bar{l} \in men[e_x, \Delta]$ (up to variable substitutions). Finally, combined with condition *(ii)*, it follows that h is not a correct hybrid-plan — a contradiction. □

If a hybrid-plan is found to be correct, then it can either be executed, or as we will show in the next chapter, improved. However, if a hybrid-plan is found to be potentially incorrect, then we determine whether it is (actually) incorrect. To this end, given a hybrid planning problem $\mathcal{H}$, we use HTN planning to determine whether h is a hybrid-solution for $\mathcal{H}$.

It is worth noting that it may be possible to *repair* an incorrect (totally-ordered) hybrid-plan in order to make it correct, by adding actions to it, removing actions from it and/or removing constraints from its constraint formula. Let us illustrate this with an example.

Suppose we have the total-order hybrid-plan $h = [\{(1 : e_1), (2 : e_2), (3 : e_3)\}, 1 \prec 2 \wedge 2 \prec 3]$, where the preconditions of e_1, e_2 and e_3 are respectively p, q and r; the set of must literals of e_1, e_2 and e_3 are respectively $\{r\}$, $\{w\}$ and $\{s\}$; the set of may literals of e_1, e_2 and e_3 are respectively $\emptyset$, $\{\neg r\}$ and $\emptyset$; the initial state is $\{p, q\}$; and

the goal state is $\{w, s\}$. Moreover, suppose that may literal $\neg r$ of e_2 is unavoidable, that is, literal $\neg r$ is true at the end of every successful decomposition of e_2.

Then, observe that hybrid-plan h is incorrect, as may literal $\neg r$ of e_2 clobbers the precondition r of e_3. However, we can repair h by removing ordering constraint $2 \prec 3$ from its constraint formula, which then allows e_3 to precede e_2; in this way, may literal $\neg r$ of e_2 is brought about only after e_3 is executed, and the clobbering of e_3's precondition can be avoided.

Alternatively, if there is an action *act* in the action-library that brings about literal r and whose precondition is applicable in state {p,q}, we could repair the original hybrid-plan h by placing *act* between e_2 and e_3 to obtain hybrid-plan $[\{(1 : e_1), (2 : e_2), (4 : act), (3 : e_3)\}, 1 \prec 2 \wedge 2 \prec 3 \wedge 2 \prec 4 \wedge 4 \prec 3]$. In this way, the clobbering of e_3's precondition can be avoided.

It is not difficult to see that, in some situations, we can only repair hybrid-plans by adding and/or removing actions – i.e., removing constraints will not work. However, in general, repairing a (sequential or partially-ordered) plan in this manner is as hard as generating a new plan from scratch (Nebel and Koehler, 1995). Therefore, if a hybrid-plan is found to be incorrect, we obtain a new hybrid-plan via classical planning, using the techniques discussed in Section 4.3.

Chapter 5

Obtaining a Preferred First Principles Plan[†]

In the previous chapter, we provided the means for obtaining correct hybrid-plans, i.e., hybrid-solutions, for a given planning problem. In particular, we showed how to summarise the plan-library, how to create operators using summary information of event-goals, and how to obtain hybrid-plans that are correct, via first principles planning. In this chapter, we investigate how to obtain *preferred* hybrid-solutions. We will present different notions of preferred hybrid-solutions, properties of such hybrid-solutions, and data structures and algorithms for realising one of these notions.

In first principles planning, a plan is said to be a solution for a planning problem if the plan is *correct* relative to the planning problem, i.e., if executing the plan from the initial state will result in the goal state being met. Correctness is an important property that all plans must meet. In addition to correctness, many domains require that plans adhere to certain other properties. This is because correct plans can still have shortcomings, such as *redundancy* and *non-minimality*. A solution for a planning problem is said to be *redundant* if one or more actions can be removed from the solution to obtain a plan that is still a solution for the problem. A solution of length n for a planning problem is said to be *non-minimal* if a solution of length less than n exists for the problem.

Similarly, correct hybrid-plans, i.e., hybrid-solutions, can also have shortcomings. A signifi-

[†]Part of the work presented in this chapter has been previously published in (de Silva et al., 2009).

cant shortcoming is that a hybrid-solution can be redundant, i.e., every primitive solution produced by the hybrid-solution, for the given planning problem, can be redundant. This means that the agent cannot avoid executing one or more redundant actions during its execution of the hybrid-solution. Intuitively, redundancy occurs as a result of tasks existing in the hybrid-solution that are unnecessary and/or overly abstract. Since tasks can be thought of as a collection of other tasks, a higher level of abstraction implies a larger collection of tasks. Consequently, having overly abstract tasks in a hybrid-solution results in the hybrid-solution producing, in addition to the necessary actions, also unnecessary (redundant) actions.

Let us illustrate our overall framework with an example. Consider the Mars Rover agent of Figure 4.4 (p. 106), excluding the optional plan-rules R_2 and R_7. Suppose that at some point, the agent invokes a planner, which returns the hybrid-solution h shown in Figure 5.1(a). Consider next the actual execution of hybrid-solution h shown in Figure 5.1(c). Now, notice that breaking the connection after sending the results for *Rock*2, and then re-establishing it before sending the results for *Rock*3 are unwarranted, or redundant steps. Such redundancy is brought about by the overly abstract task *PerformSoilExperiment*. What we would prefer to have is the *non-redundant* hybrid-solution h' shown in Figure 5.1(b). This solution avoids the redundancy inherent in the initial solution, yet still retains much of the structure of the abstract plans provided by the programmer. In particular, we retain the abstract tasks *Navigate* and *ObtainSoilResults*, which would allow us to achieve these tasks in an alternative manner to that shown here, if such existed and was warranted by the situation during execution. The replacement of each of *PerformSoilExperiment* and *TransmitSoilResults* with a subset of their components is clearly motivated in order to remove redundancy.

It is important to note that, while our framework does retain as much as possible the structure of the abstract plans provided by the user, it may be the case that the user does not want certain important tasks to be removed at all from plans, even if those tasks are redundant. For example, as illustrated in (Kambhampati et al., 1998), although the task of buying a ticket when travelling by bus may not be necessary for achieving the goal of getting to the destination, one may still want to always perform this task when travelling by bus. Here, for simplicity, we have used “redundancy” as the sole criteria for classifying a task as unnecessary for the required goal. However, a more

1. *Navigate*(*Rock*1, *Rock*2)
2. *PerformSoilExperiment*(*Rock*2)
3. *Navigate*(*Rock*2, *Rock*3)
4. *PerformSoilExperiment*(*Rock*3)

(a) Hybrid-solution h

1. *Navigate*(*Rock*1, *Rock*2)
2. *ObtainSoilResults*(*Rock*2)
3. *EstablishConnection*
4. *SendResults*(*Rock*2)
5. *Navigate*(*Rock*2, *Rock*3)
6. *ObtainSoilResults*(*Rock*3)
7. *SendResults*(*Rock*3)
8. *BreakConnection*

(b) Hybrid-solution h'

1. *Navigate*(*Rock*1, *Rock*2)
   (A) *CalibrateViaGPS*
   (B) *Move*(*Rock*1, *Rock*2)
2. *PerformSoilExperiment*(*Rock*2)
   (A) *ObtainSoilResults*(*Rock*2)
      (i) *PickSoilSample*(*Rock*2)
      (ii) *AnalyseSoilSample*(*Rock*2)
         (a) *GetMoistureContent*(*Rock*2)
         (b) *GetSoilParticleSize*(*Rock*2)
      (iii) *DropSoilSample*
   (B) *TransmitSoilResults*(*Rock*2)
      (i) *EstablishConnection*
      (ii) *SendResults*(*Rock*2)
      (iii) ***BreakConnection***
3. *Navigate*(*Rock*2, *Rock*3)
   (A) *CalibrateViaGPS*
   (B) *Move*(*Rock*2, *Rock*3)
4. *PerformSoilExperiment*(*Rock*3)
   (A) *ObtainSoilResults*(*Rock*3)
      (i) *PickSoilSample*(*Rock*3)
      (ii) *AnalyseSoilSample*(*Rock*3)
         (a) *GetMoistureContent*(*Rock*3)
         (b) *GetSoilParticleSize*(*Rock*3)
      (iii) *DropSoilSample*
   (B) *TransmitSoilResults*(*Rock*3)
      (i) ***EstablishConnection***
      (ii) *SendResults*(*Rock*3)
      (iii) *BreakConnection*

(c) Execution trace of hybrid-solution h

Figure 5.1: (a) A redundant hybrid-solution h; (b) a hybrid-solution h' with redundancy (actions in bold) removed; and (c) the execution trace of h.

flexible approach could be used for classifying a task as unnecessary for the required goal. We give insights into such an approach in Chapter 7.

As we can see from the above example, non-redundant hybrid-solutions favour specific tasks over abstract tasks. On the other hand, the *user intent* notion discussed in the previous chapter (Section 4.1, p. 87) favours abstract tasks over specific tasks. Recall that, intuitively, a hybrid-solution conforms to user intent if it can be *parsed* in terms of the method-library; therefore, any hybrid-solution composed entirely of arbitrary abstract tasks will conform to user intent, whereas only certain hybrid-solutions containing primitive tasks (actions) will conform to user intent. In this chapter, we make the preference for abstract tasks even stronger by requiring that hybrid-plans

are as abstract as possible, or *maximally-abstract*. Intuitively, a maximally-abstract hybrid-plan is one which does not contain a collection of abstract tasks which could potentially be combined into a single (more) abstract task, thus improving the abstraction level of the hybrid-plan. Therefore, maximal-abstractness ensures that hybrid-plans only contain the most abstract tasks possible. The advantage of such hybrid-plans is that, in addition to intuitively capturing the user intent property, they support flexibility and robustness in execution, i.e., they are better able to deal with failure during execution, by trying alternative reductions on the failure of less abstract tasks.

As one can observe, while non-redundancy favours specific tasks, the need for flexibility and robustness favours abstract tasks. Consequently, the main aim of this chapter is to investigate what the desired level of abstraction is for hybrid-plans. In particular, we focus on finding non-redundant hybrid-solutions that are maximally-abstract, but also *minimal*, where a minimal hybrid-solution is one that is a non-redundant hybrid-solution from which no tasks can be removed to obtain another non-redundant hybrid-solution. To this end, we define three compound notions of hybrid-plans, based on the notions of minimality, non-redundancy and maximal-abstraction. The strongest notion is called a *minimal non-redundant maximal-abstraction (MNRMA) hybrid-plan*, the second strongest notion is called a *MNRMA specialisation of a hybrid-plan*, and the weakest notion is called a *preferred specialisation of a hybrid-plan*.

A MNRMA hybrid-plan is one that is at the ideal level of abstraction. This notion is defined relative to all other conceivable hybrid-plans for the given hybrid planning problem. Consequently, finding a MNRMA hybrid-plan is very computationally expensive. The intermediate notion – MNRMA specialisations of a hybrid-plan – defines the desired level of abstraction for a *given* hybrid-plan relative to the space of *decompositions* of the hybrid-plan. Although still computationally expensive, this notion is conceptually closer to the final notion – MNRMA specialisation of a hybrid-plan, which defines the desired level of abstraction for a given hybrid-plan relative to a *single* decomposition of the hybrid-plan. A preferred specialisation for a given hybrid-plan can be computed in polynomial time.

This chapter is organised as follows. First, in Section 5.1, we give some definitions, conventions and preliminary notions. Second, in Section 5.2, we investigate the three desired properties of hybrid-plans, that is, non-redundancy, minimality and maximal-abstractness, and we then formulate our ideal (MNRMA) notion of a hybrid-plan. Third, in Section 5.3, we discuss the intermediate notion: MNRMA specialisations of a hybrid-plan, and in Section 5.4, we discuss the

weakest notion: preferred specialisation of a hybrid-plan. Finally, in Section 5.5, we provide data structures and algorithms for obtaining preferred specialisations of a given hybrid-plan.

5.1 Preliminary Definitions

In this chapter, we make use of the notion of a *labelled* primitive plan, i.e., a primitive plan constructed from labelled tasks rather than un-labelled tasks. More precisely, a ***labelled*** primitive plan $\tau = (n_1 : t_1) \cdot \ldots \cdot (n_m : t_m)$ is a sequence of labelled tasks. We will use τ to denote labelled primitive plans, and σ to denote (un-labelled) primitive plans. We will sometimes blur the distinction between primitive plans σ and labelled primitive plans τ – in particular, we will use labelled primitive plans in place of (un-labelled) primitive plans with the obvious meaning. The other conventions we use in this chapter are the following: *(i)* h for hybrid-plans or hybrid-solutions, *(ii)* d for task networks, and *(iii)* λ for decomposition traces (introduced in Section 5.4).

In HTN planning, it is sometimes convenient to specify a method which, given a particular condition, amounts to "doing nothing." However, in HTN syntax, conditions cannot be specified inside methods that have no tasks. Moreover, the semantics of HTN does not allow conditions to be specified on compound tasks that are eventually reduced into the empty set. For specifying conditions in such situations, *dummy* primitive tasks, which we call ϵ tasks in this chapter, are used. Although ϵ tasks are still primitive tasks, they have no precondition or effect; therefore, executing them amounts to "doing nothing."

> To illustrate why ϵ tasks are necessary in HTN planning, consider an elevator domain consisting of the following two methods for handling the compound task *go-to-bottom*, which keeps moving down one floor until the ground floor (floor 0) is reached:
>
> $(\textit{go-to-bottom}, [\{(1 : \textit{move-down}), (2 : \textit{go-to-bottom})\}, (1 \prec 2) \wedge (\neg Floor(0), 1)])$
> $(\textit{go-to-bottom}, [\{(1 : \epsilon)\}, (Floor(0), 1)]).$
>
> Observe that without the ϵ task in the second method, there is no way of specifying that the elevator should stop moving down once the ground floor is reached.[1]

We make two reasonable assumptions regarding ϵ tasks. First, since ϵ tasks are used solely for the purpose of specifying conditions in the two situations mentioned above, we assume in this chapter, without loss of generality, that all ϵ tasks mentioned in primitive plans in set $sol(d, \mathcal{I}, \mathcal{D})$ of HTN primitive plan solutions have been removed. Second, we assume that given any method, its task network is such that the set of labelled tasks is non-empty (even if its constraint formula is *true*), i.e., that the set of labelled tasks contains at least one labelled ϵ task.

Finally, we assume, without loss of generality, that labels within a HTN task network are unique, and that its constraint formula does not mention any task labels that do not occur in the task network's set of labelled tasks.

5.2 MNRMA Hybrid-Plans

In this section, we consider three inter-related concepts, and define these precisely in order to obtain an unambiguous description of an "ideal" hybrid-plan. These concepts we call *maximal-abstractness*, *minimality* and *non-redundancy*. Intuitively, given a hybrid planning problem $\mathcal{H}$, a *non-redundant* hybrid-solution for $\mathcal{H}$ is one which can produce (via one or more HTN reductions) a primitive plan that is a non-redundant solution for $\mathcal{H}$; a *minimal* hybrid-solution for $\mathcal{H}$ is a non-redundant hybrid-solution for $\mathcal{H}$ from which no (primitive or non-primitive) tasks can be removed to obtain a hybrid-solution that is still non-redundant for $\mathcal{H}$; and a maximally-abstract hybrid-plan is one which does not contain a collection of abstract tasks which could potentially be combined into a single (more) abstract task.

More precisely, a maximally-abstract hybrid-plan is one that is not a "refinement" of any other hybrid-plan. Intuitively, the refinements of a task network (or hybrid-plan) are all the "intermediate" or "partially reduced" task networks encountered, during the HTN search for primitive plan solutions of the given task network. The notions refinement, maximal-abstractness, non-redundancy and minimality are illustrated in the following example.

Consider the HTN domain in Figure 5.2. Observe that the refinements of hybrid-

[1]One may wonder whether the following encoding works:
$(go\text{-}to\text{-}bottom, [\{(1 : move\text{-}down), (2 : go\text{-}to\text{-}bottom)\}, (1 \prec 2) \land (\neg Floor(1), 1) \land (\neg Floor(0), 1)])$
$(go\text{-}to\text{-}bottom, [\{(1 : move\text{-}down)\}, (Floor(1), 1)])$.
This encoding will not work when the initial state is such that the elevator is at floor 0. In this case, no methods of *go-to-bottom* can be applied and the search fails.

solution t_0 in the figure is basically the following set:

$$\{t_0, \quad t_1 \cdot t_2, \quad t_1 \cdot t_3 \cdot t_4,$$
$$t_1 \cdot t_3 \cdot a_3, \quad t_1 \cdot a_2 \cdot t_4, \quad t_1 \cdot a_1 \cdot a_2 \cdot t_4,$$
$$t_1 \cdot a_2 \cdot a_3, \quad t_1 \cdot a_1 \cdot a_2 \cdot a_3, \quad a_1 \cdot t_2,$$
$$a_1 \cdot t_3 \cdot t_4, \quad a_1 \cdot t_3 \cdot a_3, \quad a_1 \cdot a_2 \cdot t_4,$$
$$a_1 \cdot a_1 \cdot a_2 \cdot t_4, \quad a_1 \cdot a_2 \cdot a_3, \quad a_1 \cdot a_1 \cdot a_2 \cdot a_3\}.$$

Refinement $t_1 \cdot a_1 \cdot a_2 \cdot t_4$ is obtained by reducing t_0 three times: first, t_0 is reduced using method m_0 to obtain task network $t_1 \cdot t_2$; second, $t_1 \cdot t_2$ is reduced using method m_2 to obtain task network $t_1 \cdot t_3 \cdot t_4$; and third, $t_1 \cdot t_3 \cdot t_4$ is reduced using method m_4 to obtain task network $t_1 \cdot a_1 \cdot a_2 \cdot t_4$. Similarly, refinement $a_1 \cdot a_2 \cdot t_4$ is obtained using methods m_0, m_1, m_2, and m_3; refinement $a_1 \cdot a_1 \cdot a_2 \cdot t_4$ is obtained using methods m_0, m_1, m_2, and m_4; refinement $a_1 \cdot a_2 \cdot a_3$ is obtained using methods m_0, m_1, m_2, m_3, and m_5; and refinement $a_1 \cdot a_1 \cdot a_2 \cdot a_3$ is obtained using methods m_0, m_1, m_2, m_4, and m_5. The rest of the refinements are obtained in a similar manner.

Next, consider the table below, which shows some of the different hybrid-solutions possible, for the hybrid planning problem consisting of initial state $\{p\}$, goal state $\{s\}$, and the HTN domain in Figure 5.2. Based on the above refinements, we can see that hybrid-solution t_2 is maximally-abstract because it is not a refinement of any other hybrid-solution (t_0 does not have a refinement that matches t_2 alone). On the other hand, hybrid-solution $t_1 \cdot t_2$ is not maximally-abstract because it is a refinement of t_0.

Hybrid-solution t_0 is non-redundant because it can produce the non-redundant primitive solution $a_1 \cdot a_2 \cdot a_3$, by selecting methods in the following sequence: m_0, m_1, m_2, m_3, and m_5. Hybrid-solution $t_5 \cdot t_2$ is redundant because all of its primitive solutions – $a_4 \cdot a_5 \cdot a_2 \cdot a_3$ and $a_4 \cdot a_5 \cdot a_1 \cdot a_2 \cdot a_3$ – are redundant; solution $a_4 \cdot a_5 \cdot a_2 \cdot a_3$ is redundant because action a_5 or a_2 can be removed from the solution and still have a solution, and solution $a_4 \cdot a_5 \cdot a_1 \cdot a_2 \cdot a_3$ is redundant because actions a_4 and a_5 (or other combinations of actions) can be removed from the solution and still have a solution.

Hybrid-solution $t_5 \cdot t_4$ is minimal because *(i)* it is a non-redundant hybrid-

solution, and *(ii)* none of its proper subsequences (t_5 and t_4) are non-redundant hybrid-solutions. Finally, although $t_1 \cdot t_2$ is non-redundant, it is not minimal, because a proper subsequence of it – t_2 – is a non-redundant hybrid-solution.

HYBRID-SOL.	NON-REDUNDANT	MINIMAL	MAXIMALLY-ABSTRACT	MNRMA
t_0	√	√	√	√
t_2	√	√	√	√
$t_5 \cdot t_4$	√	√	√	√
$t_3 \cdot t_4$	√	√	×	×
$t_5 \cdot t_2$	×	×	√	×
$t_1 \cdot t_2$	√	×	×	×
$t_1 \cdot t_3 \cdot t_4$	√	×	×	×
$t_5 \cdot t_3 \cdot t_4$	×	×	×	×

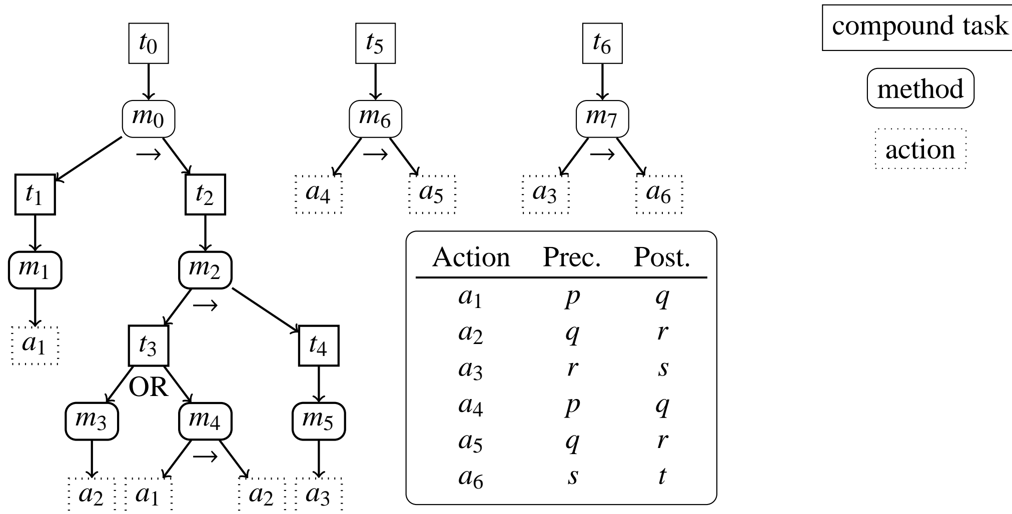


Figure 5.2: A simple totally-ordered HTN domain. An arrow below a method indicates that its steps are ordered from left to right. The table shows the preconditions and postconditions of the actions.

5.2.1 Non-Redundancy and Minimality

In what follows, we shall make precise the notions of non-redundancy and minimality for hybrid-solutions. To define non-redundancy, we extend from Fink et al. (Fink and Yang, 1992) the notion

perfect justification defined for primitive solutions.

Definition 15. (Perfect Justification (Fink and Yang, 1992)) A primitive solution σ for a classical planning problem $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$ is a *perfect justification* for C if there does not exist a proper subsequence σ' of σ such that σ' is a primitive solution for C. ■

It is easy to see from this definition that a perfect justification can be obtained from any given primitive solution. We say that a hybrid-solution is *non-redundant* if it can produce at least one perfect justification.

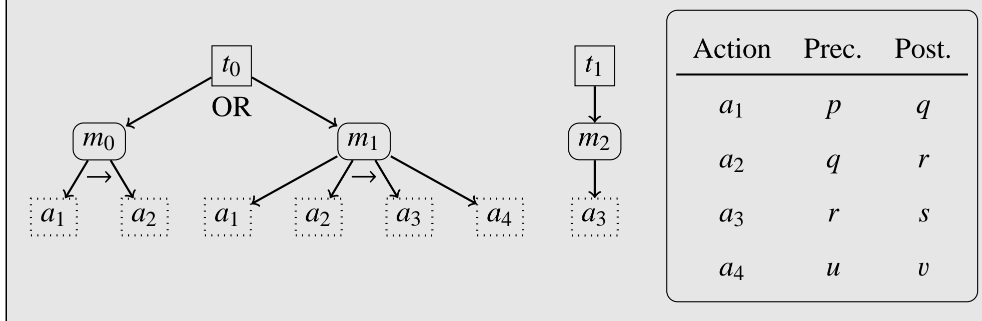
Definition 16. (Non-redundant Hybrid-Solutions) Let $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ be a hybrid planning problem. Then, h is a *weakly non-redundant* hybrid-solution for $\mathcal{H}$ if there exists $\sigma \in sol(h, \mathcal{I}, \mathcal{D}) \cap sol(\mathcal{I}, \mathcal{G}, Op)$ such that σ is a perfect justification for problem $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. Also, d is *strongly non-redundant* if every $\sigma \in sol(d, \mathcal{I}, \mathcal{D}) \cap sol(\mathcal{I}, \mathcal{G}, Op)$ is a perfect justification for problem $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. ■

In the rest of this chapter, when we refer to non-redundancy, we are referring to the weak notion. Next, we define the notion *minimality*. Intuitively, we say that a non-redundant hybrid-solution h is minimal, if there is no substructure of h which gives the same result. More precisely, a non-redundant hybrid-solution $h = [s, \phi]$ for a hybrid planning problem $\mathcal{H}$ is a *minimal non-redundant hybrid-solution* for $\mathcal{H}$ if there does not exist a non-redundant hybrid-solution $h' = [s', \phi']$ for $\mathcal{H}$ such that $s' \subset s$, where ϕ' is obtained from ϕ by replacing with *true* every (ordering) constraint that mentions some task label occurring in the set $s \setminus s'$.

Note that minimality is a stronger notion than non-redundancy. This is illustrated in the table of the previous example, with hybrid-solutions $t_1 \cdot t_2$ and $t_1 \cdot t_3 \cdot t_4$, which are non-redundant but not minimal. We do not define minimality and non-redundancy as independent concepts because this can lead to a situation in which there is a hybrid-solution that is non-redundant and non-minimal, but all minimal hybrid-solutions that can be obtained from it are redundant.

Let us illustrate with an example how such a situation can arise. But first, let us assume that minimality is defined relative to hybrid-solutions, rather than relative to non-redundant hybrid-solutions, i.e., a minimal hybrid-solution is a hybrid-solution from which no tasks can be removed to obtain a hybrid-plan that is still a hybrid-solution.

Now, let us consider hybrid-solution $t_0 \cdot t_1$ for the hybrid planning problem consisting of initial state $\{p, u\}$, goal state $\{s\}$, and the HTN domain below. Observe that this hybrid-solution is non-redundant and not minimal; it is non-redundant because it can produce the non-redundant primitive solution $a_1 \cdot a_2 \cdot a_3$, by selecting methods m_0 and m_2, and it is not minimal because its proper subsequence t_0 is also a hybrid-solution – t_0 can produce the primitive solution $a_1 \cdot a_2 \cdot a_3 \cdot a_4$, by selecting method m_1. However, although hybrid-solution t_0 is minimal, it is redundant, because its (only) primitive solution $a_1 \cdot a_2 \cdot a_3 \cdot a_4$ contains the redundant action a_4.



Action	Prec.	Post.
a_1	p	q
a_2	q	r
a_3	r	s
a_4	u	v

Consequently, although it may be possible to extract a hybrid-solution that is non-redundant (alone), and one that is minimal (alone) from a given hybrid-solution, it may not be possible to extract from the given hybrid-solution one that is both non-redundant and minimal. To avoid such situations, we define minimality as a strengthening of non-redundancy.

5.2.2 Maximal-Abstractness

Next, after building the necessary foundations, we will define the third desirable property of hybrid-plans: *maximal-abstractness*. As mentioned earlier, a hybrid-plan is maximally-abstract if it does not match a refinement of any other hybrid-plan, where the refinements of a given task network are all the “intermediate” task networks encountered, during the HTN search for primitive plan solutions of the task network. Although the HTN semantics of Erol et al. (Erol et al., 1996) provides construct $sol(d, \mathcal{I}, \mathcal{D})$ for representing the primitive plan solutions of a task network d (see Section 2.3.2), there is no construct in the semantics for representing such “intermediate” task networks. Therefore, in this section we extend the HTN semantics of (Erol et al., 1996) with such a construct.

Technically, the *refinements* of a task network d is the set of all task networks obtained by

reducing d zero or more times; a *refinement* is a member of this set. We define refinements of a task network d as the reflexive transitive closure of the set of HTN reductions $red(d, \mathcal{D})$ (see Appendix A.3). In the definition below, $refn(d, \mathcal{D})^n$ is the set of task networks obtained from n reductions of d, and $refn^{\omega}(d, \mathcal{D})$ is the set of task networks obtained from all finite reductions of d.

Definition 17. (Refinements) Let d and $\mathcal{D}$ be a task network and an HTN domain, respectively. The set of *refinements* of d relative to $\mathcal{D}$, denoted by $refn(d, \mathcal{D})$, is defined as $refn(d, \mathcal{D}) = refn^{\omega}(d, \mathcal{D})$, where

$$\begin{aligned} refn^0(d, \mathcal{D}) &= \{d\}; \\ refn^{n+1}(d, \mathcal{D}) &= \bigcup_{d' \in refn^n(d, \mathcal{D})} red(d', \mathcal{D}); \\ refn^{\omega}(d, \mathcal{D}) &= \bigcup_{n \in \mathbb{N}_0} refn^n(d, \mathcal{D}). \end{aligned}$$

■

Notice that since a refinement of a task network is *any* "intermediate" task network d encountered during the decomposition of the given task network, there is no guarantee that a refinement will produce a primitive plan solution, i.e., it is possible that $sol(d, \mathcal{I}, \mathcal{D}) = \emptyset$, for all states $\mathcal{I}$ and for the HTN domain $\mathcal{D}$ in question.

To determine whether a given hybrid-plan *matches* a given refinement, we need to determine whether: *(i)* tasks in the hybrid-plan are also present in the refinement; *(ii)* ordering constraints specified on tasks in the hybrid-plan are compatible with (i.e., do not conflict with) those specified on tasks in the refinement; and *(iii)* the refinement produces any of the primitive plan solutions of interest produced by the hybrid-plan.

The final check is necessary for the following reason. Even if a refinement and hybrid-plan contain the same labelled tasks and the same ordering constraints specified on them, the refinement may still contain state and variable binding constraints, whereas the hybrid-plan (by definition) will only contain ordering constraints. Consequently, the refinement may be more constrained than the hybrid-plan, and possibly unable to produce any of the primitive plans that the hybrid-plan can produce.

For example, let us consider Figure 5.3. The figure shows graphically the reduction

of hybrid-solution $h = [\{(2 : t_1), (3 : t_2)\}, (2 \prec 3)]$ on the right, and the reductions of hybrid-solution $h' = [\{(1 : t_0)\}, true]$ on the left. Observe that task network $d = [\{(2 : t_1), (3 : t_2)\}, (p, 2)]$ in the figure is a refinement of h' – the former is obtained from a single reduction of the latter. Observe, also, that the ordering constraint $(2 \prec 3)$ of hybrid-solution h is compatible with the ordering constraints of d.

Now, suppose h and h' are hybrid-solutions for initial state $\{\neg p\}$ and some goal state. Suppose, further, that we wish to determine whether h is maximally-abstract relative to the set $\{(4 : a_3) \cdot (5 : a_4) \cdot (6 : a_5) \cdot (7 : t_6)\}$ of primitive plan solutions for h. Although ordering constraints of hybrid-solution h are compatible with those of refinement d, and they both have the same tasks, the refinement cannot produce the primitive plan solution $(4 : a_3) \cdot (5 : a_4) \cdot (6 : a_5) \cdot (7 : a_6)$ because the state constraint of d – *(p,2)* – conflicts with initial state $\{\neg p\}$. Since h' does not have a refinement that matches h, we say that h is maximally-abstract.

On the other hand, suppose we have initial state $\{p\}$ instead of $\{\neg p\}$. In this case, h is not maximally-abstract, because refinement d of hybrid-solution h' matches h, i.e., d and h have the same tasks, their ordering constraints are compatible, and d can produce the primitive plan solution $(4 : a_3) \cdot (5 : a_4) \cdot (6 : a_5) \cdot (7 : a_6)$.

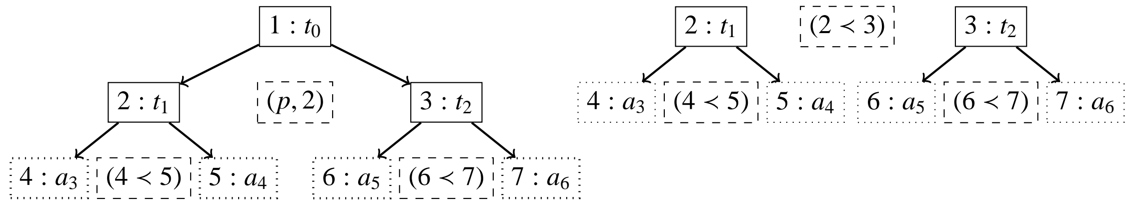


Figure 5.3: Refinements for hybrid-solution $[\{(1 : t_0)\}, true]$ (left) and hybrid-solution $[\{(2 : t_1), (3 : t_2)\}, (2 \prec 3)]$ (right) depicted graphically. Dashed rectangles represent constraints on adjacent labelled tasks.

We can now define the notion maximally-abstract. We say that a hybrid-plan h is maximally-abstract if there is no other hybrid-plan such that one of its refinements match h.

Definition 18. (Maximally-Abstract) Let $\mathcal{D}$ be an HTN domain and $\mathcal{I}$ be a state. Let Δ be a set of hybrid-plans and $h = [s_h, \phi_h] \in \Delta$ be a hybrid-plan in it. Let $\Sigma_h \subseteq sol(h, \mathcal{I}, \mathcal{D})$ be a set of primitive plan solutions.

Hybrid-plan h is (strongly) *maximally-abstract* among set Δ for Σ_h if there is no hybrid-plan $h' = [s_{h'}, \phi_{h'}] \in \Delta$ with $|s_{h'}| < |s_h|$ such that:

1. $d_1 \in refn(h', \mathcal{D})$,
2. $d_2 = [s_{d_2}, \phi_{d_2}]$ is a ground instance and task label renaming of d_1 such that $s_{d_2} \supseteq s_h$,
3. $d_3 = [s_{d_2}, \phi_{d_2} \wedge \phi_h]$, and
4. $\Sigma_h \cap sol(d_3, \mathcal{I}, \mathcal{D}) \neq \emptyset$;

moreover, if $\Sigma_h \subseteq sol(d_3, \mathcal{I}, \mathcal{D})$ also holds, h is *weakly maximally-abstract* among Δ for Σ_h. ■

In words, a hybrid-plan h is (strongly) maximally-abstract among hybrid-plans in Δ for a set of primitive plan solutions Σ_h, if there is no shorter hybrid-plan h' in Δ that can produce h by refinements, without losing *all* of the primitive plan solutions in Σ_h. Similarly, h is weakly maximally abstract among Δ for Σ_h if there is no shorter hybrid-plan h' in Δ that can produce h by refinements, without losing *any* of the primitive plan solutions in Σ_h. In the rest of the chapter we use the weaker notion of maximal-abstraction.

So far, we have defined the three desirable properties of hybrid-solutions: *non-redundancy*, *minimality*, and *maximal-abstractness*. Since ideally,[2] a hybrid-solution should satisfy all three properties, we will now define what "ideal" hybrid-solutions are, by combining the notions maximal-abstractness and minimal non-redundancy. More precisely, to conform to this "ideal" notion, a hybrid-solution must be minimal, and maximally-abstract relative to the set of perfect justifications of the hybrid-solution. We call such "ideal" hybrid-plans *minimal non-redundant maximally-abstract* (MNRMA) hybrid-plans.

Definition 19. (MNRMA Hybrid-Plans) A hybrid-plan h is a *minimal non-redundant maximal -abstraction* (MNRMA) for a hybrid planning problem $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ if and only if

1. h is a minimal non-redundant hybrid-solution for $\mathcal{H}$; and
2. h is a (strongly) maximally-abstract hybrid-plan among *all* possible hybrid-plans for the set $\Sigma \cap sol(h, \mathcal{I}, \mathcal{D})$, where Σ is the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$.

The set of all MNRMA plans for $\mathcal{H}$ is denoted MNRMA($\mathcal{H}$). ■

[2]Although there may be other desirable properties of hybrid-solutions, we are only interested in the three mentioned.

The definition states that a hybrid-plan is considered a MNRMA hybrid-plan if it is *(i)* a maximally-abstract hybrid-plan with respect to all possible hybrid-plans, and the set of all perfect justifications the hybrid-plan is able to produce, and *(ii)* a minimal (and hence non-redundant) hybrid-plan.

The following theorem states that, whenever a hybrid planning problem can be solved, there is, at least, one (ideal) MNRMA hybrid-plan.

Theorem 9. *Let $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ be a hybrid planning problem. If $sol(\mathcal{I}, \mathcal{G}, Op) \neq \emptyset$, then there exists an MNRMA for $\mathcal{H}$.*

Proof. Let Σ^{nr} be the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$, let $\sigma \in sol(\mathcal{I}, \mathcal{G}, Op)$ be a primitive solution, and let σ' be a subsequence of σ such that $\sigma' \in \Sigma^{nr}$. Moreover, let $h^1 = [\{(i : act_i) \mid act_i \in \sigma'\}, \bigwedge\{(i \prec j) \mid i, j \in \{1, \ldots, |\sigma'|\}, i < j\}]$ be the hybrid-plan representing σ'. First, suppose $\mathcal{I} \models \mathcal{G}$. Then, the theorem holds trivially, as $|\sigma'| = 0$ and $h^1 = [\emptyset, true]$ is a MNRMA for $\mathcal{H}$. Next, suppose $\mathcal{I} \not\models \mathcal{G}$. Then, observe that hybrid-plan h^1 is a minimal non-redundant hybrid-solution for $\mathcal{H}$. If h^1 is not a maximally-abstract hybrid-plan among all possible hybrid-plans for the set $\{\sigma'\}$, then according to Definition 18 (Maximally-Abstract), a more abstract hybrid-plan exists, that is, $MA(h^2, h^1, \{\sigma'\})$ holds for some hybrid-plan h^2, where $MA(h', h, \Sigma)$ (for any hybrid-plans $h = [s_h, \phi_h]$ and $h' = [s_{h'}, \phi_{h'}]$, and set of primitive plans $\Sigma \subseteq sol(h, \mathcal{I}, \mathcal{D})$) stands for the conditions in Definition 18 (i.e., $|s_{h'}| < |s_h|, d_1 \in refn(h', \mathcal{D})$, etc.).

Next, let $min(h)$ denote the set of minimal non-redundant hybrid-solutions $h_{min} = [s_{min}, \phi_{min}]$ for $\mathcal{H}$ that can be obtained from a hybrid-plan $h = [s_h, \phi_h]$ (that is $s_{min} \subseteq s_h$, and ϕ_{min} is obtained from ϕ_h by replacing with *true* every constraint that mentions some task label occurring in the set $s_h \setminus s_{min}$). Now, if there is a hybrid-plan $h^3 \in min(h^2)$ such that h^3 is a maximally-abstract hybrid-plan among all possible hybrid-plans for the set $\Sigma^{nr} \cap sol(h^3, \mathcal{I}, \mathcal{D})$, then h^3 is a MNRMA for $\mathcal{H}$, and the theorem holds. Otherwise, for each hybrid-plan $h^3 = [s^3, \phi^3] \in min(h^2)$, there must exist a hybrid-plan h^4 such that $MA(h^4, h^3, \Sigma^3)$ holds, where $\Sigma^3 = \Sigma^{nr} \cap sol(h^3, \mathcal{I}, \mathcal{D})$. This reasoning can be continued for $h^4 = [s^4, \phi^4]$ like we did before for hybrid-plan h^2. However, observe that since $|s^4| < |s^3|$ holds according to $MA(h^4, h^3, \Sigma^3)$, this reasoning can only be applied a finite number of times until some hybrid-plan h^n, with $h^n = [\{(n : t)\}, true]$ (recall hybrid-plans cannot mention state constraints) is reached for some compound task t. Hybrid-plan h^n is then a minimal non-redundant hybrid-solution for $\mathcal{H}$, and also a maximally-abstract hybrid-plan among all possible hybrid-plans for the set $\Sigma^{nr} \cap sol(h^n, \mathcal{I}, \mathcal{D})$. □

Unfortunately, it is not clear how one could compute an MNRMA for a hybrid planning problem, without considering all possible hybrid-plans.[3] Before we can develop, and show how to implement, a weaker notion than MNRMA that looks for the most "preferred" specialisation of a fixed hybrid-solution, in the next section we will develop an intermediate notion which, although still not implementable, is conceptually closer to our final implementable notion.

5.3 MNRMA Specialisations of Hybrid-Plans

Rather than exploring the set of all conceivable hybrid-plans to find one that is an ideal MNRMA hybrid-plan, we focus in this section on improving a *given* hybrid-plan, within the confines of its *refinements*, into one that contains no redundancy, while keeping it as abstract as possible. While such improved hybrid-plans are not necessarily (globally) ideal MNRMA hybrid-plans, they will still be ideal with respect to the space of hybrid-plans inherent in the refinements of the given hybrid-plan. A hybrid-plan found by improving a given hybrid-plan in this manner is called an *MNRMA specialisation* of the given hybrid-plan. MNRMA specialisations of a hybrid-plan are conceptually closer to the final implementable notion we will discuss in the next section, compared with the ideal MNRMA notion discussed in the previous section.

Intuitively, a *specialisation* is a non-redundant hybrid-plan inherent in a refinement, with the ordering enforced on tasks compatible with, but possibly more constrained than the ordering enforced on tasks in the refinement. More precisely, a specialisation of a refinement is a *subset* of the refinement, where the constraint formula of the subset has the following properties: *(i)* it does not contain state constraints, and *(ii)* it is an *extension* of the partially ordered constraint formula in the refinement, i.e., the constraint formula of the subset may contain additional ordering constraints compatible with those of the refinement. Therefore, a specialisation can be totally ordered even if its corresponding refinement is not. Observe that, by virtue of the specialisation being a subset of the refinement, all ordering constraints entailed by the refinement (on tasks in the subset) are also entailed by the specialisation. We do not include in the specialisation state constraints of the refinement because specialisations represent hybrid-plans, which do not account for state constraints. Nonetheless, by conforming to ordering constraints of the refinement, we ensure that the user's intent with respect to the ordering of tasks in the refinement is maintained in any extracted specialisation.

[3]Technically, one would consider only shorter plans than the one at hand.

It is worth noting the reason why specialisations are *subsets* of refinements. In situations where the given hybrid-solution is redundant, any refinement of the hybrid-solution will also be redundant in general.[4] By allowing specialisations to be subsets of refinements, it will be possible to remove any redundant tasks in refinements. Note, however, that removing redundant tasks in refinements is at the expense of user intent, which requires all tasks appearing in refinements to be intact.

For example, consider the (totally ordered) hybrid-solution $t_2 \cdot t_6$ for initial state $\{p\}$ and goal state $\{s, t\}$, created using tasks in the method-library of Figure 5.2. Observe that $t_2 \cdot t_6$ is a redundant hybrid-solution because its (only) primitive solution $a_1 \cdot a_2 \cdot a_3 \cdot a_3 \cdot a_6$ is redundant. Observe, also, that the redundancy cannot be eliminated by removing either of the tasks t_2 or t_6 – if t_2 is removed, the necessary actions a_1 and a_2 will also be removed; and if t_6 is removed, the necessary action a_6 will also be removed. However, if we consider a hybrid-solution (specialisation) at one level of abstraction below $t_2 \cdot t_6$, say $t_3 \cdot t_4 \cdot t_6$, we can now remove redundant task t_4 to obtain the minimal non-redundant hybrid-solution $t_3 \cdot t_6$. Moreover, observe that this hybrid-solution is maximally-abstract relative to all hybrid-plans that can be extracted from refinements of $t_2 \cdot t_6$.

Next, we define a specialisation of a given hybrid-plan as the combination of a subset of the tasks in a refinement of the hybrid-plan, with *(i)* all the constraints entailed on tasks in the subset by the refinement (ϕ_2^1), and *(ii)* possibly additional constraints that do not conflict with the constraint formula of the refinement (ϕ_2^2).

Definition 20. (Plan Specialisation) Let $\mathcal{D}$ be an HTN domain and let h_1 be a hybrid-plan. A hybrid-plan $h_2 = [s_2, \phi_2]$ is a *plan specialisation* of h_1 with respect to $\mathcal{D}$ if there exists formulas ϕ_2^1 and ϕ_2^2 with $\phi_2 \Leftrightarrow \phi_2^1 \wedge \phi_2^2$, and a ground instance and task label renaming $d = [s_d, \phi_d]$ of a refinement $d' \in refn(h_1, \mathcal{D})$ such that:

1. $s_2 \subseteq s_d$;

[4]Since a specialisation, being a hybrid-plan, will not contain any state constraints, a specialisation containing all tasks of the refinement may actually be less constrained than the refinement. In particular, the specialisation may be capable of producing a non-redundant solution that the refinement cannot produce. Therefore, in such cases, the specialisation may be non-redundant, even though its corresponding refinement is redundant.

2. ϕ_2^1 is the largest formula with unique constraints such that for all states $\mathcal{I}$, $sol(d, \mathcal{I}, \mathcal{D}) = sol([s_d, \phi_d \wedge \phi_2^1], \mathcal{I}, \mathcal{D})$; and

3. if there exists a state $\mathcal{I}$ such that $sol(d, \mathcal{I}, \mathcal{D}) \neq \emptyset$, then there also exists a state $\mathcal{I}'$ such that $sol([s_d, \phi_d \wedge \phi_2^2], \mathcal{I}', \mathcal{D}) \neq \emptyset$.

The set of all plan specialisations of h_1 with respect to $\mathcal{D}$ is denoted $spec(h_1, \mathcal{D})$. ■

In words: ϕ_2^1 is the formula of implicit and explicit constraints entailed by ϕ_d, i.e., those that can be added to ϕ_d without losing any of the primitive plan solutions that d can already produce in any initial state; and ϕ_2^2 is any formula of constraints that does not contradict ϕ_d, i.e., any formula that can be added to ϕ_d without losing all of the primitive plan solutions that d can already produce in some initial state.

Finally, a MNRMA specialisation is any specialisation of a given hybrid-solution that is *(i)* a minimal non-redundant hybrid-solution, and *(ii)* maximally-abstract among all *specialisations* of the given hybrid-solution.

Definition 21. (MNRMA Specialisation) Let $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ be a hybrid planning problem and let h be a hybrid-solution for $\mathcal{H}$. Then, h' is a *MNRMA specialisation* of h for $\mathcal{H}$ if and only if

1. $h' \in spec(h, \mathcal{D})$;

2. h' is a minimal non-redundant hybrid-solution for $\mathcal{H}$; and

3. h' is a maximally-abstract hybrid-plan among the set $spec(h, \mathcal{D})$, for the set $\Sigma \cap sol(h', \mathcal{I}, \mathcal{D})$, where Σ is the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. ■

The following theorem states that, given any hybrid-solution, there is, at least, one MNRMA specialisation for it.

Theorem 10. *Let $\mathcal{H}$ be a hybrid planning problem, and let h be a hybrid-solution for $\mathcal{H}$. Then, there exists an MNRMA specialisation of h for $\mathcal{H}$.*

Proof. Let Σ^{nr} be the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$, let $\sigma \in sol(\mathcal{I}, \mathcal{G}, Op)$ be a primitive solution, and let σ' be a subsequence of σ such that $\sigma' \in \Sigma^{nr}$. Moreover, let $h^1 = [\{(i : act_i) \mid act_i \in \sigma'\}, \bigwedge\{(i \prec j) \mid i, j \in \{1, \ldots, |\sigma'|\}, i < j\}]$ be the hybrid-plan representing σ'. First, suppose $\mathcal{I} \models \mathcal{G}$. Then, the theorem holds trivially, as $|\sigma'| = 0$ and $h^1 = [\emptyset, true]$ is a MNRMA

specialisation of h for $\mathcal{H}$. Next, suppose $\mathcal{I} \not\models \mathcal{G}$. We will now show that $h^1 \in spec(h, \mathcal{D})$, i.e., that $h^1 = [s_{h^1}, \phi_{h^1}]$ is a plan specialisation of h with respect to $\mathcal{D}$ (Definition 20).

Since there is a ground instance and task label renaming $d = [s_d, \phi_d]$ of some primitive task network $d' \in refn(h, \mathcal{D})$ such that $s_{h^1} \subseteq s_d$, the first condition of Definition 20 holds for hybrid-plan h^1. The second condition of the definition is also satisfied because h^1 is totally ordered, and consequently, we can always take as ϕ_2^1 in Definition 20 some formula constructed from constraints in ϕ_{h^1}. Finally, since σ' is a subsequence of σ, it follows that $sol([s_d, \phi_d \wedge \phi_{h^1}], \mathcal{I}, \mathcal{D}) \neq \emptyset$ holds — the constraints in ϕ_{h^1} are in agreement with plan σ. Therefore, the third condition of Definition 20 holds, and h^1 is indeed a plan specialisation of h with respect to $\mathcal{D}$.

Next, observe that h^1 is a minimal non-redundant hybrid-solution for $\mathcal{H}$. If h^1 is not a maximally-abstract hybrid-plan among $spec(h, \mathcal{D})$ for the set $\{\sigma'\}$, then according to Definition 18 (Maximally-Abstract), a more abstract hybrid-plan exists, that is, $MA(h^2, h^1, \{\sigma'\})$ holds for some hybrid-plan $h^2 \in spec(h, \mathcal{D})$, where $MA(h', h, \Sigma)$ (for any hybrid-plans $h = [s_h, \phi_h]$ and $h' = [s_{h'}, \phi_{h'}]$, and set of primitive plans $\Sigma \subseteq sol(h, \mathcal{I}, \mathcal{D})$) stands for the conditions in Definition 18 (i.e., $|s_{h'}| < |s_h|, d_1 \in refn(h', \mathcal{D})$, etc.).

Let $min(h)$ denote the set of minimal non-redundant hybrid-solutions $h_{min} = [s_{min}, \phi_{min}]$ for $\mathcal{H}$ that can be obtained from a hybrid-plan $h = [s_h, \phi_h]$ (that is $s_{min} \subseteq s_h$, and ϕ_{min} is obtained from ϕ_h by replacing with *true* every constraint that mentions some task label occurring in the set $s_h \setminus s_{min}$). Observe from Definition 20 that if a hybrid-plan h' is a plan specialisation of h with respect to $\mathcal{D}$, then each hybrid-plan in $min(h')$ is also a plan specialisation of h with respect to $\mathcal{D}$. Then, if there is a hybrid-plan h^3 such that $h^3 \in min(h^2)$, $h^3 \in spec(h, \mathcal{D})$, and h^3 is a maximally-abstract hybrid-plan among $spec(h, \mathcal{D})$ for the set $\Sigma^{nr} \cap sol(h^3, \mathcal{I}, \mathcal{D})$, hybrid-plan h^3 is a MNRMA specialisation for $\mathcal{H}$, and the theorem holds. Otherwise, for each hybrid-plan h^3 such that $h_3 \in min(h^2)$ and $h^3 \in spec(h, \mathcal{D})$, there must exist a hybrid-plan $h^4 \in spec(h, \mathcal{D})$ such that $MA(h^4, h^3, \Sigma^3)$ holds, where $\Sigma^3 = \Sigma^{nr} \cap sol(h^3, \mathcal{I}, \mathcal{D})$. This reasoning can be continued for $h^4 = [s^4, \phi^4]$ like we did before for hybrid-plan h^2. However, observe that since $|s^4| < |s^3|$ holds according to $MA(h^4, h^3, \Sigma^3)$, this reasoning can only be applied a finite number of times until some hybrid-plan h^n, with $h^n = [\{(n : t)\}, true]$ (recall hybrid-plans cannot mention state constraints) is reached for some compound task t. Hybrid-plan h^n is then a minimal non-redundant hybrid-solution for $\mathcal{H}$, and also a maximally-abstract hybrid-plan among $spec(h, \mathcal{D})$ for the set $\Sigma^{nr} \cap sol(h^n, \mathcal{I}, \mathcal{D})$. □

The next theorem shows the relationship between an ideal MNRMA hybrid-plan and an MNRMA specialisation of a hybrid-plan. It states that, whenever a specialisation of a hybrid-solution h is an ideal MNRMA hybrid-plan, it is also the case that the specialisation is a MNRMA specialisation of h.

Theorem 11. *Let h be a MNRMA hybrid-plan for a hybrid planning problem $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$. Let h' be a hybrid-solution, where h is a plan specialisation of h' with respect to $\mathcal{D}$. Then, h is a MNRMA specialisation of h' for $\mathcal{H}$.*

Proof. Since h is a MNRMA hybrid-plan for $\mathcal{H}$, we know that h is a minimal non-redundant hybrid-solution for $\mathcal{H}$, and that h is a maximally-abstract hybrid-plan among *all* possible minimal non-redundant hybrid-solutions Δ for the set $\Sigma \cap sol(h, \mathcal{I}, \mathcal{D})$, where Σ is the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. Since the set of plan specialisations $spec(h', \mathcal{D}) \subseteq \Delta$, hybrid-plan h must be a maximally-abstract hybrid-plan among $spec(h', \mathcal{D})$ for $\Sigma \cap sol(h, \mathcal{I}, \mathcal{D})$. Hence, h is a MNRMA specialisation of h' for $\mathcal{H}$. □

Like the ideal MNRMA notion, it is not clear how one could compute an MNRMA specialisation of a hybrid-plan, without considering all refinements of the given hybrid-plan. Therefore, in the next section we will present an even weaker, but implementable notion called *preferred specialisations* of a given hybrid-plan. This notion finds a preferred specialisation by exploring not the full set of refinements of a given hybrid-plan, but only the set of refinements captured within a single *decomposition* of the hybrid-plan.

5.4 Preferred Specialisations of Hybrid-Plans

Instead of improving a hybrid-solution by exploring *all* of its specialisations, in this section we focus on improving a hybrid-solution by exploring only the limited set of specialisations inherent in one of its "*decompositions*," and extracting a most abstract and non-redundant specialisation of the hybrid-solution from the limited set. Like we saw in the previous section, the particular hybrid-solution that we start from may have been produced by a first principles planner operating in the BDI domain.

We state the problem we are interested in solving as follows: given a, possibly redundant, hybrid-solution for a hybrid planning problem, together with one of its successful decompositions,

find a specialisation of the hybrid-solution that is both non-redundant and as abstract as possible, within the confines of the decomposition. We call such specialisations *preferred specialisations*.

It is important to note that a preferred specialisation is not always one that is an (ideal) MNRMA hybrid-plan as defined in Section 5.2. However, whenever a MNRMA hybrid-plan occurs in a decomposition of some given hybrid-plan h, then this MNRMA hybrid-plan will also be a preferred specialisation of h.

Intuitively, a *decomposition* (or decomposition trace) is a trace of the reductions performed on a hybrid-plan. Therefore, any hybrid-plan that produces a primitive plan solution will have a decomposition trace, which starts from the hybrid-plan and ends with the primitive task network corresponding to the primitive plan solution. Such decomposition traces that produce primitive plan solutions are called *successful* decomposition traces.

For example, suppose we are given a method library containing the following reductions: *(i)* task t_1 is reduced into labelled tasks $(2 : t_2)$ and $(3 : t_3)$ with constraint *true*, *(ii)* task t_2 is reduced into labelled tasks (actions) $(4 : a_4)$ and $(5 : a_5)$ with constraint $4 \prec 5$, and *(iii)* task t_3 is reduced into labelled tasks $(6 : a_6)$ and $(7 : a_7)$ with constraint $6 \prec 7$. Then, a possible decomposition trace of hybrid-plan $[\{(1 : t_1)\}, true]$ is the following:
$[\{(1 : t_1)\}, true] \cdot [\{(2 : t_2), (3 : t_3)\}, true] \cdot [\{(4 : a_4), (5 : a_5), (3 : t_3)\}, (4 \prec 5)] \cdot [\{(4 : a_4), (5 : a_5), (6 : a_6), (7 : a_7)\}, (4 \prec 5) \wedge (6 \prec 7)]$.

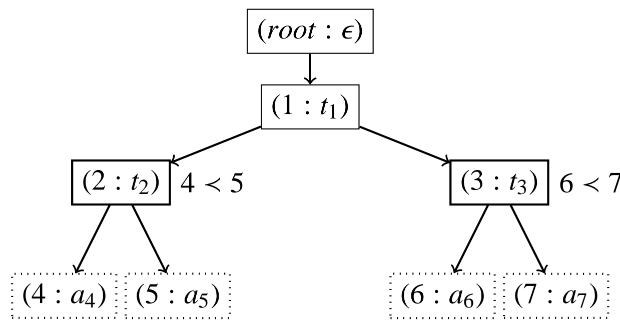


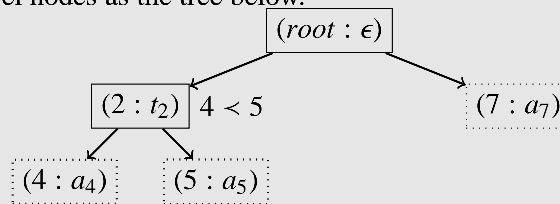
Figure 5.4: The decomposition tree corresponding to decomposition trace $[\{(1 : t_1)\}, true] \cdot [\{(2 : t_2), (3 : t_3)\}, true] \cdot [\{(4 : a_4), (5 : a_5), (3 : t_3)\}, (4 \prec 5)] \cdot [\{(4 : a_4), (5 : a_5), (6 : a_6), (7 : a_7)\}, (4 \prec 5) \wedge (6 \prec 7)]$. Dotted rectangles stand for primitive tasks/actions, and missing constraints stand for *true*.

It is easy to see that a decomposition trace induces a *decomposition tree*, i.e., intuitively, a tree depicting how compound tasks are reduced into other tasks and constraints. For example, the

decomposition tree induced by the decomposition trace above is shown in Figure 5.4. Observe that each node in the tree is a labelled task, and that each node is labelled with the constraints for its child nodes. Observe, further, that the children of the root node are the labelled tasks in the given hybrid-plan.

We can now be more specific about the aim of this section. Given a hybrid-plan h for a hybrid planning problem $\mathcal{H}$ and a decomposition tree $\mathcal{T}$ induced by a successful decomposition trace of h, we want to find a subtree $\mathcal{T}'$ of $\mathcal{T}$ that yields a perfect justification, but one that is not subsumed by some other subtree of $\mathcal{T}$ that yields the leaf-level nodes of $\mathcal{T}'$. We can then take the hybrid-plan at the top level (below root) of $\mathcal{T}'$ as the preferred specialisation of h for $\mathcal{H}$.

For example, suppose the decomposition trace shown earlier is a successful trace of hybrid-plan $h = [\{(1 : t_1)\}, true]$, and that $(4 : a_4) \cdot (5 : a_5) \cdot (7 : a_7)$ is a perfect justification for some hybrid planning problem $\mathcal{H}$ – i.e., labelled task $(6 : a_6)$ is redundant. Then, the preferred specialisation of h for $\mathcal{H}$ and the decomposition tree in Figure 5.4 is hybrid-plan $[\{(2 : t_2), (7 : a_7)\}, true]$ in the subtree of Figure 5.4 shown below. This is because *(i)* the subtree yields a perfect justification, and *(ii)* there is no other subtree of Figure 5.4 that is more abstract than the one below. Note that the subtree of Figure 5.4 with top-level hybrid-plan $[\{(2 : t_2), (3 : t_3)\}, true]$ is not more abstract than the one below, because the former tree does not yield exactly the same leaf level nodes as the tree below.



A subtree of the decomposition tree in Figure 5.4.

5.4.1 Formalisation

In order to develop an account of what a preferred specialisation for a hybrid-plan is, we will define three basic notions: *(i) decomposition trace*, *(ii) decomposition tree*, and *(iii)* a *cut* in a decomposition tree. Intuitively, a cut in a decomposition tree corresponds to a hybrid-plan – in particular, a cut is a subset of the nodes in the decomposition tree. Our aim is to find a most

abstract cut whose decomposition tree yields a perfect justification.

A *decomposition trace* of a task network is a sequence of ground task networks, where each task network d_i of the trace is a reduction of some task in the preceding task network d_{i-1}.

Definition 22. (Decomposition Trace) Let $\mathcal{D}$ be a HTN domain and let d be a task network. A *decomposition trace* of d relative to Me is a, possibly infinite, sequence of ground task networks $\lambda_d = d_1 \cdot \ldots \cdot d_n \cdot \ldots$, such that *(i)* $d_1 = d\theta$, and *(ii)* for each d_i, $i > 0$, it is the case that *(a)* $d_{i+1} = reduce(d_i, n, m)$, and *(b)* there is no common task label occurring in $s_{i+1} \setminus s_i$ and $d_1 \cdot \ldots \cdot d_i$, where n is a task label occurring in d_i and m is a ground instance of a method in Me. A finite decomposition trace $\lambda_d = d_1 \cdot \ldots \cdot d_n$ of d relative to Me is a *complete decomposition trace* if d_n is a primitive task network. A complete decomposition trace $\lambda_d = d_1 \cdot \ldots \cdot d_n$ of d relative to Me is a *successful decomposition trace* if $sol(d_n, \mathcal{I}, \mathcal{D}) \neq \emptyset$. ■

Observe that we force new labelled tasks added to a trace by a reduction, i.e., the set $s_{i+1} \setminus s_i$, to have different labels to those already occurring in the trace. This is done to ensure that task labels uniquely identify tasks in the decomposition trace. Observe, further, that a decomposition trace encodes a specific order on the reduction of tasks.

For example, consider again the following decomposition trace from the introduction:

$[\{(1:t_1)\}, true] \cdot [\{(2:t_2),(3:t_3)\}, true] \cdot [\{(4:a_4),(5:a_5),(3:t_3)\},(4 \prec 5)] \cdot [\{(4:a_4),(5:a_5),(6:a_6),(7:a_7)\},(4 \prec 5) \wedge (6 \prec 7)]$.

In this trace, task t_1 is reduced first, task t_2 is reduced second, and task t_3 is reduced last. Reducing task t_3 before t_2 will result in a different decomposition trace, namely, the one shown below:

$[\{(1:t_1)\}, true] \cdot [\{(2:t_2),(3:t_3)\}, true] \cdot [\{(2:t_2),(6:a_6),(7:a_7)\},(6 \prec 7)] \cdot [\{(4:a_4),(5:a_5),(6:a_6),(7:a_7)\},(4 \prec 5) \wedge (6 \prec 7)]$.

Finally, observe that the last task network of a complete decomposition trace can contain tasks for which no ordering is specified. This can be seen with tasks a_5 and a_6 in the above decomposition traces. Consequently, the last task network of a decomposition trace may be capable of producing more than one primitive plan solution.

A *decomposition tree* represents the structure of a decomposition trace, by depicting how compound tasks are reduced using methods, to other (child) tasks, and to constraints on child

tasks. In particular, a node of a decomposition tree is a labelled (compound or primitive) task, and each node is labelled with the constraints on its child tasks. We use ϵ-nodes to account for root nodes, and for (dummy) ϵ tasks, i.e., those having no precondition or effect. Recall from Section 5.1 that it is sometimes useful to have a method consisting of a single ϵ task – such a method, given a particular condition, amounts to "doing nothing." The following definition uses the notion of a *vertex-labelled tree* and other related notions given in Appendix B.

Definition 23. (Decomposition Tree) A *decomposition tree* $\mathcal{T} = \langle V, E, \ell_V \rangle$ of a task network $d = [s_d, \phi_d]$ relative to a HTN domain $\mathcal{D}$ is a vertex-labelled tree, where

1. for each node $u \in V$: *(i)* $u = (n : t)$, where n is a unique task label in the tree and t is a ground domain task or ϵ, and *(ii)* $\ell_V(u)$ is a ground constraint formula;
2. $root(\mathcal{T})$ is node $u = (root : \epsilon)$, $\ell_V(u) = \phi_d\theta$, and $children(u, \mathcal{T}) = s_d\theta$;
3. if $u = (n : t)$ is a non-root node in $\mathcal{T}$ such that $children(u, \mathcal{T}) = \{u_1\theta', \ldots, u_m\theta'\}$, with $\ell_V(u) = \phi\theta'$, then there exists a reduction $[s, \phi] \in red([\{(n : t)\}, true], \mathcal{D})$ of t where $s = \{u_1, \ldots, u_m\}$;
4. if $u \in leaves(\mathcal{T})$, then $\ell_V(u) = true$. ■

Note that, like decomposition traces, decomposition trees are also ground. Moreover, by the definition of a reduction, decomposition trees can always be constructed with arbitrary labels for the tasks (except for the root's children, specified in d). Moreover, note that, unlike decomposition traces, which encode a specific order on the reduction of tasks, decomposition trees are agnostic on *when* tasks are reduced. Therefore, different decomposition traces may induce the same decomposition tree, up to renaming of task labels. For example, both decomposition traces shown previously induce the decomposition tree in Figure 5.4.

Next, we define what it means for a decomposition tree to be *induced* from a decomposition trace. Basically, an induced decomposition tree of a decomposition trace is a vertex-labelled tree, where the vertices of the tree correspond to labelled tasks occurring in the trace, edges correspond to tasks introduced by reductions, and labels of nodes correspond to constraints introduced by reductions.

Definition 24. (Induced Decomposition Tree) Let $\lambda = [s_1, \phi_1] \cdot \ldots \cdot [s_k, \phi_k]$ be a decomposition trace of some task network relative to a method-library Me. Suppose V_λ is the set of labelled tasks

mentioned in λ, and $rt = (root : \epsilon)$. Then, an <u>*induced decomposition tree*</u> of λ is the vertex-labelled tree $\langle V_\lambda \cup \{rt\}, E, \ell_V \rangle$, where

$$
\begin{aligned}
E &= \bigcup_{u \in s_1} \Big(\{(rt, u)\} \cup edges^*(u) \Big); \\
edges^*(u) &= edges(u) \cup \bigcup_{(u,u') \in edges(u)} edges^*(u'); \\
edges(u) &= \{(u, u') \mid 0 < i < k,\ u' \in (s_{i+1} \setminus s_i),\ u \in s_i,\ u \notin s_{i+1}\};\ \textit{and} \\
\ell_V &= \{(rt, \phi_1)\} \cup \bigcup_{u \in V_\lambda} \{(u, \phi) \mid 0 < i < k,\ u \in s_i,\ u \notin s_{i+1},\ \phi_{i+1} = \phi_i \wedge \phi\} \cup \\
&\quad \bigcup_{u \in V_\lambda} \{(u, true) \mid (u, u') \notin E\}.^5
\end{aligned}
$$

■

Note that, like in Definition 22, set $s_{i+1} \setminus s_i$ is the set of new labelled tasks introduced by the reduction of labelled task u. Note, further, that the label of the root node is the constraint formula ϕ_1 of the first task network in λ, and that leaf nodes are assigned the label *true*.

It is not difficult to see that any induced decomposition tree of a decomposition trace is indeed a (valid) decomposition tree.

Lemma 10. *Let d be a task network, $\mathcal{D}$ be a HTN domain, $\lambda = d_1 \cdot \ldots \cdot d_n$ be a decomposition trace of d relative to Me, and $\mathcal{T}$ be a decomposition tree of d relative to $\mathcal{D}$. Then, the induced decomposition tree $\mathcal{T}_\lambda$ of λ is a decomposition tree of d relative to $\mathcal{D}$, and moreover, $\mathcal{T}$ is the induced decomposition tree of some decomposition trace of d relative to Me.*

Proof. See Appendix A.3. □

So far, decomposition trees are merely *syntactic* objects and therefore independent of any (initial) state—they describe legal syntactic ways of transforming tasks into other tasks with respect to the method library. We will now associate decomposition trees with an initial state $\mathcal{I}$; in particular, we will define what it means for a decomposition tree to be *executable* in $\mathcal{I}$, with respect to

[5] Actually, ϕ_i will have to be modified so that all occurrences of the task label in u are replaced appropriately with expressions of the form *first*[] or *last*[], as done in Definition 31 (p. 200). Also, since ℓ_V is a mapping from task labels to constraint formulas, ℓ_V is treated as a set of ordered pairs for convenience.

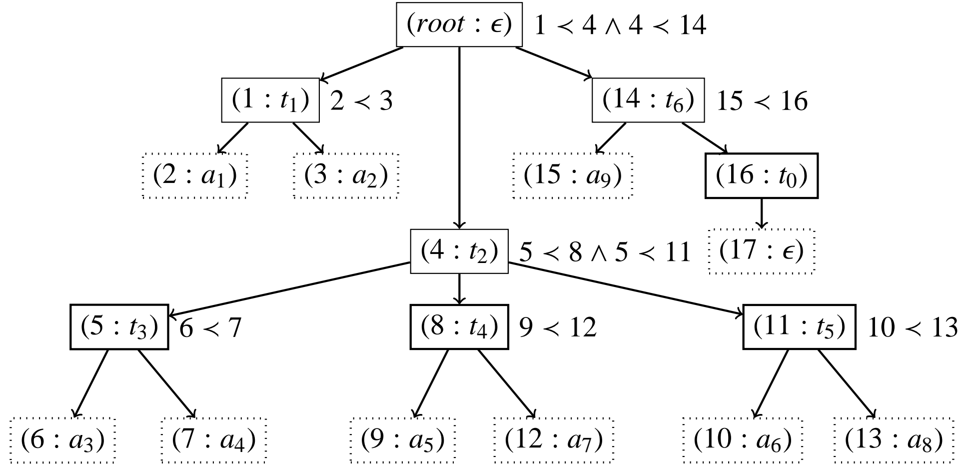


Figure 5.5: A complete decomposition tree $\mathcal{T}$ of task network $d = [\{(1 : t_1), (4 : t_2), (14 : t_6)\}, (1 \prec 4) \wedge (4 \prec 14)]$. Dotted rectangles stand for primitive tasks/actions or empty reductions (node $\langle(17 : \epsilon)\rangle$).

a primitive plan yielded by the tree. To do this, we first define the following preliminary notions: *complete decomposition tree*, *linearisation*, and *full decomposition tree*.

We say that a decomposition tree is *complete* if no leaf node represents a compound task. Notice, therefore, that in a complete decomposition tree, a leaf node can be a primitive task, or a node of the form $(n : \epsilon)$ corresponding to an ϵ task. We extract from a complete decomposition tree the set of non-ϵ nodes that are primitive as follows:

$$actions(\mathcal{T}) = \{(n : t) \mid (n : t) \in leaves(\mathcal{T}), t \neq \epsilon\}.$$

We say that a *linearisation* τ of a complete decomposition tree $\mathcal{T}$ is a permutation of the elements in $leaves(\mathcal{T}) \setminus \{(root : \epsilon)\}$, i.e., a labelled primitive plan built from exactly the non-root elements in $leaves(\mathcal{T})$. For example, a linearisation of the decomposition tree in Figure 5.5 is $(2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (12 : a_7) \cdot (10 : a_6) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$. Note that the ordering of elements in a linearisation is independent of any ordering constraints enforced on those elements in $\mathcal{T}$, and that a linearisation is independent of any (initial) state. Therefore, although a linearisation of a hybrid-plan's complete decomposition tree is a labelled primitive plan, the linearisation may not be a labelled primitive plan *solution* for the hybrid-plan at any initial state.

A *full decomposition tree* is the combination of a complete decomposition tree and one of its

linearisations. Formally, a *full decomposition tree*, denoted by $\mathcal{T}_\tau$, is a tuple $\langle \mathcal{T}, \tau \rangle$, where $\mathcal{T}$ is a complete decomposition tree, and τ is a linearisation of $\mathcal{T}$. Therefore, a full decomposition tree encodes not only how tasks are fully reduced to primitive tasks, but also how these may be ordered to form a labelled primitive plan.

Next, we define the final notion related to decomposition trees. We say that a full decomposition tree $\mathcal{T}_\tau$ is *executable* in an initial state $\mathcal{I}$ if *(i)* the tree is legal in $\mathcal{I}$, i.e., all constraints occurring in the decomposition tree are satisfied in $\mathcal{I}$; and *(ii)* the labelled primitive plan τ is executable in $\mathcal{I}$. Recall from Section 2.3.1 that $Res^*(act_1 \cdot \ldots \cdot act_n, \mathcal{I}, Op)$ is the state resulting from applying actions $act_1 \cdot \ldots \cdot act_n$ to the initial state $\mathcal{I}$, if it is possible to do so, and that it is undefined otherwise.

Definition 25. (Executable Decomposition Tree) Let $\mathcal{T}_\tau$ be a full decomposition tree, $\mathcal{I}$ be an initial state, and let Op be an operator-library. Then, $\mathcal{T}_\tau$ is *executable* in $\mathcal{I}$ relative to Op if *(i)* for all $u \in V(\mathcal{T})$, constraint formula $\ell_V(u)$ is *satisfied* in $\mathcal{T}_\tau$ relative to $\mathcal{I}$; and *(ii)* $Res^*(\tau, \mathcal{I}, Op)$ is defined. ■

To determine whether a constraint formula is satisfied, each constraint occurring in it needs to be evaluated based on the given initial state $\mathcal{I}$ and linearisation τ. More specifically, a constraint is evaluated by determining whether the primitives corresponding to task labels mentioned in the constraint are in the correct order in τ (in the case of ordering constraints), or whether certain conditions hold for the primitives corresponding to task labels mentioned in the constraint (in the case of state constraints), relative to $\mathcal{I}$ and τ.

Definition 26. (Satisfying a Constraint Formula) Let $\mathcal{T}_\tau$, with $\tau = u_1 \cdot \ldots \cdot u_m$ and $\mathcal{T} = \langle V, E, \ell_V \rangle$, be a full decomposition tree, let $\mathcal{I}$ be a state, and let Op be an operator library. Finally, for any $n \in \mathbb{N}_0$, the set of indexes $idx(n) = \{i \mid u = (n : t) \in V(\mathcal{T}),\ u' \in leaves(u, \mathcal{T}),\ i \in \{1, \ldots, m\},\ u' = u_i\}$.

Then, a ground constraint formula ϕ is *satisfied* in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and Op, denoted by $\langle \mathcal{T}_\tau, \mathcal{I}, Op \rangle \models \phi$ (or simply as $\langle \mathcal{T}_\tau, \mathcal{I} \rangle \models \phi$), if ϕ is satisfied, where the constraint formula is evaluated as follows:[6]

- $(c_1 = c_2)$ is true if c_1 and c_2 are the same constant symbols;
- $(n_1 \prec n_2)$ is true if $max(idx(n_1)) < min(idx(n_2))$;

[6]Expressions of the form $first[n_1, \ldots, n_k]$ and $last[n_1, \ldots, n_k]$ appearing in any constraint, e.g., $(last[n_1, \ldots, n_k] \prec n)$, evaluates to $min(\bigcup_{i \in \{1,\ldots,k\}} idx(n_i))$ and $max(\bigcup_{i \in \{1,\ldots,k\}} idx(n_i))$, respectively.

- (l, n_1) is true if $Res^*(u_1 \cdot \ldots \cdot u_{min(idx(n_1))-1}, \mathcal{I}, Op) \models l$, i.e., (ground) literal l is true in the state that results from applying to $\mathcal{I}$ the actions in τ up to the action immediately before n_1;
- (n_1, l) is true if $Res^*(u_1 \cdot \ldots \cdot u_{max(idx(n_1))}, \mathcal{I}, Op) \models l$;
- (n_1, l, n_2) is true if $Res^*(u_1 \cdot \ldots \cdot u_k, \mathcal{I}, Op) \models l$, for all $max(idx(n_1)) \leq k < min(idx(n_2))$; and
- logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic. ■

For example, consider the full decomposition tree $\mathcal{T}_\tau$, where $\mathcal{T}$ is the complete decomposition tree in Figure 5.5, and $\tau = (2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (10 : a_6) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (12 : a_7) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$. Suppose we want to determine whether the constraint formula of node $(4 : t_2)$, i.e., $(5 \prec 8) \wedge (5 \prec 11)$, is satisfied relative to τ (the initial state is not necessary in this example as we only deal here with ordering constraints). Observe that $(5 \prec 8)$ is satisfied in τ because all primitives corresponding to task label 5, i.e., $(6 : a_3)$ and $(7 : a_4)$, precede all primitives corresponding to task label 8, i.e., $(9 : a_5)$ $(12 : a_7)$. However, observe that constraint $(5 \prec 11)$ is not satisfied because one of the primitives corresponding to task label 11, i.e., $(10 : a_6)$, does not precede all primitives corresponding to task label 5. Consequently, formula $(5 \prec 8) \wedge (5 \prec 11)$ is also not satisfied.

On the other hand, if $\tau = (2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (12 : a_7) \cdot (10 : a_6) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$, then constraint $(5 \prec 8)$ and formula $(5 \prec 8) \wedge (5 \prec 11)$ are satisfied.

As shown in the example, the initial state is not necessary to determine whether a constraint formula is satisfied, provided the constraint formula has no state constraints. Hence, we sometimes write $\mathcal{T}_\tau \models \phi$, instead of $\langle \mathcal{T}_\tau, \mathcal{I} \rangle \models \phi$, if there are no state constraints in ϕ.[7]

It is useful to note the relationship between complete decomposition traces and complete decomposition trees, in particular, that a complete decomposition trace corresponds to a set of (induced) full decomposition trees.

Lemma 11. *Let $\mathcal{D}$ be a HTN domain, $\lambda = d_1 \cdot \ldots \cdot d_k$ be a complete decomposition trace of some task network relative to Me, and $\mathcal{T}$ be an induced (complete) decomposition tree of λ. Then, there*

[7] Notice that the truth value of any constraint in a full decomposition tree can be determined given the initial state. However, this is not necessarily the case for (non-full) decomposition trees, as the satisfaction of the constraints generally depends on a total-ordering of the primitive tasks.

exists a plan $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$ *if and only if there exists a full decomposition tree* $\mathcal{T}_\tau$ *that is executable in* $\mathcal{I}$ *relative to Op, where* $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$*.*

Proof. See Appendix A.3. □

Now we can define the third and final basic notion of this section, namely, a *cut* in a decomposition tree. Informally, a cut in a decomposition tree is a subset of the nodes in the tree which, together with the necessary constraints, can form a hybrid-plan. In particular, the set of nodes forming a cut does not contain any node that is a descendant of any other node in the set. For example, $\{(1 : t_1), (4 : t_2)\}$ in Figure 5.5 is a legal cut, but $\{(4 : t_2), (5 : t_3)\}$ is not because $(5 : t_3)$ is a descendant of $(4 : t_2)$.

Definition 27. (Cut) A *cut* in a decomposition tree $\mathcal{T}$ is a set of nodes $\pi \subseteq V(\mathcal{T})$, with $\pi \neq \{(root : \epsilon)\}$, such that for all $u, u' \in \pi$, with $u \neq u'$, it is the case that $(descendants(u, \mathcal{T}) \cup \{u\}) \cap (descendants(u', \mathcal{T}) \cup \{u'\}) = \emptyset$. ■

Given a cut π in a decomposition tree $\mathcal{T}$, we can form a new decomposition tree $\mathcal{T}'$ by projecting only on those nodes in π, trivially adding a node $(root : \epsilon)$ as root with π as its children, and adding a label $\ell_V((root : \epsilon)) = true$. Figure 5.6 shows the projected tree for the cut $\{(1 : t_1), (4 : t_2)\}$. Formally, the decomposition tree obtained by *projecting* on a cut $\pi \subseteq V(\mathcal{T})$ in a decomposition tree $\mathcal{T}$, denoted by $\mathcal{T}|_\pi$, is defined as follows:

$$
\begin{aligned}
&\mathcal{T}|_\pi = \langle V', E', \ell'_V \rangle, \text{where:}\\
&\quad \mathcal{T} = \langle V, E, \ell_V \rangle;\\
&\quad rt = (root : \epsilon);\\
&\quad V' = \{rt\} \cup \pi \cup \{u' \mid u' \in descendants(u, \mathcal{T}),\ u \in \pi\};\\
&\quad E' = \{(rt, u) \mid u \in \pi\} \cup \{(u, u') \mid (u, u') \in E \text{ and } u, u' \in V'\}; \text{and}\\
&\quad \ell'_V = \{(rt, true)\} \cup \{(u, \phi) \mid (u, \phi) \in \ell_V \text{ and } u \in V',\ u \neq rt\}.
\end{aligned}
$$

The notion of projection trivially generalises to full decomposition trees, denoted by $\mathcal{T}_\tau|_\pi$, by projecting in τ only the primitive tasks that are leaf nodes in $\mathcal{T}_\tau|_\pi$; in particular $\mathcal{T}_\tau|_\pi = \mathcal{T}'_{\tau'}$, where $\mathcal{T}' = \mathcal{T}|_\pi$, and $\tau' = \tau|_{leaves(\mathcal{T}')}$. For any set of labelled tasks π and sequence of labelled primitive tasks τ, $\tau|_\pi$ is defined as the largest subsequence τ' of τ such that for each task $u \in \tau'$, $u \in \pi$.

We have now provided most of the notions necessary for our final definition of a *preferred specialisation* of a hybrid-plan. Recall that a preferred specialisation of a hybrid-plan is one that is

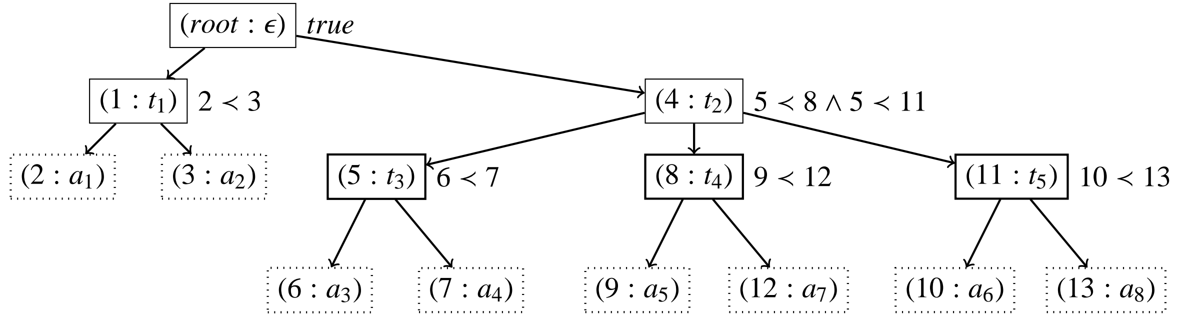


Figure 5.6: The decomposition tree obtained from the tree in figure 5.5 by projecting on the cut $\{(1 : t_1), (4 : t_2)\}$.

both non-redundant and as abstract as possible within the confines of the decomposition. Guided by the notion of maximal-abstraction in Section 5.2.2, we first define the notion of *dominance* which, intuitively, states that some cuts are more abstract than others. More specifically, a cut π' dominates a cut π if π', together with its descendants, contains π, and π' produces exactly the same non-ϵ primitive tasks as those produced by π.

Definition 28. (Dominance) Given two cuts π' and π in a decomposition tree $\mathcal{T}$, cut π' *dominates* π in $\mathcal{T}$ if $\pi \subseteq \bigcup_{u \in \pi'} descendants(u, \mathcal{T}) \cup \pi'$, and $actions(\mathcal{T}|_{\pi'}) = actions(\mathcal{T}|_{\pi})$. ■

For example, cut $\pi_1 = \{(4 : t_2)\}$ dominates cut $\pi_2 = \{(5 : t_3), (8 : t_4), (11 : t_5)\}$ in Figure 5.5 because the latter occurs in the descendants of the former, and they both yield the same non-ϵ primitive tasks. Observe that whenever a cut π' dominates a cut π, any compound task that is a descendant of π', but not in π nor its descendants, only yields ϵ tasks.

We can now use cuts and the associated notions to define what the preferred specialisations of hybrid-plans are. For convenience, we use $decsol(h, \mathcal{H})$ to denote the set of full decomposition trees $\mathcal{T}_\tau$ of a hybrid-plan h relative to a domain $\mathcal{D}$, where *(i)* $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to Op, and *(ii)* $\tau \in sol(\mathcal{I}, \mathcal{G}, Op)$, i.e., the linearisation τ achieves the goal state $\mathcal{G}$. Moreover, given a cut π in a full decomposition tree $\mathcal{T}_\tau$, we define the ordering constraints implied by $\mathcal{T}_\tau$ on π as *true* if $\pi = \emptyset$, and otherwise as follows:

$$\Phi[\mathcal{T}_\tau, \pi] = \bigwedge\nolimits_{\{n_1 \prec n_2 |\ (n_1:t_1),(n_2:t_2) \in \pi, \mathcal{T}_\tau \models n_1 \prec n_2\}} .$$

For example, suppose we are given the cut $\pi = \{(5 : t_3), (8 : t_4), (11 : t_5)\}$ and full decomposition tree $\mathcal{T}_\tau$, where $\mathcal{T}$ is the decomposition tree in Figure 5.5, and

$\tau = (2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (10 : a_6) \cdot (12 : a_7) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$. Now, since all primitives corresponding to labelled task $(5 : t_3)$, i.e., $(6 : a_3)$ and $(7 : a_4)$, occur before all primitives corresponding to labelled task $(8 : t_4)$, i.e., $(9 : a_5)$ and $(12 : a_7)$, we can conclude that ordering constraint $5 \prec 8$ holds. Similarly, we can conclude that $5 \prec 11$ also holds. However, ordering constraint $8 \prec 11$ does not hold, because the primitive $(12 : a_7)$ corresponding to $(8 : t_4)$ does not precede the primitive $(10 : a_6)$ corresponding to $(11 : t_5)$, i.e., the primitives corresponding to $(8 : t_4)$ and $(11 : t_5)$ overlap in τ. Therefore, we can conclude that the formula of ordering constraints implied by $\mathcal{T}_\tau$ on π is $\Phi[\mathcal{T}_\tau, \pi] = 5 \prec 8 \wedge 5 \prec 11$.

A preferred specialisation corresponds to a cut π in the given full decomposition tree, where the cut yields a perfect justification, and the cut is not dominated by any other cut in the tree – i.e., the cut is as abstract as possible in the tree.

Definition 29. (Preferred Specialisation) Let $\mathcal{H}$ be hybrid planning problem, h be a hybrid-solution for $\mathcal{H}$, and let $\mathcal{T}_\tau \in decsol(h, \mathcal{H})$. Then, a hybrid-plan h_π is a _preferred specialisation_ of h within $\mathcal{T}_\tau$ for $\mathcal{H}$ if $h_\pi = [\pi, \Phi[\mathcal{T}_\tau, \pi]]$ for some cut π in $\mathcal{T}_\tau$ such that

1. the projected linearisation $\tau|_{actions(\mathcal{T}|_\pi)}$ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$;
2. the projected full decomposition tree $\mathcal{T}_\tau|_\pi$ is executable in $\mathcal{I}$ relative to Op;
3. there is no cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to Op. ■

The third condition ensures that a cut π' that contains π, but also contains other compound tasks that lead to ϵ tasks, is not preferred over π. However, observe that it may be the case that π' is also a preferred specialisation, even if it contains tasks that lead to ϵ tasks. Therefore, preferred specialisations that do not contain any compound tasks leading to ϵ tasks are called *minimal* preferred specialisations.

Let us illustrate minimality with an example. Observe that hybrid-plan $h = [\{(1 : t_1), (4 : t_2), (16 : t_0)\}, (1 \prec 4) \wedge (4 \prec 16)]$ may be a preferred specialisation for a full decomposition tree corresponding to the tree in Figure 5.5, provided the hybrid-plan

can yield a perfect justification within the full decomposition tree. However, notice that hybrid-plan h is not minimal – compound task $(16 : t_0)$ is reduced into an ϵ task, thereby allowing a subset of the hybrid-plan, namely, $[\{(1 : t_1), (4 : t_2)\}, (1 \prec 4)]$ to also be a preferred specialisation.

Formally, a preferred specialisation $[s, \phi]$ of a hybrid-plan h within a full decomposition tree $\mathcal{T}_\tau$ for a hybrid planning problem $\mathcal{H}$ is a *minimal* preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$ if there does not exist a preferred specialisation $[s' \subset s, \phi']$ of h within $\mathcal{T}_\tau$ for $\mathcal{H}$, where ϕ' is obtained from ϕ by replacing all (ordering) constraints that mention some task label in $(s \setminus s')$ with *true*.

The following result guarantees that there is always a preferred specialisation of a hybrid-solution. In particular, whenever a full decomposition tree that achieves a given goal state exists for a hybrid-plan h, a preferred specialisation will also exist for h.

Theorem 12. *Let $\mathcal{H}$ be a hybrid planning problem, and let h be a hybrid-plan. If $\mathcal{T}_\tau \in decsol(h, \mathcal{H})$, then there exists at least one preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.*

Proof. Take $\mathcal{T}_\tau \in decsol(h, \mathcal{H})$, that is, $\tau \in sol(\mathcal{I}, \mathcal{G}, Op)$ is a primitive solution for $\mathcal{H}$. Then, it follows from Lemmas 10 and 11 that $\tau \in sol(h, \mathcal{I}, \mathcal{D})$ is also a primitive plan solution for h. Take τ' to be a subsequence of τ that is a perfect justification for $C = \langle \mathcal{I}, \mathcal{G}, Op \rangle$. Let π be the "low-level" cut corresponding to τ', i.e., $\pi = \{(n : t) \mid (n : t) \in \tau'\}$. If $\mathcal{I} \models \mathcal{G}$, then $\pi = \emptyset$ and hybrid-plan $h_\pi = [\emptyset, true]$ is a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.

Suppose, however, that $\mathcal{I} \not\models \mathcal{G}$, and consider hybrid-plan $h_\pi = [\pi, \Phi[\mathcal{T}_\tau, \pi]]$. Since $\tau|_{actions(\mathcal{T}|_\pi)} = \tau'$ is a perfect justification for C, the first condition of Definition 29 (Preferred Specialisation) is met for hybrid-plan h_π to be considered a preferred specialisation (of hybrid-plan h within $\mathcal{T}_\tau$ for $\mathcal{H}$). Moreover, since the label (i.e., constraint formula) of the root node in $\mathcal{T}_\tau|_\pi$ is *true*, and $Res^*(\tau', \mathcal{I}, Op)$ is defined (i.e., the state resulting from applying τ' in $\mathcal{I}$), it follows that $\mathcal{T}_\tau|_\pi$ is executable in $\mathcal{I}$ relative to Op (Definition 25). Therefore, the second condition of Definition 29 is also met for hybrid-plan h_π.

Suppose that h_π does not meet the third condition of Definition 29, that is, there exists a cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to Op. Since π' dominates π in $\mathcal{T}$, it is the case, from Definition 28 (Dominance), that $actions(\mathcal{T}|_{\pi'}) = actions(\mathcal{T}|_\pi)$, and therefore, that $\tau|_{actions(\mathcal{T}|_{\pi'})} = \tau'$ is a perfect justification for C. If hybrid-plan

$[\pi', \Phi[\mathcal{T}_\tau, \pi']]$ is not a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$, then there must exist yet another cut π'' in $\mathcal{T}$, with $|\pi''| < |\pi'|$, that dominates π' in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi''}$ is executable in $\mathcal{I}$ relative to Op. This reasoning can be continued for π'' like we did before for cut π'. However, since $|\pi''| < |\pi'|$, this reasoning can only be applied a *finite* number of times, until some cut $\pi^{top} \subseteq children((root : \epsilon), \mathcal{T})$ is reached. In such a case, since no strict subset of π^{top} can dominate π^{top}, hybrid-plan $[\pi^{top}, \Phi[\mathcal{T}_\tau, \pi^{top}]]$ is then the preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$. □

Note that, since the dominance relation among cuts is not total, and there may exist more than one perfect justification that can be extracted from a linearisation, there may actually be more than one preferred specialisation for a hybrid-plan within a given full decomposition tree.

Recall that our (ideal) MNRMA hybrid-plan (Definition 19) essentially defined a non-redundant hybrid-plan that is as abstract as possible *among all conceivable hybrid-plans*. A preferred specialisation, however, is one that is non-redundant and as abstract as possible among only the hybrid-plans that occur in a decomposition tree of a given hybrid-plan. One would expect, then, that whenever a MNRMA hybrid-plan occurs in a decomposition tree of a given hybrid-plan, then the MNRMA hybrid-plan is also a preferred specialisation of the given hybrid-plan. This is what the next theorem states. For convenience, we use $decsol^{nr}(h, \mathcal{H}) \subseteq decsol(h, \mathcal{H})$ to denote the set of full decomposition trees $\mathcal{T}_\tau$ where $\tau|_{actions(\mathcal{T})}$ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$.

Theorem 13. *Let h be a hybrid-solution for hybrid planning problem $\mathcal{H}$, and let $\mathcal{T}_\tau \in decsol(h, \mathcal{H})$. Suppose that there exists a cut π in $\mathcal{T}_\tau$ such that $h_\pi = [\pi, \Phi[\mathcal{T}_\tau, \pi]] \in MNRMA(\mathcal{H})$ and such that there is a decomposition in $decsol^{nr}(h_\pi, \mathcal{H})$ that is equivalent to $\mathcal{T}_\tau|_\pi$, modulo their root nodes. Then, hybrid-plan h_π is a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.*

Proof. Let $\sigma = \tau|_{actions(\mathcal{T}|_\pi)}$ be the projected linearisation of actions representing cut π. Observe that *(i)* σ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$; *(ii)* $\sigma \in sol(h_\pi, \mathcal{I}, \mathcal{D})$ follows from Lemmas 10 and 11; and *(iii)* the projected full decomposition tree $\mathcal{T}_\tau|_\pi$ is executable in $\mathcal{I}$ relative to Op.

Due to *(i)* and *(iii)*, the first two conditions in the definition of a preferred specialisation (Definition 29) are met by hybrid-plan h_π. Next, we will prove that the third condition in Definition 29 is also met by hybrid-plan h_π. On the contrary, suppose that there does exist a cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}'_{\tau'}$ is executable in $\mathcal{I}$ relative to Op, where $\mathcal{T}' = \mathcal{T}|_{\pi'}$ and $\tau' = \tau|_{leaves(\mathcal{T}')}$.

Informally, we shall prove below that since π' dominates π, a more abstract hybrid-plan than

π exists, namely, the one corresponding to π'. Consequently, π is not maximally abstract, which contradicts our assumption that $h_\pi \in \text{MNRMA}(\mathcal{H})$. We will now show this contradiction in more detail. Observe from Definition 18 (Maximally-Abstract), that to show $h_\pi \notin \text{MNRMA}(\mathcal{H})$, it is sufficient to show that the following four conditions hold for $h_{\pi'} = [\pi', \mathit{true}]$ (we take $h' = h_{\pi'}$ for Definition 18):

$$\hat{d}_1 \in \mathit{refn}(h_{\pi'}, \mathcal{D}); \tag{5.1}$$

$$\hat{d}_2 = [s_{\hat{d}_2}, \phi_{\hat{d}_2}] \text{ is a ground instance of } \hat{d}_1 \text{ such that } s_{\hat{d}_2} \supseteq \pi; \tag{5.2}$$

$$\hat{d}_3 = [s_{\hat{d}_2}, \phi_{\hat{d}_2} \wedge \Phi[\mathcal{T}_\tau, \pi]]; \text{ and} \tag{5.3}$$

$$\sigma \in \mathit{sol}(\hat{d}_3, \mathcal{I}, \mathcal{D}). \tag{5.4}$$

We start by showing that Equations 5.1 and 5.2 hold. First, from Lemma 10, there must exist a (complete) decomposition trace $d'_1 \cdot \ldots \cdot d'_k$ of $h_{\pi'}$ relative to *Me* such that $\mathcal{T}'$ is the induced (complete) decomposition tree of the trace, with $d'_1 = h_{\pi'}$. Second, as illustrated in (Erol et al., 1994), observe from the definition of a reduction (Definition 31, p. 200) that it does not matter in which order reductions are performed on a task network. Finally, since π' dominates π, it is not difficult to see that there is a decomposition trace that, informally, "goes through" π, namely

$$d'_1 = [s_1, \phi_1] \cdot \ldots \cdot [s_j, \phi_j] \cdot \ldots \cdot d'_k = [s_k, \phi_k],$$

where *(a)* $s_1 = \pi'$; *(b)* $\pi \subseteq s_j$; and *(c)* $j \in \{2, \ldots, k\}$ (note that $j \neq 1$ because $|\pi| > |s_1|$). Then, since task network $[s_j, \phi_j]$ is a ground instance of some refinement (Definition 17, p. 131) $\hat{d}_1 \in \mathit{refn}(h_{\pi'} = [s_1, \phi_1], \mathcal{D})$ such that $\pi \subseteq s_j$ holds, Equations 5.1 and 5.2 also hold.

Finally, let us show that Equations 5.3 and 5.4 hold. Let $\sigma' = \tau'|_{\mathit{actions}(\mathcal{T}')}$. Since π' dominates π in $\mathcal{T}$, we know that $\sigma = \sigma'$. Then, it follows from Lemma 11 that $\sigma \in \mathit{sol}([s_j, \phi_j], \mathcal{I}, \mathcal{D})$. Next, due to condition *(ii)* at the start of the proof, and since $\Phi[\mathcal{T}_\tau, \pi]$ is the conjunction of ordering constraints entailed by σ on elements in π, it follows that $\sigma \in \mathit{sol}([s_j, \phi_j \wedge \Phi[\mathcal{T}'_{\tau'}, \pi])$ holds. Therefore, Equations 5.3 and 5.4 also hold, and consequently, h_π is not a maximally-abstract hybrid-plan, and $h_\pi \notin \text{MNRMA}(\mathcal{H})$, which is a contradiction. □

In the next section, we will show how a preferred specialisation can be obtained, given a hybrid planning problem and hybrid-solution. In particular, we will first show how a full decomposition

tree can be obtained from the hybrid-solution, and then we will provide a bottom-up algorithm that obtains a preferred specialisation from the decomposition tree.

5.5 Computing Preferred Specialisations

As mentioned before, a preferred specialisation of a hybrid-plan h, relative to a hybrid planning problem, is a most abstract non-redundant hybrid-plan that can be extracted from a given *full decomposition tree* $\mathcal{T}_\tau$ *of* h; in particular, τ is a *labelled* primitive plan that achieves a given *goal state*.

We obtain such a full decomposition tree by inducing (as in Definition 24) a decomposition tree from a decomposition trace of h, where the trace produces a labelled primitive plan that achieves a given goal state, in addition to solving h. To do this, we need to address three issues. First, we need to extend the UMCP HTN algorithm of (Erol et al., 1996) so that it returns a *labelled* primitive plan solution, rather than a (un-labelled) primitive plan solution. This extension is trivial, as all we need to do is ensure that the procedure which computes the *completion* ($comp(d, \mathcal{I}, \mathcal{D})$) of the final primitive task network returns primitive plan solutions with the labels of primitive tasks left intact. Second, we need to ensure that any primitive plan solution found, for a given initial task network and initial state, also achieves a given *goal state*. Finally, we need to extend the UMCP algorithm so that, in addition to returning a primitive plan solution, it returns the *decomposition trace* that produces the primitive plan solution.

The second issue can be addressed by adding a constraint to the constraint formula of a given initial task network d, requiring the given goal state $\mathcal{G}$ to hold in the state immediately after the last action in any primitive plan solution for d. In particular, we obtain a task network d' by adding a conjunct to the constraint formula of the given task network $d = [s, \phi]$, as follows:

$$d' = [s, \phi \wedge \bigwedge_{l \in \mathcal{G}} (last[n_1, \ldots, n_k], l)], \text{ where } \{n_1, \ldots, n_k\} \text{ is the set of task labels occurring in } s.$$

Recall that $last[n_1, \ldots, n_k]$ is the task that occurs last among the tasks in $\{n_1, \ldots, n_k\}$, relative to a primitive plan solution for d.

To address the third issue, we extend UMCP in the following manner: *(i)* we keep track of the sequence of reductions performed during the HTN planning process (recall that the HTN planning process involves reducing compound tasks in the initial task network repeatedly, until only primi-

tive tasks remain); *(ii)* we make the resulting sequence into a decomposition trace, by replacing all variables occurring in the sequence in a consistent manner; and *(iii)* we return the decomposition trace together with the (labelled) primitive plan solution.

After a decomposition trace and labelled primitive plan solution are obtained, the trace and solution are simply combined to obtain a full decomposition tree.

Next, we will show how a preferred specialisation can be computed from a full decomposition tree, given a hybrid-solution h and hybrid planning problem $\mathcal{H}$. Algorithm 5.1 computes a preferred specialisation. Basically, the algorithm works bottom-up, starting at the leaf-level with a labelled primitive perfect justification plan τ' (line 1), and repetitively abstracting out one or more steps into a higher-level more abstract step (lines 3-9). Once no more abstractions are possible, the corresponding constraints entailed by the decomposition tree for the final steps are calculated (step 11) and the final hybrid-plan returned.

For example, a possible value for the perfect justification τ' (line 1), relative to some hybrid planning problem and the decomposition tree in Figure 5.5, could be $(2 : a_1) \cdot (9 : a_5) \cdot (10 : a_6) \cdot (12 : a_7) \cdot (13 : a_8) \cdot (15 : a_9)$—that is, actions $(3 : a_2)$, $(6 : a_3)$, $(7 : a_4)$, and $(17 : \epsilon)$ are redundant for achieving the goal.

Algorithm 5.1 Find-Preferred-Specialisation($h, \mathcal{H}, \mathcal{T}_\tau$)

Input: Hybrid-solution h, hybrid planning problem $\mathcal{H}$, $\mathcal{T}_\tau \in decsol(h, \mathcal{H})$, where $\mathcal{T} = \langle V, E, \ell_V \rangle$.
Output: A preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.

1: $\tau' \Leftarrow$ Get-Perfect-Justification($\tau, \mathcal{H}$) // As in (Fink and Yang, 1992); ignore ϵ tasks
2: $\pi \Leftarrow \{(n : t) \mid (n : t) \in \tau'\}$
3: **for** $\ell \Leftarrow 1$ **to** $height(\mathcal{T}_\tau) - 1$ **do** // Leaves are at level 0
4: **for** each node u at level ℓ in tree $\mathcal{T}$ **do**
5: **if** $children(u, \mathcal{T}) \subseteq \pi$ and $\langle \mathcal{T}_\tau|_\pi, \mathcal{I} \rangle \models \ell_V(u)$ **then** // $\ell_V(u)$ is satisfied in $\mathcal{T}_\tau|_\pi$ relative to $\mathcal{I}$
6: $\pi \Leftarrow (\pi \setminus children(u, \mathcal{T})) \cup \{u\}$ // Replace u's children with u
7: **end if**
8: **end for**
9: **end for**
10: $\pi \Leftarrow \pi \setminus \Delta \Leftarrow \{u \mid u \in \pi, \text{all leaves of } \mathcal{T}|_{\{u\}} \text{ are } \epsilon \text{ nodes}\}$
11: $\phi \Leftarrow \Phi[\mathcal{T}_\tau, \pi]$ // As defined just before Definition 29
12: **return** $[\pi, \phi]$

At any point in time, the algorithm maintains a "current" *cut* π (initially, a perfect justification). In line 4, a node u in the tree is selected for abstraction. If all the children of u are part of the current cut and the constraints required to decompose u into its children are indeed satisfied (line 5), then

all the children of u are abstracted out into node u itself (line 6). Therefore, the abstraction process relies not only on the children of a node being present in the current cut, but also on it being possible to satisfy the parent's constraint formula with respect to the current cut.

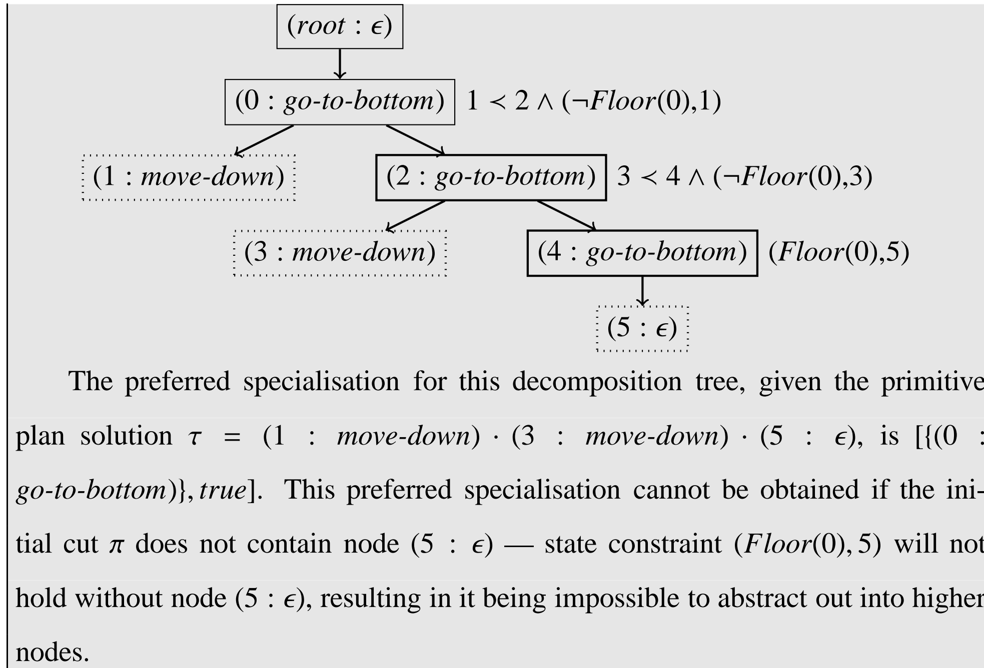
For example, if, as before, $\tau' = (2 : a_1) \cdot (9 : a_5) \cdot (10 : a_6) \cdot (12 : a_7) \cdot (13 : a_8) \cdot (15 : a_9)$, then at the end of the first iteration of the inner loop, a possible value for π is $\{(2 : a_1), (8 : t_4), (10 : a_6), (13 : a_8), (15 : a_9)\}$, that is, $(9 : a_5)$ and $(12 : a_7)$ have been replaced with their parent $(8 : t_4)$. However, for the same initial value of τ', observe that it will not be possible for π to have the value $\{(1 : t_1), (9 : a_5), (10 : a_6), (12 : a_7), (13 : a_8), (15 : a_9)\}$ at the end of the first iteration of the inner loop, because one of the children of $(1 : t_1)$, namely $(3 : a_2)$, does not exist in τ'.

We will now explain the rationale behind ignoring ϵ tasks in line 1. In this line, function Get-Perfect-Justification will consider all ϵ nodes in τ as redundant, because ϵ tasks have no effects. However, as explained before (Section 5.1), ϵ tasks are dummy tasks with no precondition or postcondition, which use up negligible resources during execution. Consequently, considering them as redundant tasks is unnecessary. More importantly, although ϵ tasks are technically redundant, they are special tasks which are necessary in order to maintain "links" to, for example, recursive compound tasks. This is illustrated by the following example.

Consider again the elevator domain example from Section 5.1. The example consists of the following two methods for handling the compound task *go-to-bottom*, which keeps moving down one floor until the ground floor (floor 0) is reached:

$(go\text{-}to\text{-}bottom, [\{(1 : move\text{-}down), (2 : go\text{-}to\text{-}bottom)\}, (1 \prec 2) \wedge (\neg Floor(0), 1)])$
$(go\text{-}to\text{-}bottom, [\{(1 : \epsilon)\}, (Floor(0), 1)])$.

Now, suppose the elevator is initially at the second floor. The decomposition tree for hybrid-plan $[\{go\text{-}to\text{-}bottom\}, true]$ is shown in the figure below.



The preferred specialisation for this decomposition tree, given the primitive plan solution $\tau = (1 : \textit{move-down}) \cdot (3 : \textit{move-down}) \cdot (5 : \epsilon)$, is $[\{(0 : \textit{go-to-bottom})\}, \textit{true}]$. This preferred specialisation cannot be obtained if the initial cut π does not contain node $(5 : \epsilon)$ — state constraint $(\textit{Floor}(0), 5)$ will not hold without node $(5 : \epsilon)$, resulting in it being impossible to abstract out into higher nodes.

Consequently, HTN ϵ tasks should not be removed from τ when finding a perfect justification of τ.

It is not difficult to see that the abstraction process can be carried on bottom-up, by performing the abstraction of all nodes at level k before abstracting to nodes at level $k + 1$. Eventually, the hybrid-plan computed as a preferred specialisation for our example based on Figure 5.5 would be $h = [s, \phi]$, where:

$$s = \{(2 : a_1), (8 : t_4), (11 : t5), (14 : t_6)\};$$

$$\phi = 2 \prec 8 \wedge 2 \prec 11 \wedge 8 \prec 14 \wedge 11 \prec 14 \wedge 2 \prec 14.$$

Notice that this is a partial-order plan, as the execution of compound tasks 8 and 11 may be interleaved (and, in fact, they are in $\mathcal{T}_\tau$).

Finally, we will explain the rationale behind line 10. Since nodes reduced into ϵ tasks (e.g., $(16 : t_0)$) may be abstracted, that is, added to the current cut π, we need to remove such trivially abstracted nodes from the final cut, if they are still present there (i.e., if they were not abstracted out). We do this because we are interested in finding *minimal* hybrid-plans, i.e., those that only include compound tasks that contribute to the perfect justification.

The algorithm can be proved correct with respect to Definition 29. In fact, it computes not any preferred specialisation, but *minimal* ones.

Theorem 14. *Algorithm 5.1 always terminates and returns a minimal preferred specialisation of hybrid-solution h within $\mathcal{T}_\tau$ for $\mathcal{H}$.*

Proof. Termination (of loops in lines 3 and 4) follows trivially by the fact that the tree (and its height) is finite. We will now prove that the algorithm returns a preferred specialisation of hybrid-solution h within $\mathcal{T}_\tau$ for $\mathcal{H}$. We will first claim that any value of cut π, after line 2 and before line 10, conforms to the first two conditions of Definition 29 (with respect to $\mathcal{T}_\tau$ and $\mathcal{H}$). Since the only line in the algorithm where π is modified after it is constructed is line 6, we will prove this claim by induction on the number k of times line 6 of the algorithm is executed.

For the base case, we take $k = 0$. Then, $\pi = \{(n : t) \mid (n : t) \in \tau'\}$ is the "low-level" cut of labelled primitive tasks. Since the projected linearisation $\tau|_{actions(\mathcal{T}|_\pi)}$ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, Op\rangle$, and since the projected full decomposition tree $\mathcal{T}_\tau|_\pi$ — in which all constraint formulas are *true* — is executable in $\mathcal{I}$ relative to Op, the first two conditions of Definition 29 hold for π.

Assume that the claim holds if $k \leq x$, for some $x \in \mathbb{N}_0$. Finally, let us take $k = x + 1$. Let π_{k-1} be the value of cut π after line 6 is executed $k - 1$ times, and let π_k be the value of cut π after line 6 is executed k times. Then, from the induction hypothesis, we know that $\pi_k = (\pi_{k-1} \setminus children(u, \mathcal{T})) \cup \{u\}$, for some node u in the decomposition tree, and that cut π_{k-1} conforms to the first two conditions of Definition 29. Since the only new task in π_k (compared with π_{k-1}) is u, and $\langle \mathcal{T}_\tau|_{\pi_{k-1}}, \mathcal{I}\rangle \models \ell_V(u)$ holds according to line 5, it follows that the first two conditions of Definition 29 also hold for π_k. Therefore, our claim holds.

Next, let π be any value of cut π immediately before line 10 in the algorithm. We will next show that the third and final condition of Definition 29 holds for cut π. Let S denote the following statement: there does not exist a node $u \in V(\mathcal{T})$, with $u \neq (root : \epsilon)$, such that $children(u, \mathcal{T}) \subseteq \pi$ and $\langle \mathcal{T}_\tau|_\pi, \mathcal{I}\rangle \models \ell_V(u)$. Observe from the algorithm that S holds. Then, to show that the third condition of the definition holds for π, it is sufficient to prove that S entails that there does not exist a cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to Op.

Let us assume the contrary (i.e., that S holds but that the third condition of the definition does not). Let $\hat{\mathcal{T}} = \mathcal{T}|_{\pi'}$ and $\hat{\tau} = \tau|_{leaves(\hat{\mathcal{T}})}$. Since π' dominates π, and since π contains *all* ϵ tasks in linearisation τ or their abstracted out nodes (i.e., those in which only ϵ tasks occur in leaves), it is not difficult to see that there must exist a sequence of cuts $\pi'_1 \cdot \ldots \cdot \pi'_k$, such that: *(i)* $\pi'_1 = \pi'$; *(ii)*

$\pi'_k = \pi$; *(iii)* for each $i \in \{2, \ldots, k\}, \pi'_i = (\pi'_{i-1} \setminus \{u\}) \cup children(u, \hat{\mathcal{T}})$ for some $u \in \pi'_{i-1}$; and *(iv)* $k > 1$ (because $|\pi'| < |\pi|$). Therefore, it follows that there is a node $u \in V(\hat{\mathcal{T}})$, with $u \neq (root : \epsilon)$, such that $children(u, \hat{\mathcal{T}}) \subseteq \pi$. Moreover, since $\hat{\mathcal{T}}_{\hat{\tau}}$ is executable in $\mathcal{I}$ relative to Op, we know that $\langle \hat{\mathcal{T}}_{\hat{\tau}}, \mathcal{I} \rangle \models \ell_V(u)$, and therefore, that $\langle \mathcal{T}_\tau|_\pi, \mathcal{I} \rangle \models \ell_V(u)$. Consequently, S cannot hold, and our assumption is incorrect. Hence, hybrid-plan $h_\pi = [\pi, \Phi[\mathcal{T}_\tau, \pi]]$ is indeed a preferred specialisation of hybrid-solution h within $\mathcal{T}_\tau$ for $\mathcal{H}$.

Finally, the fact that hybrid-plan returned by the algorithm is a minimal preferred specialisation of h within $\mathcal{T}_\tau$ follows trivially due to line 10, which removes from π labelled tasks that are either labelled ϵ tasks, or those for which only labelled ϵ tasks occur in leaves. □

It is important to note that the computational complexity of finding a perfect justification of a primitive solution, relative to a given planning problem, is NP-hard (Fink and Yang, 1992). Although we obtain a perfect justification from τ in line 1, it is possible to replace function Get-Perfect-Justification with any other function that finds a "preferred" primitive solution from τ. For example, we could obtain a so-called *well justification* (Fink and Yang, 1992), which can be found in polynomial time on the length of the given primitive solution. Intuitively, a plan is said to be well justified if all its actions are well justified, where an action is well justified if it brings about a literal not brought about by some earlier action, and which is required by the precondition of some action in the plan. While well justified plans, like perfectly justified plans, cannot contain unnecessary actions, they still may, unlike perfectly justified plans, contain unnecessary *groups* of actions. Hence, although no single action can be eliminated from a well justified plan, it may be possible to eliminate several actions together. We refer the reader to (Fink and Yang, 1992) for further details on the notion of well justification.

By replacing function Get-Perfect-Justification with a function that finds some other kind of preferred primitive solution, our algorithm would then return a preferred specialisation that is sound with respect to the corresponding definition of the preferred primitive solution obtained in line 1. It is not difficult to see that Algorithm 5.1, once a perfect justification or preferred primitive plan is obtained, runs in polynomial time on the size of the decomposition tree $\mathcal{T}$.

Lemma 12. *Algorithm 5.1, after completion of line 1, runs in polynomial time on the size of the decomposition tree* $\mathcal{T}$.

Proof. See Appendix A.3. □

Chapter 6

Implementation†

So far, we have provided three formal frameworks. First, we provided a framework for HTN planning in BDI systems, which allows an agent to look-ahead within the context of its existing plan structures in order to make the right choices during decomposition. Since look-ahead does not allow the creation of new plan structures, we provided a framework for first principles planning in BDI systems, which uses the agent's existing domain knowledge so as to construct new hybrid-plan structures not already in the agent's library. Finally, we provided a framework for improving a hybrid-plan obtained, in order to extract its most abstract and non-redundant part.

In this chapter, we implement the three frameworks into a combined system using the JACK Intelligent Agents (Busetta et al., 1999) BDI implementation. The combined system is a prototype that implements the algorithms described in Chapters 3, 4, and 5. Although JACK provides certain features — such as *meta-plans* for dynamic, programmed choice of the most appropriate plan — that are not supported by the HTN language we use in Chapter 3 or the restricted CAN language we use in Chapter 4, we still allow the programmer to exploit the full functionality of JACK when developing real-world applications, and to use planning with only a selected subset of the plan-library. Moreover, by highlighting gaps between the formal frameworks and their implementations — such as differences in how they handle negation — and showing how some of these gaps can be reduced, we give insights into how certain features of JACK that are not supported by the formal frameworks can be specified in formats that are supported by the frameworks. Finally, we shed some light on the practical utility of our implemented systems by, for instance, showing how

†Part of the work presented in this chapter has been previously published in (de Silva and Padgham, 2004, 2005; Sardina et al., 2006).

certain differences between them and the formal semantics, such as replanning at every step as indicated by the Plan derivation rule of Chapter 3, versus planning once and executing a stored solution, are necessary in order to develop practical BDI systems.

JACK is a leading edge, commercial BDI agent development platform, used for industrial software development (Jarvis et al., 2003; Wallis et al., 2002). It has similar core functionality to a collection of BDI systems, originating from the PRS (Georgeff and Ingrand, 1989) system. For the HTN planning system we use JSHOP, which is a Java version of the Lisp based SHOP (Simple Hierarchical Ordered Planner) (Nau et al., 1999) total-order HTN planner, whose successor, SHOP2 (Nau et al., 2003), won one of the top four prizes at the 2002 International Planning Competition.[1] Both JSHOP and SHOP have been integrated into many different types of applications (Muñoz-Avila et al., 2001; Dix et al., 2003; Nau et al., 2005). For first principles planning, we use the C based Metric-FF (Hoffmann, 2003) planning system, which was a top performer in the Numeric Track of third International Planning Competition. Metric-FF is largely based on the FF (Hoffmann and Nebel, 2001) planning system, which was awarded for Outstanding Performance at the second International Planning Competition, and awarded Top Performer in the STRIPS Track of the third International Planning Competition. An overview of the architecture of our combined framework is shown in Figure 6.1.

This chapter is organised as follows. First, in Section 6.1, we show the relationship between the formal languages – CAN and HTN – and their respective implementations – JACK and JSHOP. In particular, we show what gaps exist between the formal languages and their implementations, and how some of these gaps can be overcome. Next, in Section 6.2 and 6.3, we discuss our integrations of respectively JSHOP and Metric-FF into JACK. This includes highlighting the differences between the semantics of previous chapters and our implemented system, and showing why these differences are necessary in order to have a practical BDI system. Finally, in Section 6.4, we discuss our implementation of the Algorithms in Chapter 5, in particular, how we obtain a decomposition tree from JSHOP, and how a preferred specialisation is extracted from the tree.

6.1 Comparing the Formal Languages with their Implementations

In Section 3.1, we provided a mapping from AgentSpeak (Rao, 1996) BDI entities to HTN (Erol et al., 1996) entities. In this section, we will focus on showing the relationship between CAN

[1] http://ipc.icaps-conference.org/

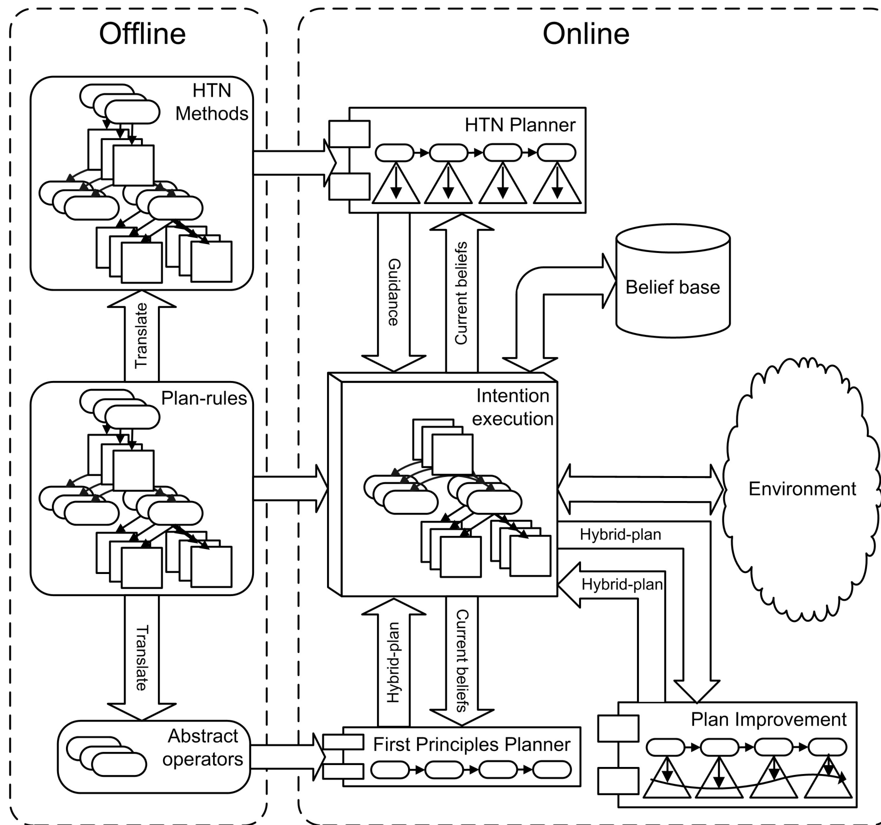


Figure 6.1: The architecture of our combined framework

entities and JACK entities, and between HTN entities and JSHOP entities. An understanding of these relationships is important in order to map from JACK entities to JSHOP entities, and for translating JACK plan-libraries into Metric-FF planning operators.

6.1.1 CAN vs. JACK

Most CAN entities have a corresponding entity within JACK. However, JACK provides a variety of features not supported in CAN, such as *meta-plans* for dynamic, programmed choice of the most appropriate plan, and *maintenance conditions* for ensuring that solutions pursued are aborted if the world changes in unspecified ways. An overview of JACK syntax can be found in Section 2.1.4.

Beliefs and belief operations

The belief base of a CAN agent corresponds to a JACK *beliefset*. A JACK beliefset is a database consisting of multiple relations, each representing a different characteristic of the environment. However, while the belief base of CAN is *closed world*, where any atom not present in the belief base is assumed to be false, JACK additionally allows the use of *open world* relations, where a tuple not present in a relation can either be false or unknown. The belief addition $+b$ and belief removal $-b$ operations of CAN correspond respectively to the *add()* and *remove()* beliefset methods of JACK.

For example, the CAN belief operation $+At(rover, rock3)$ can be encoded in JACK as $at.add($"*Rover*", "*Rock*3"$)$, where at is an instance of the beliefset relation At.

Actions

Like AgentSpeak, JACK does not have a model of actions; actions in JACK correspond to arbitrary Java functions. However, like CAN and AgentSpeak actions, which are non-interruptible, JACK has an @*action* reasoning statement that allows steps requiring lengthy execution (e.g., physically moving a robot to a new location) to be executed until completion before the JACK engine moves on to other intentions. Therefore, we can represent the actions of our revised version of CAN using events and @*action* reasoning statements in JACK. This is done as follows. First, a new JACK event is created along with a new JACK plan for it, and the Java function corresponding to the action is specified as an @*action* reasoning statement within the body of the plan. Second, the precondition of the Java function (if any) is encoded within the context condition of the new JACK plan, and the effects brought about when the function is invoked are specified in the body of the plan, either as JACK beliefset operations, or within *beginEFF* and *endEFF* tags. We have introduced these tags into the system so that the effects of actions can be specified within them. Effects specified within these tags are later extracted during the translation to JSHOP and Metric-FF.

For example, consider the JACK plan on the right in Figure 6.2. The body of this plan contains actions for calibrating and moving. The code for calibrating, which is encapsulated within event *CalibrateEvent*, calls some Java function for calibrating the rover's instruments via GPS. The code for moving, which is encapsulated within event *MoveEvent*, calls a Java function which moves the rover to the destination.

The effects of executing the two pieces of code are specified immediately after them within *beginEFF* and *endEFF* tags. The *@wait_for* statement in the plan for navigating makes the intention wait, for a maximum of *timeout* seconds, until an event is received from the environment confirming the rover's new location.

```
1
2  plan CalibratePlan extends Plan
3  {
4     #handles event CalibrateEvent calibrate;
5
6     context()
7     {
8        true;
9     }
10
11    body()
12    {
13       @action(new CalibrateViaGPS());
14       /** beginEFF
15          (Status Calibrated)
16       endEFF **/
17    }
18 }
```

```
1
2  plan MovePlan extends Plan
3  {
4     #handles event MoveEvent move;
5
6     #uses data At at;
7
8     context()
9     {
10       at.query(move.src);
11    }
12
13    body()
14    {
15       @action(new Move(move.src, move.dst));
16       /** beginEFF
17          (not (At ?nav.src))
18          (At ?nav.dst)
19       endEFF **/
20    }
21 }
```

```
1
2  plan NavigatePlan extends Plan
3  {
4     #handles event Navigate nav;
5
6     #uses data At at;
7
8     #posts event CalibrateEvent calibrate;
9     #posts event MoveEvent move;
10
11    context()
12    {
13       at.query(nav.src);
14    }
15
16    private int timeout = 5;
17
18    body()
19    {
20       @subtask(calibrate.post());
21       @subtask(move.post(nav.src, nav.dst));
22
23       @wait_for(at.query(nav.dst), timeout);
24    }
25 }
```

Figure 6.2: JACK plans for the *Navigate* event-goal and its corresponding actions in the Mars Rover agent of Figure 4.2.

Achievement and test goals

Event-goal programs (!e) of CAN correspond to the posting of JACK events of type *BDIGoalEvent*, via the *@subtask* reasoning statement. However, unlike CAN, JACK provides a variety of event types with different behaviours (e.g., *BDIFactEvent* and *InferenceGoalEvent*), as well as attributes

for customising the behaviour of events. Moreover, there are different options for posting events, via reasoning statements such as *@achieve* and *@insist*: statement *@achieve* posts an event only if a given condition does not hold, and trivially succeeds otherwise; and *@insist* posts an event repeatedly until a given success condition is met.

> For example, a CAN event-goal program $!Move(Rock1, Rock2)$ mentioned within some plan-body can be specified within a JACK plan-body as $@subtask(move.post(\text{“}Rock1\text{”}, \text{“}Rock2\text{”}))$, where *move* is an instance of the event type $Move$.

The test for a condition ($?\phi$) of CAN has a straightforward mapping to a JACK *beliefset query*, modulo a discrepancy in how the two systems handle negation, which we will discuss in Section 6.2.

> A CAN test condition $?On(b, Block1)$ mentioned within some plan-body can be specified within a JACK plan-body as the beliefset query $on.query(\$b, \text{“}Block1\text{”})$, where *on* is an instance of a beliefset relation, method *query* is a Java method which queries the relation, and $\$b$ is a logical variable.

Plan-rules

A CAN plan-rule corresponds to a JACK plan: the context condition of a plan-rule maps to a JACK context condition, modulo the discrepancy mentioned above regarding negation, and the plan-body of a plan-rule maps to a JACK plan-body.

> For example, the CAN context condition $On(block, Table) \wedge Clear(block)$ can be encoded as the JACK context condition $on.query(\$block, \text{“}Table\text{”})\ \&\&\ clear.query(\$block)$. Moreover, the CAN parallel program $!Move(Block1, Table) \parallel !Move(Block2, Table)$ can be written in

JACK as follows:

$@parallel(\ldots)$
$\{$
$@subtask(move.post(\text{“}Block1\text{”}, \text{“}Table\text{”}));$
$@subtask(move.post(\text{“}Block2\text{”}, \text{“}Table\text{”}));$
$\}.$

6.1.2 HTN vs. JSHOP

The HTN language allows partial ordering of tasks, and the ability to specify conditions that must hold before, after or between tasks in a task network. Although JSHOP does not have either of these features, it has features not supported in the HTN language, such as axioms (derived predicates) and the ability to call functions from within methods. Next, we will show how certain features of the two systems can be mapped, and we will identify the entities that do not have a mapping. An overview of JSHOP syntax can be found in Section 2.3.2.

Task networks and methods

The representation of a method in JSHOP is similar to the representation of a method in HTN. Recall from Section 2.3.2 that a HTN method is of the form $(\alpha, [\{(n_1 : t_1), \ldots, (n_m : t_m)\}, \phi])$, where the first component α is a compound task, and the second component is a task network. In the task network, the first component is a set of labelled tasks, and the second component is a constraint formula. Similarly, a JSHOP method is of the form $(: method\ \alpha\ [h]\ C\ T)$, where α is a compound task, C is a conjunction of literals representing the precondition of the method, T, called the *tail*, is a sequence of (primitive and compound) tasks, and h is an optional name for the method.[2] Note that, while the tasks in a tail T must be totally ordered, the labelled tasks in a HTN task network can be partially ordered. Therefore, a HTN task network is more expressive than a JSHOP tail, and a mapping cannot be performed from a partially-ordered HTN task network to a single JSHOP tail.[3] Moreover, note that, although a HTN method does not have a precondition per

[2]Note that in JSHOP, a question mark before a symbol indicates that the symbol is a variable, and an exclamation before a symbol indicates that the symbol is a primitive task/action.

[3]Although it is sometimes (Nau et al., 1998) possible to create a JSHOP tail for every possible (viable) total-ordering of a given partially ordered HTN task network, this will lead to an exponential number of JSHOP tails in the worst case.

se, the precondition C of a JSHOP method can be encoded as state constraints within the constraint formula ϕ of a HTN method.

For example, the following HTN method:

$$(move(b1, b2, b3),\ [\{(1 : pickup(b1)), (2 : stack(b1, b3))\}, \phi]),$$

where

$$\phi = (1 \prec 2) \wedge (Clear(b1), 1) \wedge (On(b1, b2), 1) \wedge (Clear(b3), 1),$$

can be represented in JSHOP with the following method:

```
(: method (move ?b1 ?b2 ?b3)
  ((Clear ?b1)(On ?b1 ?b2)(Clear ?b3))
  ((!pickup ?b1)(!stack ?b1 ?b3))
),
```

where *pickup* and *stack* are primitive tasks and *move* is a compound task.

Initial states, compound tasks, and primitive tasks

The initial state in both HTN and JSHOP is a set of ground atoms. Moreover, primitive tasks and compound tasks in both frameworks have an identical representation. However, there is a subtle difference between a JSHOP operator and HTN operator in that the precondition of a JSHOP operator can only contain atoms, whereas the precondition of a HTN operator can contain literals.[4] This restriction in JSHOP can be overcome by using JSHOP methods as “wrappers” around operators that require literals within their preconditions. In this way, the expressivity of the preconditions of methods can be exploited.

[4]Note that, although, according to some JSHOP related publications, preconditions cannot be specified for operators, the implementation does, in fact, allow such preconditions.

For example, the HTN operator

$$
\begin{aligned}
&[\mathit{operator}\ \mathit{pickup}(b)\\
&\quad(\mathit{pre} : \mathit{On}(b, b2), \mathit{Clear}(b), \neg \mathit{ArmOccupied})\\
&\quad(\mathit{post} : \neg \mathit{On}(b, b2), \mathit{ArmOccupied}, \mathit{Clear}(b2))\\
&],
\end{aligned}
$$

can be represented in JSHOP with the following method and operator:

$$
\begin{aligned}
&(: \mathit{method}\ (\mathit{pickup}\ ?b)\\
&\quad((\mathit{On}\ ?b\ ?b2)(\mathit{Clear}\ ?b)(\neg \mathit{ArmOccupied}))\\
&\quad((!\mathit{pickup}2\ ?b\ ?b2))\\
&),
\end{aligned}
$$

and

$$
\begin{aligned}
&(: \mathit{operator}\ (!\mathit{pickup}2\ ?b\ ?b2)\\
&\quad()\\
&\quad((\mathit{On}\ ?b\ ?b2))\\
&\quad((\mathit{ArmOccupied})(\mathit{Clear}\ ?b2))\\
&).
\end{aligned}
$$

State constraints

It is not, in general, possible to specify in JSHOP that a condition must hold before a task — if a condition needs to hold before the first task in a HTN task network, then this can be specified within the precondition of a JSHOP method. Moreover, it is not possible to specify in JSHOP that a condition must hold after a task or between two tasks.

These restrictions in JSHOP can be overcome, to a certain extent, by using compound tasks and methods. To specify that a condition must hold between two tasks t_1 and t_2 within a JSHOP tail, *(i)* a new compound task is created along with a method that is always relevant, with the precondition of the method containing the condition in question; and *(ii)* a call to the compound task is placed in the tail of the JSHOP method, between the two tasks t_1 and t_2. Other state constraints can be handled in a similar way.

For example, to specify that literal $On(b, Block1)$ should hold immediately before some task t_i in the tail $T = t_1 \cdot \ldots \cdot t_n$ of a JSHOP method, we first create the new method:

(: *method* (*test* ?*block*)
  ((*On* ?*block* *Block*1))
  ()
),

and we then place JSHOP compound task (*test* ?*b*) immediately before task t_i in the tail. Similarly, if literal $On(b, Block1)$ needs to hold between two tasks t_i and t_j (where $i < j$) in T, task (*test* ?*b*) is placed after all tasks t_k in T such that $i \leq k < j$.

The limitation of this approach, however, is that JSHOP does not allow the substitutions applied to variables within the precondition of a method to also be applied (or propagated) to corresponding variables within the tail that calls the method. Therefore, substitutions applied to a precondition within a tail cannot be applied to the remaining steps in the tail.

Consider once again the previous example. Suppose compound task (*test* ?*b*) occurs twice in tail T. Moreover, suppose variable b does not occur anywhere else in the tail except in these two compound tasks. Finally, suppose that some value *Block*2 is assigned to variable *block* when precondition (*On* ?*block* *Block*1) is evaluated.

Now, when precondition (*On* ?*block* *Block*1) is evaluated for the second time to solve the second occurence of compound task (*test* ?*b*) in T, the value assigned to variable *block* may not be *Block*2. This is because, when the precondition is evaluated for the first time, the substitution applied to variable *block* is not applied to variable b occurring in tail T.

This limitation can be overcome by obtaining a binding for b from within the precondition corresponding to tail T in the above examples. In this way, all occurences of variable b in T will be bound before T is solved. For example, JSHOP literal (*Block* ?*b*) could be included in the precondition corresponding to T, which would allow variable b to be bound to *any* block in the domain. Of course, alternative bindings can be tried for b until a binding is found (if one exists) that allows T to be solved.

6.2 Integrating JSHOP into JACK

So far, we have shown the relationship between CAN and JACK, and the relationship between HTN and JSHOP. In this section, we show the mapping from JACK entities to JSHOP entities by incorporating the mappings discussed in the previous sections, and we discuss our implementation of the operational semantics in Chapter 3.

6.2.1 Mapping JACK to JSHOP

Our mapping from JACK to JSHOP includes mapping between certain features of JACK and JSHOP that do not conform exactly to their respective formalisations, but are nonetheless useful in practice. We point out discrepancies between certain basic functionalities of the two systems, and show how these can be addressed.

Belief operations

Since a JACK beliefset is a database, a subset of the attributes of a relation can be chosen to form the *primary key* of the relation. Consequently, adding to a relation a tuple with the same value for the primary key as a tuple that already exists in the relation will cause the old tuple to be replaced with the new tuple. On the other hand, since a JSHOP (as well as a HTN and CAN) belief base is simply a set of ground atoms, all arguments of atoms (relations) are treated as their primary keys. Hence, the removal of an existing belief atom, on the addition of a new one, has to be explicitly handled by the programmer.

For example, consider the initial CAN belief base $\{At(Rover, Rock1), At(Lander, Rock3)\}$. Observe that the corresponding JACK agent has At as a beliefset relation. If in JACK the first attribute in relation At is a primary key of this relation, then performing belief addition $at.add(\text{“}Rover\text{”}, \text{“}Rock3\text{”})$ (where at is an instance of beliefset relation At) will result in the previous location $Rock1$ of the rover being removed from the beliefset. However, performing belief operation $+At(Rover, Rock3)$ in CAN will only result in this new belief atom being added to the set of base beliefs – the previous location of the rover is not automatically removed from the set of base beliefs.

While JSHOP could be modified to allow the programmer to choose a subset of an atom's arguments as its primary key, we have not implemented this for simplicity. However, for an industrial strength system, this should be implemented, and it could be implemented straightforwardly. In the current system, the JACK programmer must select all attributes of relations to be their primary keys, and handle the deletion of existing tuples explictly.

Preferences

Although not directly supported in agent programming languages such as CAN, AgentSpeak, and 3APL, JACK allows plans to have preferences. Preferences can be specified for JACK plans by declaring within a JACK agent the order in which plans should be tested for applicability. Such preferences are useful to specify the order in which plans should be tried, if more than one plan is applicable.

Similarly, it is also possible to specify within a JSHOP domain file the order in which methods should be tried when decomposing compound tasks. It is therefore straightforward for JSHOP to use and respect the preference information available for JACK plans.

Negation

The negation of a beliefset query in JACK is *negation as failure* (Clark, 1978), while negation in JSHOP (as well as CAN and HTN) is standard logical negation. Consequently, while the negation of a beliefset query in JACK succeeds only if no appropriate bindings exist for variables occurring in the query, the negation of an atom in JSHOP succeeds if bindings exist for variables occurring in the atom such that the resulting ground atom is false in the world state.

> For example, suppose we have the JACK or CAN initial state $\{Person(John),$- $Person(Mark), Person(David), Person(James), Married(John), Married(David)\}$. In JACK, the beliefset query $!married.query(\$x)$,[5] where $\$x$ is a logical variable, returns false, since there is a binding for $\$x$ such that $married.query(\$x)$ holds. In JSHOP, on the other hand, literal $\neg Married(?x)$ (i.e., (not ($Married$ $?x$)) in JSHOP syntax) will hold, with a binding of either $James$ or $Mark$ for variable x.

[5]Note that the exclamation symbol ! in JACK means negation, whereas the same symbol is used in CAN in order to distinguish between event-goals and event-goal programs.

To address this difference, we disallow the use of negation on a JACK beliefset query if variables occurring in the beliefset query will not already be bound by the time the query is evaluated. Such bindings can be obtained from elsewhere in the JACK plan (e.g., if the query occurs in a context condition, bindings for all of its variables can be obtained from a previous conjunct in the context condition). In this way, negation as failure and logical negation are evaluated in the same manner.

> For example, we can add beliefset query *person.query*($x) as a conjunct to the beliefset query !*married.query*($x) in the previous example, such that *person.query*($x) occurs before !*married.query*($x) in the formula. This results in the formula *person.query*($x) && !*married.query*($x). This formula will ensure that variable $x is bound by the time !*married.query*($x) is evaluated, because JACK evaluates formulas from left to right.

Calling arbitrary Java functions

JSHOP allows arbitrary Java functions (e.g., *getShortestDistance*) to be called from within the precondition and tail of a method, and for values returned to be compared with variables, constants, or the values returned by other such function calls. In HTN methods, on the other hand, functions cannot be mentioned within a task network – only first order literals are allowed. Like JSHOP, JACK context conditions and plan-bodies also allow arbitrary functions to be called.

JSHOP functions only accept arguments of a single generic type called *JSTerm*, and they only return the same type. Consequently, the main step that is required to be able to use a JACK function from within JSHOP is to modify the JACK function so that it accepts and returns objects of type *JSTerm*. All JACK functions that need to be translated into JSHOP functions are specified in a separate Java Class file, and the translation is done automatically at compile time.

> For example, a JSHOP method can have a precondition that calls a user defined function to obtain the shortest distance between two locations ?x and ?y, and then compares the value returned with the amount of remaining fuel ?f, as follows: (*call* <= (*call getShortestDistance* ?x ?y) ?f), where *call* denotes a procedure call. In JACK the same requirement could be written as *getShortestDistance*($x.*as_int*(), $y.*as_int*()) <= $f.*as_int*(), where $x, $y and $f

are logical variables. If the JACK code for function *getShortestDistance* is encoded as follows:

```
public static double getShortestDistance (double x, double y)
{
  double shortestDist = calculateShortestDist(x, y);
  return shortestDist;
},
```

then the corresponding JSHOP function would look as follows:

```
public static JSTerm getShortestDistance (JSTerm x, JSTerm y)
{
  double x2 = numericValue(x);
  double y2 = numericValue(y);
  JSTerm t = new JSTerm();
  t.makeConstant();
  double shortestDist = calculateShortestDist(x2, y2);
  t.addElement(new Double(shortestDist));
  return t;
}.
```

Axioms

JSHOP supports the use of *axioms*, or *derived predicates* (Thiébaux et al., 2005) within preconditions of methods, which can be used to define new predicates in terms of predicates that already exist in the domain. This allows preconditions to be written that are more elegant and concise than those that do not make use of axioms (Thiébaux et al., 2005), as single predicates can be used in place of their constituent predicates.

For example, the following JSHOP axiom states that a location is within walking distance if *(i)* the weather is good and the location is within two kilometres from

home, or *(ii)* if the location is within one kilometre from home:

```
((WalkingDistance ?x)
  ((Weather Good)(Distance Home ?x ?d)(call <= ?d 2))
  ((Distance Home ?x ?d)(call <= ?d 1))
)
```

When writing a precondition, axiom (*WalkingDistance ?x*) can be used as a predicate.

It is not difficult to see that such axioms can easily be specified as Java functions in JACK, which can then be called from within JSHOP preconditions. Therefore, we do not deal separately with translating parts of JACK context conditions into JSHOP axioms.

Summary of the mapping from JACK to JSHOP

Table 6.1 shows a summary of the mapping from JACK to JSHOP. Note that, for simplicity, we do not automatically translate JACK beliefset queries to JSHOP compound tasks and methods — the programmer needs to encode beliefset queries as context conditions of JACK plans, as we described for JSHOP and state constraints earlier (p. 169). Finally, since a JSHOP context condition is a conjunction of literals, we assume that JACK context conditions do not contain disjunction; no generality is lost here, as any plan with a disjunctive context condition can be split into multiple plans, each containing one of the disjuncts. In the example below, we illustrate some of the mappings highlighted in the Table 6.1.

Figure 6.5 shows a possible encoding of a JACK plan that handles the *ObtainSoilResults* event of the Mars Rover agent in Figure 4.2, along with the corresponding JSHOP method. Note that the actions of picking and dropping a soil sample have not been encapsulated within separate JACK events (as done in Figure 6.2) for simplicity. Observe that the context condition is translated into a JSHOP precondition which uses the JACK event types as predicate symbols. Similarly, the body of the JACK plan is translated into a JSHOP tail which uses JACK event types as compound tasks, and JACK beliefset types for naming primitive actions.

The JACK belief addition and removal associated with the two @*action* statements are translated respectively into JSHOP actions *(!ADD_HaveSoilSample ?dst)* and *(!DEL_HaveSoilSample ?dst)*, along with operators to handle these actions (not shown).

Statements within *beginEFF* and *endEFF* tags are translated to JSHOP actions, similarly to the translation of JACK beliefset operations. For example, the effects *(not (At ?src))* and *(At ?dst)* in Figure 6.2 are translated into the JSHOP actions *(!DEL_At ?src)* and *(!ADD_At ?src)*, respectively, along with operators to handle these actions.

```
1  plan ObtainResultsPlan extends Plan
2  {
3     #handles event ObtainSoilResults obtain;
4
5     #posts event AnalyseSoil analyse;
6
7     #uses data SoilCompartment compartment;
8     #uses data HaveSoilSample haveSample;
9     #uses data At at;
10
11    context()
12    {
13       at.query(obtain.dst) &&
14       compartment.query(``Empty'');
15    }
16
17    body()
18    {
19       @action(new PickSoilSample(obtain.dst));
20       haveSample.add(obtain.dst);
21
22       @subtask(analyse.post(obtain.dst));
23
24       @action(new DropSoilSample(obtain.dst));
25       haveSample.remove(obtain.dst);
26    }
27 }
```

(a) JACK code for ObtainResultsPlan

```
1  (:method (ObtainResultsPlan ?dst)
2
3     (
4        (At ?dst)
5        (SoilCompartment Empty)
6     )
7
8     (
9        (!ADD_HaveSoilSample ?dst)
10       (AnalyseSoil ?dst)
11       (!DEL_HaveSoilSample ?dst)
12    )
13 )
```

(b) Translation of ObtainResultsPlan

Figure 6.5: Mapping from a JACK plan in the Mars Rover agent of Figure 4.2 to a JSHOP method

6.2.2 Implementation Issues

In this section, we discuss our implementation of the operational semantics presented in Chapter 3. The implementation is in the form of a Java package, called *JACKPlan*, which can be imported from JACK when building planning agents. Although the implementation does not precisely realise the operational semantics, it does incorporate the most important concepts from it.

JACK Entities	JSHOP Entities
beliefset	state
beliefset operation (*add()* and *remove()*)	primitive task
effects (within *beginEFF* and *endEFF* tags)	primitive task
action (i.e., event and associated plan)	compound task and associated method
posting of an event (@*subtask*)	compound task
beliefset query	method precondition
context condition	method precondition
parallelism (@*parallel*)	no mapping
plan-body	tail
Java function	Java function accepting and returning *JSTerm*
plan	method
plan preference	method preference
plan-library	set of methods

Table 6.1: Summary of the mapping from JACK to JSHOP

In particular, the implementation allows the programmer to specify from within a JACK plan the points at which JSHOP should be called. This is done with the planHTN function, which takes as an argument a sequence of ground JACK event types,[6] which, as discussed before (p. 164), can also represent actions. On invoking function planHTN, the agent's current set of beliefs is automatically sent to JSHOP, which JSHOP uses as the initial state for HTN planning.

As one example of the use of the JACKPlan package, consider the Mars Rover agent in Figure 3.1 of Chapter 3. Recall that HTN planning is needed for plan-rule R_0 in this figure in order to avoid failure, in certain initial states, due to a wrong choice between plan-rules R_2 and R_3. Figure 6.8(a) shows the implementation of plan-rule R_0 of Figure 3.1, and Figure 6.8(b) shows how HTN planning can be incorporated into the plan-body of Figure 6.8(a). In particular, this is done by replacing the three @*subtask* statements of Figure 6.8(a) with function planHTN, and passing in as an argument the sequence of three ground event types corresponding to the three @*subtask* statements. Observe, further, from Figure 6.8(b), that the *JACKPlan.Planning* class has been imported and inherited; this class provides the HTN planning functionality. Finally, observe that the *#posts event* declarations have been removed because events are no longer posted from the plan-body, although such removal is

[6] A ground JACK event type is a JACK event class name, followed by a constant for each argument in the first Java method for posting an instance of the event. Note that variables can occur in function planHTN provided they will be bound before the function is invoked.

not strictly necessary.

```
1  plan ExplorePlan extends Plan
2  {
3     #handles event ExploreSoilLocation expl;
4
5     #posts event Navigate navigate;
6     #posts event AnalyseSoil analyse;
7     #posts event TransmitData transmit;
8
9     #uses data AvailableBattery battery;
10    #uses data AvailableMemory memory;
11    #uses data At at;
12
13    logical int $bat;
14    logical int $mem;
15
16    context()
17    {
18       at.query(expl.src) &&
19       battery.query($bat) &&
20       memory.query($mem) &&
21       (
22          ($bat.as_int() >= 6 && $mem.as_int() >= 5) ||
23          ($bat.as_int() >= 7 && $mem.as_int() >= 4)
24       );
25    }
26
27    body()
28    {
29       @subtask(navigate.post(expl.src, expl.dst));
30       @subtask(analyse.post(expl.dst));
31       @subtask(transmit.post(expl.dst));
32    }
33 }
```

(a) JACK code for ExplorePlan

```
1  import JACKPlan.Planning;
2
3  plan ExplorePlan extends Planning
4  {
5     #handles event ExploreSoilLocation expl;
6
7     #uses data AvailableBattery battery;
8     #uses data AvailableMemory memory;
9     #uses data At at;
10
11    logical int $bat;
12    logical int $mem;
13
14    context()
15    {
16       at.query(expl.src) &&
17       battery.query($bat) &&
18       memory.query($mem) &&
19       (
20          ($bat.as_int() >= 6 && $mem.as_int() >= 5) ||
21          ($bat.as_int() >= 7 && $mem.as_int() >= 4)
22       );
23    }
24
25    body()
26    {
27       planHTN
28       (
29          ``(Navigate ''+expl.src+`` ''+expl.dst+``)''+
30          ``(AnalyseSoil ''+expl.dst+``)''+
31          ``(TransmitData ''+expl.dst+``)''
32       );
33    }
34 }
```

(b) JACKPlan code for ExplorePlan

Figure 6.8: Incorporating HTN planning into the Mars Rover agent of Figure 3.1

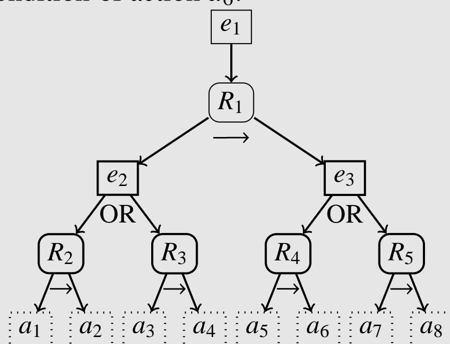
Consistent with the semantics, JSHOP uses the same domain representation as JACK, that is, the plan-library and belief base. To this end, we provide a compilation procedure in JACKPlan that can be used offline in order to build the JSHOP domain representation from the domain representation of JACK. Specifically, JACK entities are converted into JSHOP entities according to the mapping discussed before.

Unlike the semantics, however, the implementation does not re-plan at every step, as indicated by the Plan derivation rule in the semantics. This would be unnecessarily inefficient, since in some cases, HTN planning would need to be performed numerous times, e.g., as many times as the number of steps in the first plan returned. Instead, we have modified JSHOP to return the methods (JACK plans) that should be chosen at the different choice points, as well as the bindings that should be given to variables occurring in preconditions (JACK context conditions). On invoking

function planHTN, the BDI execution engine calls JSHOP once, and then follows step-by-step the decomposition suggested by JSHOP. If by the time execution happens, decomposition information returned by JSHOP has become invalid due to a change in the environment, JACK will detect this change when a step in the returned decomposition is no longer applicable within the BDI execution cycle. At that point, failure will occur in the BDI system, and the planner can be called again to provide an updated plan.

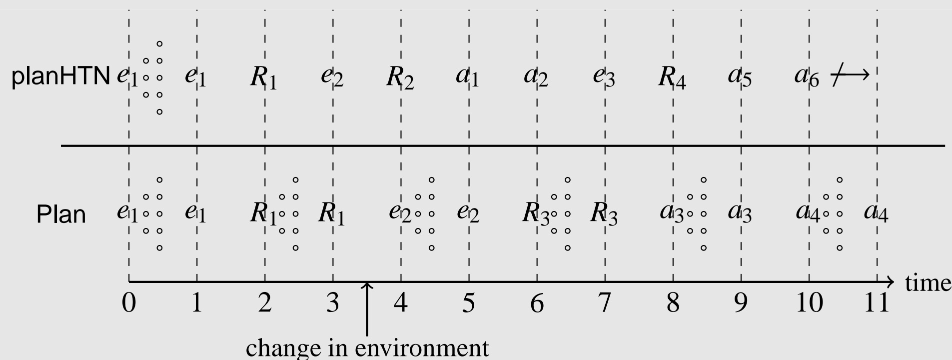
Although storing the plan has its benefits, executing a stored plan also makes the implementation incapable of predicting failure due to a change in the environment, until the failure occurs during execution. On the contrary, the semantics can detect failure early, as replanning is done after executing every action, which ensures that actions are executed only when a solution exists with respect to the current state of the world. However, this drawback in the implementation is offset by the much greater efficiency in what can be expected to be the majority of cases. If early detection is required, some form of plan monitoring (e.g., (Veloso et al., 1998)) could be developed.

Let us illustrate how failure is detected in the semantics and implementation. Consider the plan-library shown below. Observe that events e_2 and e_3 are each handled using two plan-rules. Suppose that, initially, the environment is such that e_1 can be successfully decomposed irrespective of the plan-rules chosen, and that action a_4 brings about the precondition of action a_6.



Next, consider the execution of event e_1 using the Plan construct of the semantics, and the planHTN function of the implementation (for simplicity, we assume that HTN planning takes the same amount of time as executing a step.) The figure below shows how the Plan construct performs full look-ahead (represented by small circles) on event-goal e_1, executes the first step of e_1 (which results in the selection of plan-rule R_1), performs full look-ahead on R_1, executes the first step of R_1 (which

checks whether the context condition of R_1 is met in the initial state), and so on. On the other hand, the planHTN function of the implementation first performs full look-ahead on event e_1, and then executes the resulting decomposition step-by-step.



Now, suppose that a change occurs in the environment between time steps 3 and 4, which results in the precondition of action a_6 being no longer applicable. Observe, then, that the Plan construct realises this change between time steps 4 and 5, and that it pursues an alternative decomposition, which involves selecting R_3 in order to make a_6 applicable once more. The planHTN function, on the other hand, does not realise this change in the environment, until it fails when trying to execute action a_6 at time step 10. However, as hinted by the above figure, if such a change in the environment does not occur, planHTN would most likely complete the successful execution of e_1 before Plan does.

Since the HTN planner is called only at specific points in the BDI program where planning is deemed necessary, we expect the runtime performance of the integrated system consisting of JACK and JSHOP to be sufficiently efficient for many applications. In fact, when we tested the combined system on simple domains such as variations of the Mars Rover domain in Figure 4.2, and a "meeting scheduler" domain where the agent's task is to schedule new meeting requests into a user's diary, JSHOP returned solutions within a matter of a few seconds, even for solving the top level tasks. Specifically, in the latter domain, when a new meeting request arrives along with a set of suitable time slots for the meeting, the agent tries to find an empty slot in the diary that is suitable for the new meeting, and if no such slot is found, the agent tries to clear a slot that is suitable for the new meeting by moving an already scheduled meeting into one of its alternative

suitable slots, failing if no such alternative can be found.[7] This domain consisted of a goal-plan hierarchy of three levels, with two actions ("insert" and "delete"), five event-goals, and seven plan-rules.

We also expect the runtime performance of the combined system to be efficient in many applications because HTN planners have been shown to be practical in many real-world, non-trivial problems that are very difficult for humans to solve (Ghallab et al., 2004, p. 257)(Nau, 2007). The user base of HTN planners includes government laboratories, industries, and universities, with projects such as fighting forest fires, controlling multiple UAVs, and statistical goal recognition – i.e., inferring the goals of other agents (Nau et al., 2005). The reason for the practicality of the HTN approach is that it relies on user-supplied "recipes," which provide control knowledge to the planner, resulting in a substantial reduction in the search space (Ghallab et al., 2004, pp. 229, 259).

6.3 Integrating Metric-FF into JACK

In Chapter 4, we presented algorithms for first principles planning in the CAN BDI agent programming language. Specifically, these were algorithms for summarising effects and preconditions of CAN programs, for creating abstract planning operators from summary information, and finally, for obtaining hybrid-solutions using such operators. In this section, we discuss our implementation of these algorithms in the JACKPlan Java package.

In the implementation, we translate JACK plan-libraries into Metric-FF operators using the summary algorithms of Chapter 4. The subset of the JACK functionalities that we account for in the translation is the subset that is common between JACK and CAN as described in Section 6.1.1, that is, closed world beliefsets, @*subtask* reasoning statements, belief operations, context conditions, sequential plan-bodies, and actions. Restrictions in Section 6.2, which were introduced in the context of JSHOP, also apply to this section, namely, the restrictions on negation and belief operations in JACK.

The functionality of JACK that is not translated into Metric-FF is as follows. First, we do not translate certain features of JACK that were used with JSHOP, namely, preferences on JACK plans and arbitrary function calls within JACK plans. Although such preference information is

[7]Note that the purpose of these experiments was not to motivate the need for HTN planning (as done in Chapter 3), but merely to check how long JSHOP takes to find solutions for real-world (albeit simple) BDI programs.

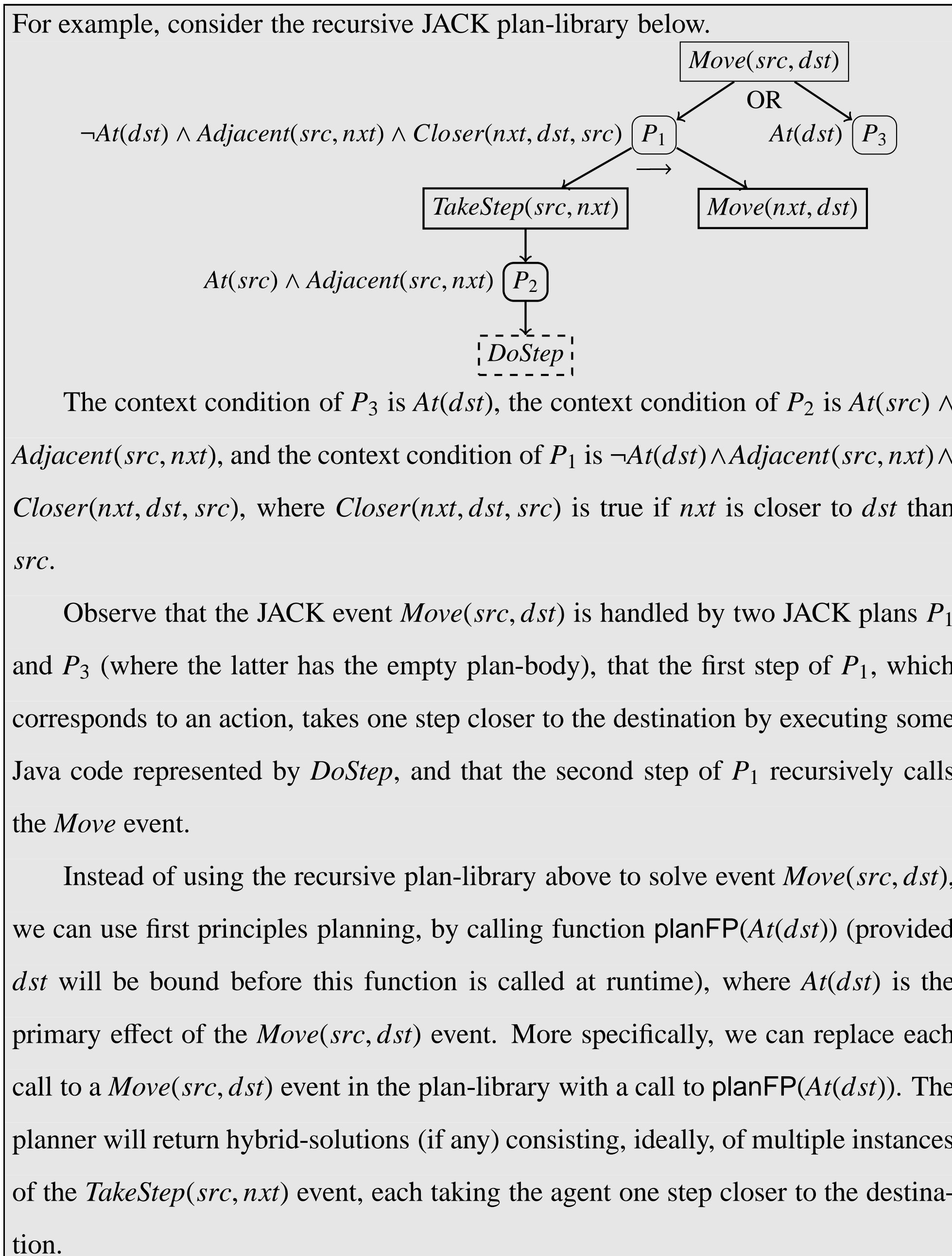
useful for JSHOP, it is not useful for first principles planning, because a first principles planning operator encodes information regarding effects that are brought about irrespective of the JACK plans chosen to achieve a JACK event.

Next, since function symbols cannot occur in *literals* according to the summary algorithms of Chapter 4, nor in literals within Metric-FF operators, calls to arbitrary functions cannot occur in beliefset queries or beliefset operations within JACK plans that need to be summarised. For example, the following JACK belief operation cannot be summarised: *at*.*add*("*Rover*1", *getClosestLander*($*x*)), which specifies that *Rover*1, whose location is represented by variable $*x*, is at the same location as that of the lander which is closest to it. However, such belief operations can sometimes be rewritten in a way that makes their summarisation possible, as illustrated by the following example.

> Consider belief operation *at*.*add*("*Rover*1", *getClosestLander*($*x*)). We could rewrite this as follows. First, we calculate offline the closest lander location for all locations that the rover may travel to. Next, we include this information in the agent's initial belief base. This information could be encoded using a JACK beliefset called *ClosestLander*, with two attributes representing locations, and with the second attribute of each tuple in the beliefset representing the closest lander location from the location represented by the first attribute of the tuple. Finally, the belief addition *at*.*add*("*Rover*1", *getClosestLander*($*x*)) can be rewritten as *at*.*add*("*Rover*1", $*y*), where the value of variable $*y* is obtained from a beliefset query such as *closestLander*.*query*($*x*, $*y*).

Rewriting beliefset queries or beliefset operations in this manner may not always be feasible, since it requires all possible inputs into a function to be mapped to one or more outputs, and for all such mappings to be encoded in the agent's initial belief base. It is important to note, however, that it is still possible to specify arbitrary function calls within JACK plans, provided the values returned by these function calls are not used by beliefset queries or beliefset operations. For example, it is still possible to call a function from within a JACK plan as shown in Figure 6.2.

Finally, while JACK (and CAN) allow parallelism and recursion in plans, we do not allow these features within first principles planning, as discussed in Chapter 4. However, by using first principles planning, we can, to a certain extent, emulate such recursion. This is illustrated in the following example.


For example, consider the recursive JACK plan-library below.

The context condition of P_3 is $At(dst)$, the context condition of P_2 is $At(src) \wedge Adjacent(src, nxt)$, and the context condition of P_1 is $\neg At(dst) \wedge Adjacent(src, nxt) \wedge Closer(nxt, dst, src)$, where $Closer(nxt, dst, src)$ is true if nxt is closer to dst than src.

Observe that the JACK event $Move(src, dst)$ is handled by two JACK plans P_1 and P_3 (where the latter has the empty plan-body), that the first step of P_1, which corresponds to an action, takes one step closer to the destination by executing some Java code represented by *DoStep*, and that the second step of P_1 recursively calls the *Move* event.

Instead of using the recursive plan-library above to solve event $Move(src, dst)$, we can use first principles planning, by calling function planFP($At(dst)$) (provided dst will be bound before this function is called at runtime), where $At(dst)$ is the primary effect of the $Move(src, dst)$ event. More specifically, we can replace each call to a $Move(src, dst)$ event in the plan-library with a call to planFP($At(dst)$). The planner will return hybrid-solutions (if any) consisting, ideally, of multiple instances of the $TakeStep(src, nxt)$ event, each taking the agent one step closer to the destination.

The disadvantage of this approach, however, is that, while the recursive event *Move* will intuitively conform to the user intent property (p. 87) – i.e., it will find a solution that includes only multiple instances of the *TakeStep* event, the first principles planner may find hybrid-plans that involve moving by some other means, i.e., without using the *TakeStep* event.

Although we only translate a limited subset of JACK into first principles planning operators, we note that planning with operators of limited expressivity has been shown to be useful for real-

world problems. In particular, planning from the PRS BDI system using operators of limited expressivity was shown to be useful for controlling the operation of a furnace (Despouys and Ingrand, 1999), and similarly, planning with the overlapping subset between the language of the PRS-CL BDI system and the SIPE-2 planner was shown to be useful for planning military operations (Wilkins et al., 1995).

Consistent with the formalisms presented in Chapter 4, the implementation lets the programmer specify in JACK plans the points at which Metric-FF should be invoked.[8] This is done with the planFP function, which takes as the argument the goal state to be achieved, described in terms of JACK beliefset types. Like the planHTN function, on invoking function planFP, the agent's current set of beliefs is automatically sent to Metric-FF, which it uses as the initial state for planning. Consistent with the summary algorithms of Chapter 4, a compiler supplied with the JACKPlan package can be used offline to translate JACK events into Metric-FF operators, which are then accessed by Metric-FF at runtime.

As one example of the use of the JACKPlan package for first principles planning, consider the Mars Rover agent in Figure 4.2 of Chapter 4. Recall that plan-rule R_6 in this figure is used to perform first principles planning in the event that a failure occurs during exploration.

One possible encoding of plan-rule R_6 using JACK is shown in Figure 6.9. Observe that function planFP takes as an argument the goal state, which, in this case, is the state in which results have been transmitted for the destination.

There are also certain differences between the formalisms of Chapter 4 and the implementation. Unlike the summary algorithms of Chapter 4, the implementation allows the programmer to choose what JACK events need to be summarised. This is particularly useful when certain events cannot be summarised due to an associated JACK plan containing a recursive call, or due to an associated JACK plan using some other a feature not supported in our translation such as parallelism. By letting the programmer choose the JACK events that need to be summarised, we allow the full functionality of JACK to be exploited when developing practical BDI applications, and for first principles planning to be used only with a selected subset of JACK events. Finally, unlike the formalisms, where the domain for first principles planning is built by combining abstract operators

[8]Since Metric-FF is implemented in C, it is called via the Java Native Interface.

```
1  import JACKPlan.Planning;
2
3  plan ExplorePlanViaFP extends Planning
4  {
5     #handles event ExploreSoilLocation expl;
6
7     #uses data ResultsTransmitted transmitted;
8     #uses data At at;
9
10    context()
11    {
12       !at.query(expl.src) && !transmitted.query(expl.dst);
13    }
14
15    body()
16    {
17       planFP(``(ResultsTransmitted ''+expl.dst+``)'');
18    }
19 }
```

Figure 6.9: The JACKPlan specification of plan-rule R_6 in the Mars Rover agent of Figure 4.2

(event-goals) with the primitive actions of the agent, the domain for first principles planning in the implementation is built entirely from JACK events, because primitive actions of the agent are also encoded as events.

At runtime, if Metric-FF fails to find a hybrid-plan, then the step which invoked the planner also fails. If a hybrid-plan is found, then the system validates it. As discussed in Chapter 4, this validation is necessary because abstract operators only encode the must literals of JACK events, and it could happen that when an action within a hybrid-plan is mapped back into an event and executed, other (*may*) literals brought about by the execution of the event create a situation where later JACK events (abstract actions in the plan) are unable to successfully execute. To determine whether all such conflicts (if any) within a hybrid-plan can be avoided, we check to see if there is a complete HTN decomposition of the hybrid-plan, using JSHOP. Since JSHOP needs to also take into account the goal state sent to Metric-FF, we *(i)* encode the goal state into the precondition of a JSHOP method; *(ii)* add the corresponding compound task to the hybrid-plan; and *(iii)* order the compound task to occur after all other compound tasks in the hybrid-plan. Note that we do not perform the polynomial time validation discussed in Chapter 4 because we choose to always improve the hybrid-plan, which requires calling JSHOP anyway. However, the implementation could be easily extended to allow the programmer to choose whether to improve the hybrid-plan, or to simply obtain a correct (but possibly redundant) hybrid-plan using the polynomial-time validation.

If the hybrid-plan returned by Metric-FF is found to be valid, i.e., JSHOP is able to find a complete decomposition of the hybrid-plan, then this hybrid-plan is improved using the algorithms

discussed in Chapter 5. If JSHOP is not able to find a complete decomposition of the hybrid-plan, i.e., a conflict exists in the hybrid-plan that cannot be avoided, a new hybrid-plan is requested from Metric-FF. Since, if the world has not changed, Metric-FF will most likely return the same hybrid-plan that it returned the first time it was called, we randomly rearrange the operators in the Metric-FF domain file, as well as atoms in the initial state, before calling Metric-FF for the second and consequent times. This is done in order to influence Metric-FF to make different choices regarding actions and variable bindings, compared to the choices made the previous time(s) it was called. However, a better approach could be to extend Metric-FF to accept an "exclusion set" of plans, so that it only looks for plans that are not in this set.

We performed experiments with a Mars Rover domain, and with the meeting scheduler domain mentioned in Section 6.2.2, to get insights on whether planning from first principles is practical in real-world applications. The Mars Rover experiment consisted of simple setups such as that denoted by Figure 4.2, and in the meeting scheduler domain we used the planner to reschedule multiple meetings in order to incorporate a new meeting request – i.e., clearing a slot to accommodate a new meeting sometimes involved rescheduling multiple other meetings into their alternative suitable slots. In both domains, in general, the first principles planner returned solutions within a few seconds. However, it was also possible, in the meeting scheduler domain, to create initial and goal states for which the planner took in the order of minutes to return solutions.

Hence, we acknowledge that, while like JSHOP, Metric-FF is also used only at specific points in the BDI program rather than at every step of BDI execution, planning from first principles may still turn out to be too slow for some applications. However, we note that modern classical planners are very fast in general, returning solutions within a few seconds for very large combinatorial problems with hundreds of actions in the plan solutions (e.g., see (Hoffmann and Edelkamp, 2005, p. 553)), and that even for situations in which planning takes some time, it may still be possible to use techniques such as: *(i)* time-outs, to obtain solutions within a given time limit; *(ii)* planning while idle (e.g., in a Mars rover domain the rover could plan while waiting for a command from earth); or *(iii)* planning while acting, in a way that will not interfere with the planning process. For example, a rover could, while moving from one location to another, plan for a goal related to the analysis of previously acquired soil samples. Finally, we note that the ability to plan from first principles has a practical benefit as shown in Chapter 4, even if it does take some time.

6.4 Improving and Executing Hybrid-Solutions

After obtaining a valid hybrid-plan, i.e., a hybrid-solution, we focus on improving it, by extracting its most abstract and non-redundant part. To this end, we mainly follow Algorithm Find-Preferred-Specialisation (Algorithm 5.1) of Chapter 5. This involves obtaining a decomposition tree from the successful decomposition found by JSHOP when validating the hybrid-plan returned by Metric-FF, and abstracting out JACK events in the decomposition tree, starting from the primitive actions, i.e., leaf-level JACK events.

Like Algorithm Find-Preferred-Specialisation, the implementation finds a preferred specialisation of a complete decomposition tree, which is a decomposition tree combined with one of its primitive solutions for the goal state at hand. To build such a tree, we modify the JSHOP domain file to return, in addition to a primitive solution for the input task network and goal state, also the choices that led to the solution.

In particular, this is information regarding what ground compound and primitive tasks were selected to reduce other ground compound tasks, and what ground preconditions were encountered during the decomposition, which, as mentioned before, can be considered restricted HTN constraint formulas.[9] With this information we straightforwardly build a decomposition tree, which is then used along with the associated primitive solution, initial state, and goal state to find a preferred specialisation in the manner described in Algorithm Find-Preferred-Specialisation.

In order to obtain a non-redundant primitive solution from the leaf-level primitive solution associated with the decomposition tree, we use the Linear-Greedy-Justification algorithm of Fink et. al (Fink and Yang, 1992).[10] This is different to what we do in Algorithm 5.1, which obtains a perfect justification (Fink and Yang, 1992) from the primitive solution associated with the decomposition tree. The reason for this difference is that, although the notion of a perfect justification has an intuitive definition that is useful for formalisations (i.e., that a primitive solution is perfectly justified if it does not have a subsequence that is still a primitive solution), it is not feasible to find perfect justifications in practice, as it is NP-hard to compute (Fink and Yang, 1992). On the other hand, although the Linear-Greedy-Justification algorithm does not have an intuitive definition, it

[9] Note that this information is slightly different to the information required by the planHTN function – the latter requires information about methods (JACK plans) chosen at the different choice points, and substitutions applied to variables.

[10] Note that step nine in function Linear-Greedy-Justification of (Fink and Yang, 1992) should recursively call function Linear-Greedy-Justification, instead of calling function Linear_Well_Justification.

is able to find, in polynomial-time, primitive solutions that are "almost" perfect justifications (Fink and Yang, 1992).

It is important to note that non-redundancy is only one notion of what constitutes a "good" primitive solution. One may want to define other notions of "good" primitive solutions, and algorithms for obtaining such solutions, given any primitive solution as input. Such algorithms can be easily incorporated into the implementation by replacing function Linear-Greedy-Justification with the new function. Then, a preferred specialisation obtained from a decomposition tree will be with respect to the new algorithm for finding a "good" primitive solution.

Once a preferred specialisation is obtained for the hybrid-solution found by Metric-FF, the specialisation is then executed. Since the specialisation is still a totally-ordered hybrid-plan, execution simply involves posting, via JACK *@subtask* statements, the events contained in the specialisation, in the same order in which they are specified in the specialisation, while also taking into account binding information for variables of events. Since such an execution does not guarantee that the non-redundant primitive solution will eventually be reached, it is not difficult to extend the implementation to execute the hybrid-plan by following the choices and bindings specified in an associated decomposition tree. It is worth noting that, while a hybrid-plan may be valid with respect to the initial state, goal state, and expected (must or may) effects of its events, it could still be the case that while the hybrid-plan is being executed, the world changes in a way which makes the plan no longer valid, i.e., makes one or more context conditions of associated plan-rules false when they would otherwise have been true. In such a situation, the BDI engine will detect this "discrepancy" between expected beliefs (initial state and expected effects of events) and actual beliefs (state of the world) as a failure, in the manner discussed in Section 6.2.2, and continue execution to recover from the failure by trying alternative plan-rules.

Chapter 7

Discussion and Conclusion

BDI systems are extremely flexible and responsive to the environment, and thereby able to work effectively in complex and dynamic environments. An important aspect of such systems is that they execute as they reason. In particular, they execute by context dependent expansion of subgoals, acting as they go. However, BDI systems do not incorporate a generic mechanism to do any kind of planning. In this thesis, we have incorporated two types of planning techniques into BDI agents, namely, HTN planning and first principles planning.

Incorporating HTN planning into CAN

In Chapter 3, we incorporated look-ahead deliberation in the style of HTN planning into the BDI model. We first compared the syntax and semantics of the AgentSpeak BDI agent-oriented programming language with that of HTN planning, and we then incorporated HTN planning into the CAN BDI agent-oriented programming language via the $\mathsf{Plan}(P)$ construct. This construct was added in a precise and formal manner, and in a way that allows an agent to perform look-ahead at programmer specified points in the plan-library. We showed that the combined architecture is more expressive than HTN planning alone, and that the architecture allows agents to detect and avoid executions that are bound to lead to certain types of failures, such as those that result from negative interactions between plan-rules.

An interesting avenue for future work would be to investigate a resource-bounded account of our HTN planning module. For example, one could investigate looking ahead up to a given number of decompositions, in order to cater for domains in which there is limited time for plan-

ning. We have already begun work in this direction by extending the planning module $\mathsf{Plan}(P)$ to take into account an additional parameter corresponding to the maximum number of steps (e.g., decompositions) up to which look-ahead should be performed. Some of the theoretical and empirical results from this approach can be found in (de Silva and Dekker, 2007; Dekker and de Silva, 2006).

First Principles Planning

While look-ahead is useful for reasoning about the consequences of choosing one expansion of an event-goal over another, look-ahead cannot be used to create new plan-rules not already a part of the agent's plan-library. Creating new plan-rules on demand is desirable, for instance, when all plan-rules associated with an event-goal have failed. To this end, in Chapter 4, we incorporated first principles planning into the BDI architecture. Unlike previous work on adding first principles planning into BDI systems, which focuses on producing low-level plans, losing much of the domain knowledge inherent in BDI systems, we presented a novel technique where first principles planning is used to create *hybrid-plans*, namely, those that can contain, in addition to primitive actions, also event-goals. Since event-goals capture the agent's procedural domain knowledge, our approach allows such knowledge to be reused and respected when formulating solutions.

It is worth noting that, while we have provided a formal framework for first principles planning in BDI systems, we have not provided an operational semantics that defines the behaviour of a BDI system with an in-built first principles planner. To this end, one would need to add module $\mathsf{Plan}(\phi)$ into a language such as CAN, where ϕ is a goal state to achieve, and provide derivation rules for this module that reuse and respect the procedural domain knowledge in the plan-library.

Summarisation algorithms

To use event-goals for first principles planning, we provided mechanisms for translating them into abstract planning operators. To this end, we first defined precisely what information we need to extract from event-goals, in particular, the notions of a precondition and postcondition of an event-goal, and we then provided algorithms and data structures for obtaining this information. Our algorithms are based on the summary algorithms of (Clement et al., 2007), which are used to calculate offline the summary information of HTN-like hierarchical structures (task-networks)

belonging to multiple agents, so that this information can be used at runtime to coordinate those agents. There are, however, important differences between their summary algorithms and ours. First, the precondition of an event-goal in our work is a standard classical precondition (with disjunction), whereas in their work, a precondition of the corresponding entity – a compound task – is essentially two sets of literals: those that must hold at the start of any successful execution of the task, and those that must hold at the start of one or more successful executions of the task. Second, as discussed in Chapter 4, our algorithms, unlike theirs, allow for the specification of a wider range of BDI plan-libraries, as well as variables in literals, event-goals and actions. However, our summary algorithms do not allow parallelism in plan-bodies – plan-bodies can only contain steps specified in a sequence – whereas their algorithms do allow such parallelism.

Our summary algorithms are also related to the summary algorithms of Thangarajah et al. (Thangarajah et al., 2003a,b). In their work, the summary information obtained from event-goals is used for detecting and avoiding potential interference between event-goals executing simultaneously, and for facilitating the merging at runtime of plan-bodies belonging to multiple event-goals. Like our algorithms, calculating the summary information of an event-goal in their work involves merging the summary information belonging to all associated plan-rules, and classifying literals in the combined summary information as *definite* or *potential*, similar to how we classify literals as *must* or *mentioned*. However, there are also important differences between the two approaches. Most importantly, the summary postcondition of a plan-rule in their work contains every intermediate literal that will (definitely or possibly) be brought about during the execution of the plan-rule. In contrast, the must literals of a plan-rule in our work contains only literals that will be brought about at the *end* of the plan-rule's execution. Moreover, like the algorithms of Clement and Durfee, the work of Thangarajah et al. does not allow variables in literals, event-goals and operators, whereas we do allow variables in those entities. However, while our work only allows plan-bodies to contain steps specified in a sequence, the work of Thangarajah et al. allows parallelism within plan-bodies.

In addition to not allowing parallelism, our summary algorithms, like those of Clement and Durfee and Thangarajah et al., do not allow recursion within plan-bodies (i.e., calling from within a plan-body the event-goal that the plan-body handles, or an event-goal that is an ancestor of the one that the plan-body handles). Parallelism could be incorporated into our algorithms by borrowing ideas from the algorithms of Clement and Durfee, and recursion could be incorporated

into our algorithms by following (Fritz et al., 2008), who take a limited form of recursion into account when translating from a subset of the language of ConGolog into planning operators.

Finding hybrid-plans

After providing algorithms and data structures for summarising plan-libraries, we explored the soundness and completeness properties of the algorithms, and we finally provided mechanisms for obtaining correct hybrid-plans using information collected via summarisation. One shortcoming of this approach is the following. If a hybrid-plan is found to be potentially incorrect, standard HTN decomposition is used to verify whether the hybrid-plan is definitely incorrect, in which case a new hybrid-plan is obtained. However, since such verification requires determining whether there is a complete decomposition of the hybrid-plan that achieves some goal state, and HTN planners do not, unlike first principles planners, make use of any heuristics to guide the process of decomposition toward the goal state, our verification step is not always efficient. One could improve it by extending the UMCP HTN planning algorithm of (Erol et al., 1996) with heuristics, so that the process of decomposition is biased toward methods that take the search closer to the goal state, as done in (Lotem et al., 1999). Alternatively, one could investigate extending UMCP to perform HTN decomposition by working backward from the goal state, as opposed to the default HTN planning process of working forward from the initial state.

Abstraction and redundancy

While obtaining correct hybrid-plans is essential, it is also important to obtain desirable hybrid-plans. To this end, in Chapter 5, we recognised an intrinsic tension between striving for hybrid-plans and, at the same time, ensuring that unnecessary actions, unrelated to the specific goal state to be reached, are avoided. To explore this tension, we first characterised the set of "ideal" hybrid-plans, which are non-redundant while maximally-abstract. Next, we developed a more limited but feasible account of "preferred" hybrid-plans in which a hybrid-plan is "specialised" into a new hybrid-plan that is non-redundant while preserving abstraction as much as possible. We also presented an intermediate notion which is conceptually closer to the feasible notion of a preferred hybrid-plan. We analysed the properties of the different notions presented, and showed, for instance, that an ideal hybrid-plan always exists provided the planning problem can be solved, and that an ideal hybrid-plan is also one that is preferred. Finally, we described algorithms for

computing preferred hybrid-plans.

The work of (Kambhampati et al., 1998) is similar to our work on hybrid-planning and is indeed motivated by the desire to combine HTN and first principles planning. Our work is different in that we construct abstract operators from a BDI plan-library, and then execute the resulting hybrid-plan within our framework, whereas in their work, event-goals are decomposed during the process of first principles planning. There are also differences in the details of the approach. Most importantly, they require the programmer to provide effects, whereas we compute these automatically, and they do not address the issue of the balance between abstraction and redundancy, which we explore in depth.

Perhaps the most interesting direction for future work is the investigation of a more general framework for finding good (e.g., "ideal" or "preferred") hybrid-solutions. In our current framework, we consider redundancy as one of the underlying factors that determine whether a hybrid-solution is good. While removing redundant steps is reasonable in some domains (e.g., the Mars Rover domain of Figure 4.4, p. 106), it may be inappropriate in other domains, in particular, because HTN structures sometimes encode strong preferences from the user. For example, consider a hybrid-solution containing the following sequence of tasks (Kambhampati et al., 1998): get in the bus, buy a ticket, get out of the bus. Although it may be possible to travel by bus without buying a ticket, if we remove this task when it is redundant, we may go against a strong preference of the user which requires that task to be performed after getting into the bus and before getting out of it. To cater for strong preferences, one could use ideas from (Sohrabi et al., 2009) and generalise our framework with a more flexible account in which, for instance, all HTN preferences are assumed to be strong, and a redundant task is only removed if the user has separately specified that the task is not strongly preferred. For example, while the task of buying a bus ticket may be redundant, it is not removed from a hybrid-solution unless the user has specified that the task is not strongly preferred. Such specifications could be encoded as hard constraints or soft preferences, and included either within an extended version of HTN methods, or as global constraints outside of methods as done in (Sohrabi et al., 2009).

Implementation

In Chapter 6, we discussed the implementation of the algorithms presented in previous chapters, using the JACK BDI development platform, JSHOP total-order HTN planner, and the Metric-FF

first principles planner. We also gave insights into the practical utility of the formal frameworks, by, for instance, highlighting the gaps between the frameworks and their implementations, and showing how some of these gaps can be reduced. Moreover, we showed how certain differences in the semantics and implementation, such as replanning at every step as indicated by the Plan derivation rule of Chapter 3, versus planning once and executing a stored solution, are necessary in order to develop practical BDI systems.

As discussed in Chapter 6, extensions to the implementation are required before it can be used as an industrial strength system, such as extensions to JSHOP to allow the selection of a subset of an atom's arguments as its primary key, similarly to how primary keys can be specified in a JACK beliefset. Another avenue to consider would be to actually use the proposed planning facilities in applications, and to evaluate and validate the effectiveness and applicability of the proposed facilities in practice. For example, the types of domains in which planning from first principles is worthwhile could be explored, or we could investigate the feasibility of planning from first principles as a part of the standard BDI execution cycle, e.g., whenever an applicable plan is not available, instead of letting the event-goal fail. Intuitively, this approach is likely to be more robust in some applications since it tries to prevent the failure of event-goals at every opportunity, rather than only at user specified points in the BDI hierarchy. However, this approach is also likely to be very computationally expensive, as the planner may fail to find a solution each time it is called from one level higher in the BDI hierarchy. The work presented in this thesis provides a firm foundation for further work on planning in BDI systems, both theoretical and practical.

Appendix A

Lemmas and Proofs

A.1 Proofs for Chapter 3

Proof of Lemma 1 (p. 74). We prove this by induction on the length n of the plan-type derivation. For the base case, let us take $n = 0$. Then, $P = nil$, $\mathcal{B}_f = \mathcal{B}$, and $\mathcal{A}_f = \mathcal{A}$. By using derivation rule Plan_t, we obtain $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$. Next, assume that the theorem holds for $n \leq k$. Finally, suppose $n = k + 1$. Then, there exists $C' = \langle \mathcal{B}', \mathcal{A}', P' \rangle$ such that *(a)* $\langle \mathcal{B}, \mathcal{A}, P \rangle \xrightarrow{\mathsf{plan}} C'$, and *(b)* $C' \xrightarrow{\mathsf{plan}_k} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$. Using *(a)* and *(b)*, we can use derivation rule Plan to obtain $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\mathsf{bdi}} \langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle$. Moreover, from *(b)* and the induction hypothesis, $\langle \mathcal{B}', \mathcal{A}', \mathsf{Plan}(P') \rangle \xrightarrow{\mathsf{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$ holds. Therefore, $\langle \mathcal{B}, \mathcal{A}, \mathsf{Plan}(P) \rangle \xrightarrow{\mathsf{bdi}_*} \langle \mathcal{B}_f, \mathcal{A}_f, nil \rangle$ follows. □

A.2 Lemmas and Proofs for Chapter 4

Proof of Lemma 4 (p. 109). First, suppose $P = ?\phi$. Then, $post(P) = \emptyset$ (p. 96), and there is exactly one tuple $\langle P, \epsilon, \emptyset, \emptyset \rangle \in \Delta$. Since no literal is added to the belief base upon the execution of P, and $\emptyset$ is a valid set of must literals (Definition 12), the theorem holds.

Next, suppose $P = +b$. Let $b\theta$ be any ground instance of b. Then, there is exactly one tuple $\langle P, \epsilon, \{b\}, \{b\} \rangle \in \Delta$, and for all belief bases $\mathcal{B}$ and action sequences $\mathcal{A}$, the following two conditions hold: $\langle \mathcal{B}, \mathcal{A}, +b\theta \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}' = \mathcal{B} \cup \{b\theta\}, \mathcal{A}, nil \rangle$, i.e., there is a successful HTN execution of $+b\theta$; and $\mathcal{B}' \models b\theta$. Therefore, b is a must literal of P. Similarly, since the addition of belief atom $b\theta$ to belief base $\mathcal{B}$ is the only modification that can happen to $\mathcal{B}$ on the successful HTN execution of

$+b\theta$, it follows that set $\{b\}$ captures P (Definition 14), and the theorem holds. The case $P = -b$ can be proved analogously.

Finally, suppose $P = act$. Let $\hat{\Phi}^+ = \Phi^+\theta$ and let $\hat{\Phi}^- = \Phi^-\theta$, where $act' : \psi \leftarrow \Phi^+; \Phi^- \in \Lambda$ and $act = act'\theta$. Finally, let the set of literals $L_{act} = \hat{\Phi}^+ \cup \{\neg b \mid b \in \hat{\Phi}^-\}$. Observe, then, that there exists exactly one tuple $\langle P, \epsilon, L_{act}, L_{act}\rangle \in \Delta$. Let l be any literal in L_{act}. We will now prove that l is a must literal of act. Since, from the definition of an action-rule (Section 3.2.3), free variables in l will also be free in act, the first condition in Definition 12 is satisfied for l and P. Next, let $act\theta'$ be any ground instance act. Suppose there is a successful HTN execution of $act\theta'$, i.e., that $\langle \mathcal{B}, \mathcal{A}, act\theta'\rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}' = (\mathcal{B} \setminus \hat{\Phi}^-\theta') \cup \hat{\Phi}^+\theta', \mathcal{A} \cdot act\theta', nil\rangle$ holds. Then, $\mathcal{B}' \models l\theta'$ also holds. Hence, the second condition of Definition 12 is satisfied for l and P, and l is a must literal of P. The fact that L_{act} captures P follows trivially from the fact that for any ground literal l' such that $\mathcal{B}' \models l'$ and $\mathcal{B} \not\models l'$ hold, it is also the case that l' is a ground instance of some literal in L_{act}. $\square$

Proof of Lemma 5 (p. 109). Consider line 1 of procedure Summarise-Plan-Body. From the assumption of the theorem (condition 2.), it is clear that on the completion of this line, for each event-goal program $!e$ mentioned in P, there is exactly one tuple $\langle !e, \phi_e, L_e^{mt}, L_e^{mn}\rangle \in \Delta$ such that the tuple is the summary information of $!e$, and L_e^{mn} captures $!e$. Then, from the assumption of the theorem (condition 1.), we can conclude that on the completion of line 1, for each atomic program P^a mentioned in P, there is exactly one tuple $\langle P^a, \phi_{P^a}, L_{P^a}^{mt}, L_{P^a}^{mn}\rangle \in \Delta$ such that the tuple is the summary information of P^a, and $L_{P^a}^{mn}$ captures P^a.

To prove that $\langle P, \epsilon, L^{mt}, L^{mn}\rangle$ is the summary information of P, we will first prove that each literal in L^{mt} (where L^{mt} is a set of must literals of P) is a must literal of P. Let program $P = P_1; P_2; \ldots; P_n$, where each P_i is an atomic program. Observe from line 3 of procedure Summarise-Plan-Body that the only literals included in the set L_P^{mt} of must literals of P are the literals that are must literals l of atomic programs P_i mentioned in P, where l is not may-undone (i.e., $\neg$May-Undone($l, P_{i+1}; \ldots; P_n$, Δ)) in $P_{i+1}; \ldots; P_n$. Let l be such a literal. Next, we prove that l is a must literal of P.

Let us assume the contrary, i.e., that l is not a must literal of P. Then, informally, it must be the case that the complement of l is true at the end of a successful HTN execution of program $P_{i+1}; \ldots; P_n$ – in particular, the complement of l must be true at the end of a successful HTN execution of some atomic program mentioned in $P_{i+1}; \ldots; P_n$. Formally, observe from Definition

12 (Must Literal) that there exists an atomic program $\hat{P}$ mentioned in $P_{i+1}; \ldots; P_n$, a ground instance $\hat{P}^g$ of $\hat{P}$, and some successful HTN execution $\langle \mathcal{B}_1, \mathcal{A}_1, \hat{P}^g \rangle \cdot \ldots \cdot \langle \mathcal{B}_m, \mathcal{A}_m, nil \rangle$, such that *(a)* $\mathcal{B}_1 \not\models l'$; *(b)* $\mathcal{B}_m \models l'$; and *(c)* $\bar{l}\theta = l'$, for some ground literal l' and set of substitutions θ. Let $\langle \hat{P}, \hat{\phi}, L^{mt}_{\hat{P}}, L^{mn}_{\hat{P}} \rangle \in \Delta$. We know from before that $L^{mn}_{\hat{P}}$ captures $\hat{P}$. Then, from Definition 14 (Capturing a Program), it is also the case that there is a literal $\hat{l} \in L^{mn}_{\hat{P}}$ such that $l' = \hat{l}\hat{\theta}$, for some set of substitutions $\hat{\theta}$. Then, using *(c)* above, we can conclude that $\bar{l}\theta = \hat{l}\hat{\theta}$. However, since $\neg$May-Undone$(l, P_{i+1}; \ldots; P_n, \Delta)$ holds according to the algorithm, observe from the definition of May-Undone (Section 4.2.3) that $\bar{l}\theta = \hat{l}\hat{\theta}$ cannot hold. This contradicts our assumption, and therefore, literal l is indeed a must literal of P.

Next, we will prove that the set of mentioned literals L^{mn} of program P captures P. To this end, all we need to show is that any must or mentioned literal of an atomic program occurring in P that is not included in L^{mn}_P (line 4) is not needed for L^{mn}_P to capture P.

Let P_i be an atomic program mentioned in P, with $\langle P_i, \phi_i, L^{mt}_{P_i}, L^{mn}_{P_i} \rangle \in \Delta$, such that a must or mentioned literal l of P_i is not added to the set constructed in line 4 of the algorithm, that is, Must-Undone$(l, P_{i+1}; \ldots; P_n, \Delta)$ holds. Then, according to the definition of Must-Undone (Section 4.2.3), it is the case that $\bar{l} = \hat{l}$ holds, where $\hat{l} \in L^{mt}_{\hat{P}}$ is a must literal of an atomic program $\hat{P}$ mentioned in $P_{i+1}; \ldots; P_n$, with $\langle \hat{P}, \phi_{\hat{P}}, L^{mt}_{\hat{P}}, L^{mn}_{\hat{P}} \rangle \in \Delta$. Next, let $P\theta$ be any ground instance of P. Suppose a successful HTN execution $\langle \mathcal{B}, \mathcal{A}, P_i\theta \rangle \cdot \ldots \cdot \langle \mathcal{B}_j, \mathcal{A}_j, nil \rangle$ of $P_i\theta$ exists, such that $\mathcal{B}_j \models l\theta$ holds. Then, since $\hat{l}$ is a must literal of $\hat{P}$, it is the case that $\mathcal{B}'_k \models \hat{l}\theta$ holds for any successful HTN execution $\langle \mathcal{B}', \mathcal{A}', \hat{P}\theta \rangle \cdot \ldots \cdot \langle \mathcal{B}'_k, \mathcal{A}'_k, nil \rangle$ of $\hat{P}\theta$. However, since $\bar{l}\theta = \hat{l}\theta$, literal $l\theta$ is guaranteed to be removed from the belief base by $\hat{P}\theta$ during any successful HTN execution of $P\theta$. Therefore, a set of literals that captures P does not need to include literal l of P_i. (Note, however, that it is still possible that the same literal l from the set of must or mentioned literals of some other atomic program P_j $(j \neq i)$ occurring in P is included in the set of literals that captures P.) □

Proof of Lemma 6 (p. 110). First, we will prove that L^{mt} is a set of must literals of e. Let $R = \{e'\theta : \psi\theta \leftarrow P\theta \mid e' : \psi \leftarrow P \in \Pi, e = e'\theta,\ \theta \text{ is a renaming substitution for } e' : \psi \leftarrow P\}$. Let literal $l \in L^{mt}$, and let $!e\theta$ be any ground instance of $!e$. Suppose a successful HTN execution of $!e\theta$ exists. Then, from the *Sel* transition rule (p. 63), there is a plan-rule $e : \psi \leftarrow P \in R$ such that $\langle \mathcal{B}, \mathcal{A}, !e\theta \rangle \xrightarrow{\text{plan}} \langle \mathcal{B}, \mathcal{A}, (\!|\{\psi\theta : P\theta, \ldots\}|\!) \rangle \xrightarrow{\text{plan}} \langle \mathcal{B}, \mathcal{A}, P\theta\theta' \rhd (\!|\Delta|\!) \rangle \xrightarrow{\text{plan}_*} \langle \mathcal{B}', \mathcal{A}', nil \rangle$ holds (up to variable renaming of plan-library Π). Moreover, from line 11 of procedure Summarise-Event,

all variables occurring in l also occur in e. Therefore, since $\langle \mathcal{B}, \mathcal{A}, P\theta\theta' \rhd (\!|\Delta|\!) \rangle \xrightarrow{\mathsf{plan}_*} \langle \mathcal{B}', \mathcal{A}', nil \rangle$ holds, and since l is a must literal of P from the assumption of the theorem, it must be the case that $\mathcal{B}' \models l\theta$ holds, and that l is a must literal of e. The fact that L^{mn} captures e follows trivially from the fact that L^{mn} includes all literals (up to variable renaming) in the sets of mentioned literals of plan-bodies occurring in R, and each such set captures the corresponding plan-body according the assumption of the theorem. □

Proof of Lemma 7 (p. 110). Let $R = \{e'\theta : \psi\theta \leftarrow P\theta \mid e' : \psi \leftarrow P \in \Pi, e = e'\theta,\ \theta$ is a renaming substitution for $e' : \psi \leftarrow P\}$. Then, $\phi' = \psi_1 \vee \ldots \vee \psi_n$ according to procedure Summarise-Event, where $R = \{e_1 : \psi_1 \leftarrow P_1, \ldots, e_n : \psi_n \leftarrow P_n\}$, and ϕ' is some variable renaming of ϕ. We will now show that ϕ' is the precondition of e. Let $!e\theta$ be any ground instance of $!e$, and let $\mathcal{B}$ be any belief base. Observe from Definition 11 (Precondition) that there are two cases to consider.

[Case $\Rightarrow$]: Suppose $\mathcal{B} \models \phi'\theta$ holds. Then, $\mathcal{B} \models \psi\theta\theta'$, where ψ is some disjunct of ϕ'. Let $e : \psi \leftarrow P \in R$ be the plan-rule corresponding to ψ. Since $\mathcal{B} \models \psi\theta\theta'$, we know from rules *Event* and *Sel* (p. 63) that the following transitions are possible: $\langle \mathcal{B}, \mathcal{A}, !e\theta \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, (\!|\{\psi\theta : P\theta, \ldots\}|\!) \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, P\theta\theta' \rhd (\!|\Delta|\!) \rangle$ (up to variable renaming of plan-library Π). Moreover, from our assumption in Section 4.2.1 (p. 92) which states that $e : \psi \leftarrow P$ is *safe*, there is a successful HTN execution $C_1 \cdot \ldots \cdot C_n$ of $P\theta\theta'$ such that $C_1|_{\mathcal{B}} = \mathcal{B}$. Therefore, it follows that there is also a successful HTN execution $C'_1 \cdot \ldots \cdot C'_m$ of $!e\theta$ such that $C'_1|_{\mathcal{B}} = \mathcal{B}$.

[Case $\Leftarrow$]: Suppose there is a successful HTN execution $C_1 \cdot \ldots \cdot C_m$ of $!e\theta$ such that $C_1|_{\mathcal{B}} = \mathcal{B}$. Then, according to the *Event* and *Sel* rules, there must exist a plan-rule $e : \psi \leftarrow P \in R$, with $\mathcal{B} \models \psi\theta\theta'$, such that the following transitions are possible: $\langle \mathcal{B}, \mathcal{A}, !e\theta \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, (\!|\{\psi\theta : P\theta, \ldots\}|\!) \rangle \xrightarrow{\mathsf{plan}} \langle \mathcal{B}, \mathcal{A}, P\theta\theta' \rhd (\!|\Delta|\!) \rangle$ (up to variable renaming of Π). Therefore, $\mathcal{B} \models \phi'\theta$ holds. □

Proof of Lemma 8 (p. 110). Termination of the loops in lines 3 and 5 follows trivially from the fact that there is a finite number of atomic programs mentioned in the plan-body P, and from the fact that the summary postcondition of any atomic program is a finite set of elements. □

Proof of Lemma 9 (p. 110). Termination of the loop in line 3 follows trivially from the fact that

there is a finite number of plan-rules in the plan-library Π. □

A.3 Proofs for Chapter 5

The following two definitions from (Erol et al., 1996) are used by some of the proofs in this section. The first definition is that of a completion of a task network. Intuitively, a completion of a primitive task network is an ordering and grounding of the primitive tasks in the task network, such that the ordering conforms with the constraints imposed on those tasks by the network.

Definition 30. (Completion of a Task Network (Erol et al., 1996)) Let $\sigma = act_1 \cdot \ldots \cdot act_m$ be a plan, Op be an operator-library, S_0 be the initial state, and $S_i = Res(act_i, S_{i-1}, Op)$ for $i \in \{1, \ldots, m\}$ be the intermediate states, which are all defined (i.e., the preconditions of each act_i are satisfied in S_{i-1} and thus actions in the plan are executable). Let $d = [\{(n_1 : act'_1), \ldots, (n_m : act'_m)\}, \phi]$ be a ground primitive task network, and ρ be a permutation such that whenever $\rho(i) = j$, $act'_i = act_j$. Then, $\sigma \in comp(d, S_0, \mathcal{D})$, if the constraint formula ϕ of d is satisfied. The constraint formula is evaluated as follows:

- $(c_i = c_j)$ is true, if c_i, c_j are the same constant symbols;
- $(n_i \prec n_j)$ is true if $\rho(i) < \rho(j)$;
- (l, n_i) is true if l holds in $S_{\rho(i)-1}$;
- (n_i, l) is true if l holds in $S_{\rho(i)}$;
- (n_i, l, n_j) is true if l holds for all S_k, $\rho(i) \leq k < \rho(j)$;
- $first[n_i, n_j, \ldots]$ evaluates to $min(\{\rho(i), \rho(j), \ldots\})$;
- $last[n_i, n_j, \ldots]$ evaluates to $max(\{\rho(i), \rho(j), \ldots\})$;
- logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic.

If d is a primitive task network containing variables, then

$$comp(d, S_0, \mathcal{D}) = \{\sigma \mid \sigma \in comp(d', S_0, \mathcal{D}),\ d' \text{ is a ground instance of } d\}.$$

If d contains compound tasks, then $comp(d, S_0, \mathcal{D}) = \emptyset$. ■

Next, we define what a HTN reduction means. Suppose that $d = [s, \phi]$ is a task network, $(n : t) \in s$ is a labelled compound task occurring in d, and that $m = (t', d') \in Me$ is a method that may be used to decompose t (i.e., t and t' unify). Then, $reduce(d, n, m)$ denotes the task network that results from decomposing labelled task $(n : t)$ in task network d using method m. Informally, such decomposition involves updating both the set s in d, by replacing labelled task $(n : \alpha)$ with the tasks in d' (by arbitrarily renaming task labels), and the constraints ϕ in s to take into account constraints in d'.

Definition 31. (HTN Reduction (Erol et al., 1996)) Let $d = [\{(n : t), (n_1 : t_1), \ldots, (n_m : t_m)\}, \phi]$ be a task network containing a non-primitive task t. Let $me = (t', [\{(n'_1 : t'_1), \ldots, (n'_k : t'_k)\}, \phi'])$ be a method,[1] and θ be the most general unifier of t and t'. Then,

$$reduce(d, n, me) = [\{(n'_1 : t'_1)\theta, \ldots, (n'_k : t'_k)\theta, (n_1 : t_1)\theta, \ldots, (n_m : t_m)\theta\}, \phi'\theta \wedge \psi],$$

where ψ is obtained from $\phi\theta$ with the following modifications:

- replace $(n \prec n_j)$ with $(last[n'_1, \ldots, n'_k] \prec n_j)$, as n_j must come after every task in the decomposition of n;
- replace $(n_j \prec n)$ with $(n_j \prec first[n'_1, \ldots, n'_k])$;
- replace (l, n) with $(l, first[n'_1, \ldots, n'_k])$, as l must be true immediately before the first task in the decomposition of n;
- replace (n, l) with $(last[n'_1, \ldots, n'_k], l)$, as l must be true immediately after the last task in the decomposition of n;
- replace (n, l, n_j) with $(last[n'_1, \ldots, n'_k], l, n_j)$;
- replace (n_j, l, n) with $(n_j, l, first[n'_1, \ldots, n'_k])$;
- everywhere that n appears in ϕ in a $first[]$ or a $last[]$ expression, replace it with $n'_1, \ldots, n'_k$.

[1] All variables and task labels in the method must be renamed with variables and task labels that do not appear anywhere else.

The set of reductions of d, denoted $red(d, \mathcal{I}, \mathcal{D})$, is defined as

$$red(d, \mathcal{I}, \mathcal{D}) = \{d' \mid d' \in reduce(d, n, me),\ n \text{ is the label for a non-primitive task in } d, \text{ and } me \text{ is a method in } \mathcal{D} \text{ for that task.}\}$$

■

Proof of Lemma 10 (p. 144). First we will prove that the induced decomposition tree $\mathcal{T}_\lambda$ of λ is a decomposition tree of d relative to $\mathcal{D}$. To this end, we will show that all conditions of Definition 23 (Decomposition Tree) are met for tree $\mathcal{T}_\lambda = \langle V, E, \ell_V \rangle$. Since task labels occurring in d_1 are unique, and since according to condition *(ii)(b)* of Definition 22 (Decomposition Trace), for each $i \in \{1, \ldots, n-1\}$, no task label occurring in $s_{i+1} \setminus s_i$ (with $d_i, d_{i+1} \in \lambda$, $d_i = [s_i, \phi_i]$ and $d_{i+1} = [s_{i+1}, \phi_{i+1}]$) also occurs in $d_1 \cdot \ldots \cdot d_i$, it follows that for each $(n : t) \in V$, n is a unique task label in the tree $\mathcal{T}_\lambda$. Moreover, due to Definition 24 (Induced Decomposition Tree), and from the fact that λ is ground, the first condition of Definition 23 holds. The second and fourth conditions of Definition 23 hold trivially due to Definition 24. Finally, we will show that the third condition of Definition 23 holds.

Observe from Definition 24 that for any internal non-root node $u = (n : t) \in V$, $children(u, \mathcal{T}_\lambda) = s_{i+1} \setminus s_i$, and $\phi_{i+1} = \phi_i \wedge \phi'$ (after appropriate modifications to ϕ_i, as described in Definition 31) for some constraint formula ϕ', where $u \in s_i$, $u \notin s_{i+1}$, and $i \in \{1, \ldots, n-1\}$. Since task network $d_{i+1} = reduce(d_i, n, m)$ for some task label n and method m, it follows that there exists a task network $[\hat{s}, \hat{\phi}] \in red([\{(n : t)\}, true], \mathcal{D})$ such that $\hat{s}\theta = s_{i+1} \setminus s_i$ and $\ell_V(u) = \hat{\phi}\theta = \phi'$. Hence, the third condition of Definition 23 holds, and $\mathcal{T}_\lambda$ is indeed a decomposition tree of d relative to $\mathcal{D}$.

Next, we will prove the second part of the theorem — i.e., that $\mathcal{T} = \langle V', E', \ell'_V \rangle$ is the induced decomposition tree of some decomposition trace of d — by induction on the number of non-leaf nodes k in V'.

Let $rt = (root : \epsilon)$. There are two base cases to consider. First, let us take $k = 0$. In this case, $V' = \{rt\}$, $E' = \emptyset$, and $\ell'_V = \{(rt, true)\}$, which is the induced decomposition tree of decomposition trace $[\emptyset, true]$. Second, let us take $k = 1$. Then, $V' = \{rt, (n_1 : t_1), \ldots, (n_m : t_m)\}$; $m > 0$; $E' = \{(rt, (n_1 : t_1)), \ldots, (rt, (n_m : t_m))\}$; $\ell'_V(rt) = \phi$ for some constraint formula ϕ; and $\ell'_V((n_i : t_i)) = true$ for all $i \in \{1, \ldots, m\}$. Hence, $\mathcal{T}$ is the induced decomposition tree of decomposition trace $[\{(n_1 : t_1), \ldots, (n_m : t_m)\}, \phi]$. Next, assume that the second part of the theorem holds if

$k \leq x$, for some $x \in \mathbb{N}_1$. Finally, suppose $k = x + 1$. From the induction hypothesis, we know that there is a decomposition tree $\mathcal{T}_{k-1}$ with $k - 1$ non-leaf nodes, such that $\mathcal{T}_{k-1}$ is the induced decomposition tree of some trace $[s_1, \phi_1] \cdot \ldots \cdot [s_j, \phi_j]$, with $j > 0$, and such that $\mathcal{T}_{k-1}$ is equivalent to $\mathcal{T}$ modulo the children in $\mathcal{T}$ of some leaf node in $\mathcal{T}_{k-1}$. More specifically, according to Definition 23 there must exist a node $(n : t) \in leaves(\mathcal{T}_{k-1})$, such that the following conditions hold: *(i)* there exists a non-empty set of ground labelled tasks $children((n : t), \mathcal{T}) = \{(n_1 : t_1)\theta, \ldots, (n_m : t_m)\theta\}$; *(ii)* $\ell'_V((n : t)) = \phi'\theta$ for some constraint formula ϕ'; and *(iii)* there exists a reduction $[s', \phi'] \in red([\{(n : t)\}, true], \mathcal{D})$ of t where $s' = \{(n_1 : t_1), \ldots, (n_m : t_m)\}$. Since $leaves(\mathcal{T}_{k-1}) = s_j$ according to Definition 24, it follows that $[s_1, \phi_1] \cdot \ldots \cdot [s_j, \phi_j] \cdot [(s_j \setminus \{(n : t)\}) \cup s'\theta, \phi'_j \wedge \phi'\theta]$ is a decomposition trace of d relative to Me (where ϕ'_j is obtained from ϕ_j as in Definition 31). Finally, $\mathcal{T}$ is the induced decomposition tree of this trace according to Definition 24, and the second part of the theorem holds. □

Proof of Lemma 11 (p. 147). We will prove this by induction on the length $k > 0$ of the (complete) decomposition trace $\lambda = d_1 \cdot \ldots \cdot d_k$. Let $\mathcal{T} = \langle V, E, \ell_V \rangle$.

[Base Case: $k = 1$.**]** In this case, d_k is a primitive task network (i.e., one that does not mention any compound tasks). If $d_k = [\emptyset, true]$, then the theorem holds trivially, as $comp(d_k, \mathcal{I}, \mathcal{D})$ consists of the empty plan; tree $\mathcal{T} = \langle \{(root : \epsilon)\}, \emptyset, \{((root : \epsilon), true)\} \rangle$; and the full tree $\mathcal{T}_\tau$, where τ is the empty plan, is executable in $\mathcal{I}$ relative to Op. If $d_k = [s_k, \phi_k]$, where s_k is a non-empty set, there are two cases to consider. For case $\Rightarrow$, suppose $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$. We will now show that $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to Op (Definition 25), where $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$. By taking permutation $\rho = 1 \cdot \ldots \cdot m$, and τ as the permutation of s_k for Definition 30, it is not difficult to see that constraint formula ϕ_k evaluates to true according to Definition 30 if and only if the formula evaluates to true according to Definition 26. Moreover, since $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$, $Res^*(act_1 \cdot \ldots \cdot act_m, \mathcal{I}, Op)$ also holds according to Definition 30. Therefore, $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to Op. The proof for case $\Leftarrow$ is similar.

[Induction Hypothesis] Assume that the theorem holds if $k \leq x$, for some $x \in \mathbb{N}_1$.

[Inductive Step] Suppose $k = x + 1$. Observe from Definition 22 (Decomposition Trace) that $d_2 =$

$reduce(d_1, n, me)$ (with $d_i = [s_i, \phi_i]$ for all $i \in \{1, \ldots, k\}$) for some task label n occurring in s_1 and ground method $me = (t, [s_{me}, \phi_{me}])$. Moreover, observe from Definition 31 (HTN Reduction) that $\phi_2 = \phi'_1 \wedge \phi_{me}$, (where ϕ'_1 is ϕ_1 after appropriate modifications), and that $s_2 = (s_1 \setminus \{(n : t)\}) \cup s_{me}$. Then, there are two cases to consider.

For case $\Rightarrow$, suppose $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$. From the induction hypothesis, there is a full decomposition tree $\mathcal{T}'_\tau$, with $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$, that is executable in $\mathcal{I}$ relative to Op, such that $\mathcal{T}' = \langle V', E', \ell'_V \rangle$ is the induced decomposition tree of $d_2 \cdot \ldots \cdot d_k$. Next, we prove that $\mathcal{T}_\tau$ is also executable in $\mathcal{I}$ relative to Op, by showing that all constraint formulas of labelled tasks in $\mathcal{T}$ are satisfied.

Let $rt = (root : \epsilon)$. Observe from Definition 24 that: *(i)* $V = V' \cup \{(n : t)\}$ (recall $(n : t)$ is the task that was reduced); *(ii)* $children((n : t), \mathcal{T}) = s_{me}$; *(iii)* $children(rt, \mathcal{T}) = \big(children(rt, \mathcal{T}') \setminus s_{me}\big) \cup \{(n : t)\}$; and that *(iv)* $\ell_V = \big(\ell'_V \setminus \{(rt, \phi_2)\}\big) \cup \{(rt, \phi_1), ((n : t), \phi_{me})\}$. Since, from the induction hypothesis, constraint formula $\ell'_V(rt) = \phi_2 = \phi'_1 \wedge \phi_{me}$ (for some formula ϕ'_1) is satisfied in $\mathcal{T}'_\tau$ relative to $\mathcal{I}$ and Op, it follows that $\ell_V((n : t)) = \phi_{me}$ is also satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and Op. Then, all we need to show is that $\ell_V(rt) = \phi_1$ is satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and Op. To this end, since constraints in ϕ'_1 represent the updated versions of those in ϕ_1 due to the reduction of task n using method me, we only need to consider the possible structural differences between the two formulas.

Consider the case where a constraint $(last[n'_1, \ldots, n'_i] \prec n_j)$ occurring in ϕ'_1 has the form $(n \prec n_j)$ in ϕ_1. According to Definition 26, $(last[n'_1, \ldots, n'_i] \prec n_j)$ evaluates to $max(\bigcup_{y \in \{1,\ldots,i\}} idx(n'_y)) < min(idx(n_j))$, which holds in $\mathcal{T}'_\tau$ due to the induction hypothesis. From the same definition, we know that $(n \prec n_j)$ is evaluated as $max(idx(n)) < min(idx(n_j))$. Since $\{(n'_1 : t_1), \ldots, (n'_i : t_i)\} = children((n : t), \mathcal{T})$ according to Definition 31, it follows that $idx(n) = \bigcup_{y \in \{1,\ldots,i\}} idx(n'_y)$ holds, and therefore, that $(n \prec n_j)$ is satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and Op. The remaining cases — e.g., where a constraint $(n_j \prec last[n'_1, \ldots, n'_i])$ occurring in $\ell'_V(rt)$ has the form $(n_j \prec n)$ in $\ell_V(rt)$ — can be proved similarly. The proof for case $\Leftarrow$ is similar to that of case $\Rightarrow$. □

Proof of Lemma 12 (p. 159). The only lines in the algorithm that are non-trivial are lines 5 and 11. Line 5 runs in polynomial time because $\Phi[\mathcal{T}_\tau, \pi]$ can be computed by first finding all the 2-permutations of set π using a nested *for* loop, and then determining for each pair $(n_1 : t_1), (n_2 : t_2)$ whether all leaves of n_1 in $\mathcal{T}$ occur before all leaves of n_2 in $\mathcal{T}$, with respect to τ. Line 11 runs

in polynomial for the following reason. Observe from Definition 26 that a constraint formula is evaluated by assigning truth values to each of its individual constraints, and then checking whether the resulting constraint formula is satisfied, which can be done in polynomial time. When assigning truth values to constraints, the only non-trivial part is in computing the state that results from applying a labelled primitive plan τ' to a state $\mathcal{I}$ — i.e., $Res^*(\tau', \mathcal{I}, Op)$ (see Section 2.3.1) — which simply requires checking if the ground precondition (set of literals) of each primitive action in τ' is met in some state, and then updating the state with the add and delete lists of the action. $\square$

Appendix B

Graphs and Trees

In this appendix, we define some notions to do with Graphs and Trees that are used in Chapter 5. We begin with the definition of a directed graph.

Definition 32. (Directed Graph) A *directed graph* G is the tuple $\langle V(G), E(G)\rangle$, where $V(G)$ is a set of vertices and $E(G) \subseteq V(G) \times V(G)$ is a set of edges. For any edge $(v_1, v_2) \in E(G)$, we call v_1 the *parent* vertex and v_2 the *child* vertex; moreover, we say that the edge is directed from v_1 to v_2. ■

Next, we provide an overview of some of the basic terms associated with directed graphs.

Definition 33. (Basic Graph Terminology) Let $G = \langle V, E\rangle$ be a directed graph.

- Graph G is *cyclic* if there exists a sequence of vertices $v_1 \cdot v_2 \cdot \ldots \cdot v_n \in V^n$, such that:
  - $\forall i \in \{1, \ldots, n-1\}$, $(v_i, v_{i+1}) \in E$, i.e. for each pair of adjacent vertices in the sequence, there is an edge directed from the left vertex of the pair to the right vertex of the pair; and
  - $(v_n, v_1) \in E$, i.e., there is an edge directed from the last vertex in the sequence to the first.
- Graph G is *acyclic* if it is not cyclic.
- Graph G is *rooted* if $|\{v \mid v \in V, \forall v' {\in} V \ ((v', v) \notin E)\}| = 1$, i.e., there is exactly one vertex without a parent vertex. Given a rooted, directed graph $G' = \langle V', E'\rangle$, the *root* of G', denoted $root(G')$, is the vertex $v \in V'$ such that for each $v' {\in} V'$, $(v', v) \notin E'$.

- A *tree* is a rooted and acyclic directed graph. ■

Next, we provide an overview of some of the basic terms associated with trees.

Definition 34. (Basic Tree Terminology) Let $G = \langle V, E \rangle$ be a tree.

- The *children* of a vertex $v \in V$ in G, denoted $children(v, G)$, is the set $\{v' \mid (v, v') \in E\}$.
- The *descendants* of a vertex $v \in V$ in G, denoted $descendants(v, G)$, is defined inductively as follows:

$$descendants(v, G) = children(v, G) \cup \bigcup_{v' \in children(v,G)} descendants(v', G).$$

- The *leaves* of G, denoted $leaves(G)$, is the set $\{v \mid v \in V, \ (v, v') \notin E\}$. Moreover, the *leaves* of a vertex $v \in V$ in G, denoted $leaves(v, G)$, is the set $(descendants(v, G) \cup \{v\}) \cap leaves(G)$.

■

Finally, we define a vertex-labelled tree as follows.

Definition 35. (Vertex-labelled Tree) Let L_V be a finite set of labels. A *vertex-labelled tree* is the tuple $\langle V, E, \ell_V \rangle$, where:

- $\langle V, E \rangle$ is a tree; and
- $\ell_V : V \mapsto L_V$ is a function that assigns each vertex with a label — given a vertex $v \in V$, we say that $\ell_V(v)$ is the *label* of v. ■

Bibliography

Agre, P. E. and Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pages 268–272.

Ambros-Ingerson, J. (1987). *IPEM: Integrated Planning, Execution, and Monitoring*. PhD thesis, Department of Computer Science, University of Essex, U.K.

Baier, J. A., Fritz, C., and McIlraith, S. A. (2007). Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 26–33.

Barringer, H., Fisher, M., Gabbay, D., Gough, G., and Owens, R. (1989). METATEM: A framework for programming in temporal logic. In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430, pages 94–129. Springer.

Benfield, S. S., Hendrickson, J., and Galanti, D. (2006). Making a strong business case for multiagent technology. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-06)*, pages 10–15.

Blum, A. and Furst, M. (1995). Fast planning through planning graph analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1636–1642.

Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *Proceedings of the European Conference on Planning (ECP-99)*, pages 360–372.

Bordini, R. H., Bazzan, A. L. C., de O. Jannone, R., Basso, D. M., Vicari, R. M., and Lesser, V. R. (2002). AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task

scheduling. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, pages 1294–1302.

Bordini, R. H., Fisher, M., Pardavila, C., and Wooldridge, M. (2003). Model checking AgentSpeak. In *Proceedings of the International Joint Conference on Autonomous agents and Multiagent Systems (AAMAS-03)*, pages 409–416.

Bordini, R. H. and Moreira, Á. F. (2004). Proving BDI properties of agent-oriented programming languages. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):197–226.

Bordini, R. H., Wooldridge, M., and Hübner, J. F. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons.

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research (JAIR)*, 24:581–621.

Bratman, M. E. (1987a). *Intention, Plans and Practical Reason*. Harvard University Press.

Bratman, M. E. (1987b). What is intention? In *Intentions in Communication*, pages 15–32. MIT Press.

Bratman, M. E., Israel, D., and Pollack, M. (1991). Plans and resource-bounded practical reasoning. In *Philosophy and AI: Essays at the Interface*, pages 1–22. MIT Press.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

Brooks, R. A. (1990). Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15.

Busetta, P., Rönnquist, R., Hodgson, A., and Lucas, A. (1999). JACK Intelligent Agents - components for Intelligent Agents in Java, AgentLink News Letter, Agent Oriented Software Pty. Ltd., Melbourne, Australia.

Clark, K. L. (1978). Negation as failure. In *Logic and Data Bases*, pages 293–322.

Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model Checking*. MIT Press.

Claßen, J., Eyerich, P., Lakemeyer, G., and Nebel, B. (2007). Towards an integration of Golog and planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 1846–1851.

Clement, B. J. and Durfee, E. H. (1999). Theory for coordinating concurrent hierarchical planning agents using summary information. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-99)*, pages 495–502.

Clement, B. J. and Durfee, E. H. (2000). Exploiting domain knowledge with a concurrent hierarchical planner. In *Proceedings of the Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning, Working Notes*, pages 57–62.

Clement, B. J., Durfee, E. H., and Barrett, A. C. (2007). Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research (JAIR)*, 28:453–515.

Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42:213–261.

da Costa Móra, M., Lopes, J. G. P., Vicari, R. M., and Coelho, H. (1998). BDI models and systems: Bridging the gap. In *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, pages 11–27. Springer.

Dastani, M. (2008). 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 16(3):214–248.

Dastani, M., van Riemsdijk, M. B., Dignum, F., and Meyer, J.-J. C. (2003). A programming language for cognitive agents: Goal directed 3APL. In *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS-03)*, pages 111–130. Springer.

De Giacomo, G. and Levesque, H. (1999). An incremental interpreter for high-level programs with sensing. In *Logical Foundation for Cognitive Agents: contributions in honor of Ray Reiter*, pages 86–102. Springer.

de Silva, L. and Dekker, A. (2007). Planning with time limits in BDI agent programming languages. In *Proceedings of Computing: the Australasian Theory Symposium (CATS-07)*, pages 131–139.

de Silva, L. and Padgham, L. (2004). A comparison of BDI based real-time reasoning and HTN based planning. In *Proceedings of the Australian Joint Conference on Artificial Intelligence (AI-04)*, pages 1167–1173.

de Silva, L. and Padgham, L. (2005). Planning on demand in BDI systems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-05) Poster Session*, pages 37–40.

de Silva, L., Sardina, S., and Padgham, L. (2009). First Principles Planning in BDI systems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-09)*, pages 1105–1112.

Dekker, A. and de Silva, L. (2006). Investigating organisational structures with networks of planning agents. In *Proceedings of the International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC-06)*, pages 25–30.

Despouys, O. and Ingrand, F. F. (1999). Propice-Plan: Toward a unified framework for planning and execution. In *Proceedings of the European Conference on Planning (ECP-99)*, pages 278–293.

d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1998). A formal specification of dMARS. In *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, pages 155–176. Springer.

d'Inverno, M. and Luck, M. (1998). Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):233–260.

Dix, J., Muñoz-Avila, H., Nau, D. S., and Zhang, L. (2003). IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence*, 37(4):381–407.

Do, M. B. and Kambhampati, S. (2001). Sapa: A domain-independent heuristic metric temporal planner. In *Proceedings of the European Conference on Planning (ECP-01)*, pages 109–120.

Erol, K., Hendler, J., and Nau, D. S. (1994). Semantics for hierarchical task-network planning. Technical Report UMIACS-TR-94-31, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, U.S.A.

Erol, K., Hendler, J. A., and Nau, D. S. (1996). Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93.

Ferguson, I. A. (1992). Touring machines: Autonomous agents with attitudes. *IEEE Computer*, 25(5):51–55.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208.

Fink, E. and Yang, Q. (1992). Formalizing plan justifications. In *Proceedings of the Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-92)*, pages 9–14.

Firby, R. J. (1987). An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pages 202–206.

Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Computer Science Department, Yale University, U.S.A.

Fisher, M. (1994). A survey of concurrent METATEM - the language and its applications. In *Proceedings of the International Conference on Temporal Logic (ICTL-94)*, pages 480–505.

Franklin, S. and Graesser, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the International Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages (ATAL-97)*, pages 21–35. Springer.

Fritz, C., Baier, J. A., and McIlraith, S. A. (2008). ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for planning and beyond. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-08)*, pages 600–610.

Gabbay, D. M., Hogger, C. J., and Robinson, J. A., editors (1994). *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press.

Georgeff, M. and Ingrand, F. (1989). Decision making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972–978.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc.

Graham, J. R., Decker, K. S., and Mersic, M. (2003). DECAF - a flexible multi agent system architecture. *Autonomous Agents and Multiagent Systems (JAAMAS)*, 7(1-2):7–27.

Gupta, N. and Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254.

Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (1998). A formal embedding of AgentSpeak(L) in 3APL. In *Selected papers from the Australian Joint Conference on Artificial Intelligence*, pages 155–166. Springer.

Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (1999). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 2(4):357–401.

Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (2000). Agent programming with declarative goals. In *Proceedings of the International Workshop on Intelligent Agents VII, Agent Theories, Architectures, and Languages (ATAL-00)*, pages 228–243. Springer.

Hoffmann, J. (2003). The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)*, 20:291–341.

Hoffmann, J. and Brafman, R. I. (2006). Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6-7):507–541.

Hoffmann, J. and Edelkamp, S. (2005). The deterministic part of ipc-4: an overview. *Journal of Artificial Intelligence Research (JAIR)*, 24(1):519–579.

Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302.

Huber, M. J. (2001). JAM agents in a nutshell, version 0.61+0.79i. http://www.marcush.net/irs/jam/jam-man-01nov01-draft.htm.

Hübner, J. F., Bordini, R. H., and Wooldridge, M. (2006). Programming declarative goals using plan patterns. In *Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT-06)*, pages 123–140. Springer.

Ingrand, F. F., Georgeff, M. P., and Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert Magazine*, 7(6):33–44.

Jarvis, J., Jarvis, D., and McFarlane, D. (2003). Achieving holonic control: an incremental approach. *Computers in Industry*, 51(2):211–223.

Kaelbling, L. P. (1987). An architecture for intelligent reactive systems. In *Proceedings of the Workshop on Reasoning about Actions and Plans*, pages 395–410. Morgan Kaufmann.

Kambhampati, S., Mali, A. D., and Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*, pages 882–888.

Knoblock, C. (1995). Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1686–1693.

Knoblock, C., Tenenberg, J. D., and Yang, Q. (1991). Characterizing abstraction hierarchies for planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-91)*, pages 692–697.

Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proceedings of the European Conference on Planning (ECP-97)*, pages 273–285.

Laborie, P. and Ghallab, M. (1995). Planning with sharable resource constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1643–1651.

Lemai, S. and Ingrand, F. F. (2004). Interleaving temporal planning and execution in robotics domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-04)*, pages 617–622.

Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Scherl, R. B. (1995). Foundations of a logical approach to agent programming. In *Proceedings of the International Workshop on Intelligent Agents II, Agent Theories, Architectures, and Languages (ATAL-95)*, pages 331–346. Springer.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83.

Ljungberg, M. and Lucas, A. (1992). The OASIS air-traffic management system. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI-92)*, pages 15–18.

Lotem, A., Nau, D. S., and Hendler, J. (1999). Using planning graphs for solving HTN problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-99)*, pages 534–540.

Lyons, D. M., Hendriks, A., and Mehta, S. (1991). Achieving robustness by casting planning as adaptation of a reactive system. In *IEEE International Conference on Robotics and Automation (ICRA-91)*, pages 198–203.

Machado, R. and Bordini, R. H. (2002). Running AgentSpeak(L) agents on SIM_AGENT. In *Revised Papers from the International Workshop on Intelligent Agents VIII, Agent Theories, Architectures, and Languages (ATAL-02)*, pages 158–174. Springer.

Maes, P. (1989). The dynamics of action selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 991–997.

Mccarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502.

Mcdermott, D. (1991). A reactive plan language. Technical Report CSD-RR-864, Computer Science Department, Yale University, U.S.A.

Mcdermott, D. (1992). Transformational planning of reactive behavior. Technical Report YALEU/DCS/RR-941, Computer Science Department, Yale University, U.S.A.

Meneguzzi, F. and Luck, M. (2007). Composing high-level plans for declarative agent programming. In *Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT-07)*, pages 69–85. Springer.

Meneguzzi, F. and Luck, M. (2008). Leveraging new plans in AgentSpeak(PL). In *Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT-08)*, pages 111–127. Springer.

Meneguzzi, F., Zorzo, A. F., and da Costa Móra, M. (2004a). Mapping mental states into propositional planning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-04)*, pages 1514–1515.

Meneguzzi, F., Zorzo, A. F., and da Costa Móra, M. (2004b). Propositional planning in BDI agents. In *Proceedings of the ACM Symposium on Applied Computing (SAC-04)*, pages 58–63.

Minton, S., Bresina, J., and Drummond, M. (1994). Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research (JAIR)*, 2:227–262.

Moreira, Á. and Bordini, R. (2002). An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02)*, pages 45–59.

Moreira, Á. F., Vieira, R., and Bordini, R. H. (2003). Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT-03)*, pages 135–154. Springer.

Morley, D. and Myers, K. (2004). The SPARK agent framework. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-04)*, pages 714–721.

Müller, J. P. (1997). A cooperation model for autonomous agents. In *Proceedings of the International Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages (ATAL-97)*, pages 245–260. Springer.

Muñoz-Avila, H., Aha, D. W., Nau, D. S., Weber, R., Breslow, L., and Yaman, F. (2001). SiN: Integrating case-based reasoning with task decomposition. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 999–1004.

Nau, D., Au, T.-C., Ilghami, O., Kuter, U., Muñoz-Avila, H., Murdock, J. W., Wu, D., and Yaman, F. (2005). Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41.

Nau, D., Cao, Y., Lotem, A., and Muñoz-Avila, H. (1999). SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 968–973.

Nau, D. S. (2007). Current trends in automated planning. *AI Magazine*, 28(4):43–58.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404.

Nau, D. S., Smith, S. J. J., and Erol, K. (1998). Control strategies in HTN planning: Theory versus practice. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*, pages 1127–1133.

Nebel, B. (2000). What is the expressive power of disjunctive preconditions? In *Proceedings of the European Conference on Planning (ECP-99)*, pages 294–307.

Nebel, B. and Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76:427–454.

Paolucci, M., Kalp, D., Pannu, A., Shehory, O., and Sycara, K. (1999). A planning component for RETSINA agents. In *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, pages 147–161.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, University of Aarhus, Denmark.

Pokahr, A., Braubach, L., and Lamersdorf, W. (2003). Jadex: Implementing a BDI-infrastructure for JADE agents. *EXP - in search of innovation*, 3(3):76–85.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the European workshop on Modelling Autonomous Agents in a Multi-Agent World : agents breaking away (MAAMAW-96)*, pages 42–55. Springer.

Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-91)*, pages 473–484.

Rao, A. S. and Georgeff, M. P. (1992). An abstract architecture for rational agents. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 439–449.

Rao, A. S. and Georgeff, M. P. (1995). BDI-agents: from theory to practice. In *Proceedings of the International Conference on Multiagent Systems (ICMAS-95)*, pages 312–319.

Refanidis, I. and Vlahavas, I. (2002). The MO-GRT system: Heuristic planning with multiple criteria. In *Proceedings of the Workshop on Planning and Scheduling with Multiple Criteria*.

Reiter, R. (1987). On closed world data bases. In *Readings in Nonmonotonic Reasoning*, pages 300–310. Morgan Kaufmann Publishers Inc.

Russell, S. J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.

Sardina, S., De Giacomo, G., Lespérance, Y., and Levesque, H. J. (2004). On the semantics of deliberation in IndiGolog—from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2-4):259–299.

Sardina, S., de Silva, L., and Padgham, L. (2006). Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-06)*, pages 1001–1008.

Sardina, S. and Padgham, L. (2007). Goals in the context of BDI plan failure and planning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-07)*, pages 16–23.

Sardina, S. and Padgham, L. (2010). A BDI agent progarmming language with failure recovery, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*. In Press; accepted for publication 16/3/2010.

Schoppers, M. (1987). Universal plans for reactive robots in unpredictable environments. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046.

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92.

Sohrabi, S., Baier, J. A., and McIlraith, S. A. (2009). HTN planning with preferences. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-09)*. (to appear).

Spivey, J. M. (1989). *The Z notation: A Reference Manual*. Prentice Hall.

Steels, L. (1990). Cooperation between distributed agents through self-organisation. In *Decentralized A.I. : Proceedings of the European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 175–196. North-Holland.

Tambe, M. and Zhang, W. (2000). Towards flexible teamwork in persistent teams: Extended report. *Autonomous Agents and Multi-Agent Systems*, 3(2):159–183.

Thangarajah, J., Padgham, L., and Winikoff, M. (2003a). Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 721–726.

Thangarajah, J., Padgham, L., and Winikoff, M. (2003b). Detecting and exploiting positive goal interaction in intelligent agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, pages 401–408.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of PDDL axioms. *Artificial Intelligence*, 168(1):38–69.

van Riemsdijk, B., van der Hoek, W., and Meyer, J.-J. C. (2003). Agent programming in dribble: from beliefs to goals using plans. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, pages 393–400.

van Riemsdijk, M. B., Dastani, M., and Meyer, J.-J. C. (2005). Semantics of declarative goals in agent programming. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-05)*, pages 133–140.

Veloso, M. M., Pollack, M. E., and Cox, M. T. (1998). Rationale-based monitoring for planning in dynamic environments. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 171–180.

Wallis, P., Rönnquist, R., Jarvis, D., and Lucas, A. (2002). The automated wingman - using JACK Intelligent Agents for unmanned autonomous vehicles. In *Proceedings of the IEEE Aerospace Conference*, pages 2615–2622.

Wilkins, D. E. (1990). Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246.

Wilkins, D. E. and Myers, K. L. (1995). A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761.

Wilkins, D. E. and Myers, K. L. (1998). A multiagent planning architecture. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 154–162.

Wilkins, D. E., Myers, K. L., Lowrance, J. D., and Wesley, L. P. (1995). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):197–227.

Winikoff, M., Padgham, L., Harland, J., and Thangarajah, J. (2002). Declarative and procedural goals in intelligent agent systems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*, pages 470–481.

Wobcke, W. (2001). An operational semantics for a PRS-like agent architecture. In *Proceedings of the Australian Joint Conference on Artificial Intelligence (AI-01)*, pages 569–580.

Wooldridge, M. (2002). *An Introduction to Multiagent Systems*. John Wiley & Sons.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10:115–152.