Robust Execution of Belief-Desire-Intention-Based Agent Programs



Yuan Yao

School of Computer Science

The University of Nottingham

This dissertation is submitted for the degree of

Doctor of Philosophy

September 2017

I would like to dedicate this thesis to my loving grandparents…

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

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

- Yao, Y., Logan, B., and Thangarajah, J. (2014). SP-MCTS-based Intention Scheduling for BDI Agents. In Proceedings of the 21st European Conference on Artificial Intelligence (ECAI-2014), pages 1133-1134, Prague, Czech Republic. IOS Press.

- Yao, Y., Logan, B., and Thangarajah, J. (2016). Robust execution of BDI agent programs by exploiting synergies between intentions. In Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16), pages 2558–2565, Phoenix, USA. AAAI, AAAI Press.

- Yao, Y. and Logan, B. (2016). Action-level intention selection for BDI agents. In Proceedings of the 15th International Conference on

Autonomous Agents and Multiagent Systems (AAMAS 2016), pages 1227–1236, Singapore. IFAAMAS, IFAAMAS.

- Yao, Y., Logan, B., and Thangarajah, J. (2016). Intention Selection with Deadlines. In Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-2016), pages1700-1701, The Hague, The Netherlands. IOS Press.

- Yao, Y., de Silva, L., and Logan, B. (2016). Reasoning about the Executability of Goal-Plan Trees. In Proceedings of the 4th International Workshop on Engineering Multi-Agent Systems (EMAS 2016), pages 181-196, Singapore.

Yuan Yao

September 2017

Acknowledgements

I would like to express my deep gratitude to Dr. Brian Logan, my principal supervisor for his scientific advice and knowledge, and insightful discussion and suggestion. He taught me a lot during the last four years, and this thesis would not exist at all if it were not for his tireless help and encouragement. I would also like to thank Dr. Natasha Alechina, my second supervisor for not only the constructive feedback, but also her constant support in my study as well as my life. 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, whether it be networking, scholarships, travel funding, or job interviews. I could not have asked for better supervisors.

I would like to thank all the people I have met and worked with, especially Dr. John Thangarajah for the discussions of using goal-plan trees to evaluate my approach, and Dr. Lavindra de Silva for the discussions of extending the existing goal-plan trees structure. I would also like to thank Dr. Koen Hindriks and Professor Tony Pridmore for their valuable comments which helped improve this thesis, and Dr. Andrew Parks who has provided constructive feedback on my research.

I would also like to thank the University of Nottingham and the School of Computer Science for offering me a scholarship to do my PhD and for all the

resources made available to me throughout my candidature. I also thank the various technical support staff and administrative staff that have assisted me at different stages of my candidature.

Finally, I would like to thank my parents who have supported my education all my life. A special thank you to my colleagues in Agent Lab, my wonderful friends in Nottingham, for your invaluable help and support during last four years. Thank you!

Abstract

Belief-Desire-Intention (BDI) agent systems are a popular approach to building intelligent agents for complex and dynamic domains. In the BDI approach, agents select plans to achieve their goals based on their beliefs. When BDI agents pursue multiple goals in parallel, the interleaving of steps in different plans to achieve goals may result in conflicts, e.g., where the execution of a step in one plan makes the execution of a step in another concurrently executing plan impossible. Conversely, plans may also interact positively with each other, e.g., where the execution of a step in one plan assists the execution of a step in other concurrently executing plans. To avoid negative interactions and exploit positive interactions, an intelligent agent should have the ability to reason about the interactions between its intended plans.

We propose *SAM*, an approach to scheduling the progression of an agent's intentions (intended plans) based on Monte-Carlo Tree Search and its variant Single-Player Monte-Carlo Tree Search. *SAM* is capable of selecting plans to achieve an agent's goals and interleaving the execution steps in these plans in a domain-independent way. In addition, *SAM* also allows developers to customise how the agent's goals should be achieved, and schedules the progression of the agent's intentions in a way that best satisfies the requirements of a particular application. To illustrate the flexibility of SAM, we show how our approach can be configured to prioritise criteria relevant in a range of different

scenarios. In each of these scenarios, we evaluate the performance of SAM and compare it with previous approaches to intention progression in both synthetic and real-world domains.

Table of contents

List of figures **xv**

List of tables **xvii**

1 Introduction **1**

2 Background **13**

2.1 Introduction to Agents 13

2.2 The Belief-Desire-Intention Agent Architecture 15

2.2.1 The Abstract BDI Interpreter 18

2.3 BDI-based Agent Programming Languages and Platforms . . . 21

2.4 Approaches to Reasoning About Interactions Between Intentions 43

2.4.1 Goal-Plan Trees 43

2.4.2 Summary-Information-Based Approach 45

2.4.3 Coverage-Based Approach 48

2.4.4 Petri Nets Approach and Constraints-Based Model . . 49

2.5 Summary 51

3 Reasoning About Goal-Plan Trees **53**

3.1 Beliefs, Goals, Plans and Actions 53

3.2 Goal-Plan Trees With Actions 56
3.3 The Intention Progression Problem 60

4 The *SAM* Scheduler 65

4.1 Monte-Carlo Methods 65
4.2 Multi-Armed Bandit Problem 67
4.3 Monte-Carlo Tree Search 69
4.3.1 Single-Player Monte-Carlo Tree Search 73
4.4 The *SAM* Scheduler 76
4.4.1 *SAM* and Agent's Deliberation Cycle 87

5 Avoiding Conflicts Between Intentions 91

5.1 Avoiding Conflicts 91
5.2 Problem Specification and *SAM* Configuration 93
5.3 Evaluation 97
5.3.1 Synthetic Domain 101
5.3.2 Miconic-10 Elevator Domain 114
5.4 Summary 116

6 Intention Progression With Deadlines 117

6.1 Real Time Agency 117
6.2 Problem Specification and *SAM* Configuration 120
6.3 Evaluation 121
6.3.1 Synthetic Domain 125
6.3.2 Blocks World Domain 133
6.4 Summary 137

7 Exploiting Synergies 139

7.1 Scheduling With Fallible Actions 139

7.1.1 Recovering From Execution Failure 141

7.1.2 Problem Specification and *SAM* Configuration 142

7.1.3 Evaluation 144

7.2 Saving Execution Steps 159

7.2.1 Merging Actions 159

7.2.2 Problem Specification and *SAM* Configuration 160

7.2.3 Evaluation 162

7.3 Summary 169

8 Computational Overhead 171

8.1 Anytime *SAM* 171

8.2 Experiments 173

9 Conclusion and Future Work 175

References 183

Appendix A GenGPT 191

List of figures

2.1 Agent and environment. 14

2.2 Deliberation cycle in 3APL [26] 31

2.3 Deliberation cycle in 2APL [23] 35

2.4 An example of goal-plan tree 45

3.1 BNF of goal-plan tree with actions 57

3.2 An example of goal-plan tree. 59

4.1 One iteration of MCTS 71

4.2 Asymmetric search tree [21] 73

5.1 An example of the synthetic goal-plan tree. 105

6.1 Goal-plan tree for the Blocks World domain 134

7.1 MoveTo Action with unintended postcondition 155

7.2 Goal-plan tree for the nondeterministic Miconic-10 elevator domain 156

8.1 Number of merged actions with increasing α and β 174

8.2 Computational overhead with increasing α and β 174

A.1 An example of synthetic goal-plan tree. 194

A.2 XML representation of goal-plan trees 195

List of tables

5.1 Goals & variance with increasing % p-effects 106

5.2 Goals & variance with increasing % 1-plan (100% p-effects) . 108

5.3 Goals & variance with increasing % 1-plan (100% p-effects, average p-effect size 5) 109

5.4 Goals & variance with increasing % 1-plan (100% p-effects, each plan contains 5 actions) 110

5.5 Goals & variance with increasing dynamism (100% p-effects, 0% 1-plan) 111

5.6 Goals & variance with increasing dynamism (100% p-effects, 25% 1-plan) 112

5.7 Goals & variance with increasing dynamism (100% p-effects, p-effect size 5) 112

5.8 Goals & variance with increasing dynamism (100% p-effects, action number 5) 113

5.9 Goals & variance for 10 goals 115

5.10 Goals & variance with decreasing time between goals 116

6.1 Goals & tardiness with increasing numbers of atomic goal-plan trees 129

6.2 Goals & tardiness with decreasing average deadline 130

6.3 Goals & tardiness with and without estimation error 131

6.4 Tardiness with increasing numbers of atomic goal-plan trees . 131

6.5 Tardiness with decreasing average preferred achievement time 132

6.6 Tardiness with and without estimation error 132

6.7 Goals and tardiness with decreasing deadline 136

6.8 tardiness with decreasing preferred achievement time 137

7.1 *RR*+; all fallible actions fail non-disruptively, 10 goals 148

7.2 *SAM*; all fallible actions fail non-disruptively, 10 goals 148

7.3 *RR*+; all fallible actions fail non-disruptively, 5 top-level goals 150

7.4 *SAM*; all fallible actions fail non-disruptively, 5 top-level goals 151

7.5 *RR*+; all fallible actions fail disruptively, 10 goals 152

7.6 *SAM*; all fallible actions fail disruptively, 10 goals 153

7.7 *RR*+; all fallible actions fail disruptively, 5 goals 154

7.8 *SAM*; all fallible actions fail disruptively, 5 goals 154

7.9 *RR*+; non-deterministic elevator domain, 10 goals 156

7.10 *SAM*; non-deterministic elevator domain, 10 goals 157

7.11 *RR*+; non-deterministic elevator domain, 10 floors 158

7.12 *SAM*; non-deterministic elevator domain, 10 floor 158

7.13 Goals & the number of merged actions with decreasing number of environment variables, each plan contains 3 actions 164

7.14 Goals & the number of merged actions with increasing number of actions in each plan 165

7.15 Goals & the number of merged actions with increasing number of top-level goals 166

7.16 Goals & the number of merged actions with increasing number of top-level goals, the number of cells is set to 20 168

7.17 Goals & the number of merged actions with increasing number of top-level goals, the number of cells is set to 10 168

A.1 Arguments required to generate the goal-plan trees 192

Chapter 1

Introduction

Agent-based systems are a promising paradigm for building software that exhibits a high degree of autonomy and operates in dynamic, complex domains. Agents have been used to develop applications in a wide variety of domains, including power systems [7], transportation systems [16], spacecraft [44] and mobile devices [13].

The term "agent" is widely used in different areas, including Engineering, Computer Science and many others, but there is no widely accepted definition of it. In the definition proposed in [73], an agent is defined as an autonomous entity which is situated in some environment and achieves its design objectives by perceiving information and performing actions upon that environment.

One popular architecture used by programmers to build agent-based systems is the Belief-Desire-Intention (BDI) architecture, which is based on the philosophical theory of practical reasoning [10]. In BDI-based agent systems, the behaviour of an agent is specified in terms of beliefs, goals, and plans. *Beliefs* represent the agent's information about the environment (and itself). *Goals* represent desired states of the environment the agent is trying to bring

about. *Plans* are the means by which the agent can modify the environment in order to achieve its goals. Plans are composed of *steps* which are either *basic actions* that directly change the agent's environment or *subgoals* which are in turn achieved by other plans. For each top-level goal, the agent selects a plan which forms the root of an *intention*, and commences executing the steps in the plan. If the next step in an intention is a subgoal, a (sub)plan is selected to achieve the subgoal and added to the intention, and the steps in the (sub)plan are then executed and so on.

Many agent programming languages and platforms based on the BDI agent architecture have been developed. The Procedural Reasoning Systems (PRS), which was developed by SRI International, was perhaps the first agent system based on the BDI agent architecture [33]. In PRS, beliefs are represented as Lisp-like first order formulas and agents are provided with a library of plans which are used to perform means-ends reasoning. PRS has been used to build many different kinds of applications to date including the reaction control system (RCS) on the NASA space shuttle [32], and a network management monitor called the Interactive Real-time Telecommunications Network Management System (IRTNMS) for Telecom Australia [71]. There are also many other agent programming languages and platforms developed using the BDI architecture. For example, AgentSpeak(L) [45], Jason [9], 2APL [23], 3APL [35, 25, 26] and JACK [68].

In many agent-based applications, agents are often required to pursue multiple goals in parallel. Even if a set of goals are consistent, the ways of achieving those goals could possibly result in *conflicts*. The conflicts can arise from the availability of resources, e.g., money to purchase some goods, the

fuel level of a vehicle or the availability of the machines in a factory. These resources can be categorised as *reusable resources* which are made available after use (e.g., the machines in a factory), and *consumable resources* which are no longer available after use (e.g., fuel). Both kinds of the resources can affect the agent and result in potential failures. For example, consider a Mars Rover which has a goal G_0 to perform a rock experiment at location A and another goal G_1 to recharge its battery at location B. Suppose its current battery level only allows the Mars Rover to move from the current position to A or B, and it cannot move from A to B (or vice versa) without recharging its battery. In this situation, the only way to achieve both goals is to pursue the goal of recharging the battery (G_1) first.

Apart from the availability of resources, conflicts can also be caused by the interleaving of execution steps in achieving different goals. For example, consider a Mars Rover which has a goal G_0 to perform a rock experiment at location A, and another goal G_1 to perform a soil experiment at location B. Suppose a plan P_0 consisting of two actions, move to location A and perform a rock experiment at A, is selected to achieve G_0, and plan P_1 consisting of two actions, move to location B and perform a soil experiment at B, is selected to achieve G_1. If the Mars Rover chooses to progress each plan in turn, i.e., it first moves to location A, and then moves to location B, then the action to perform rock experiment at location A will fail. Obviously, the action to perform the rock experiment at location A relies on the agent being at location A. Any actions or plans that move the agent to other locations will potentially cause execution failures in achieving G_0.

However, not all interactions between the intentions are negative. Agents may also benefit from exploiting *synergies* while achieving different goals in parallel. Synergy happens when the execution of a step in achieving one goal assists the achievement of other goals, or the actions and plans which are used to achieve different goals produce similar effects. Exploiting synergies allows the agent to behave more efficiently and intelligently. For example, consider a Mars Rover which has a goal G_0 to perform a rock experiment at location A, a goal G_1 to perform a rock experiment at location B and another goal G_2 to perform a soil experiment at location A. A Mars Rover that doesn't have any reasoning ability might achieve its goals as follows: move to location A, perform the rock experiment, move to location B, perform the rock experiment, move to location A, perform the soil experiment. If the Mars Rover is clever enough, it might achieve its goals in a more sensible way, i.e., move to location A, perform the rock experiment, perform the soil experiment, move to location B, perform the rock experiment. By merging the actions or plans which have the same effects (e.g., move to location A in the example above), the agent is able to save execution time when achieving its goals. Moreover, *synergies* could also help the agent to recover from execution failures. An execution failure occurs when it is not possible to execute the intention chosen for execution. Instead of dropping the failed intention, it may be possible for the agent to opportunistically recover from an execution failure by progressing a progressable intention, which reestablishes the missing condition in the failed intention. For example, consider a robot agent which has two goals. The first goal G_0 is to deliver a package to location A. A plan P_0, which consists of a sequence of three actions, pick up the package, move to location A, release the

package, is selected to achieve this goal. The second goal G_1 is to recharge its battery at location B. A plan P_1, which consists of a sequence of two actions, move to location B, recharge battery, is selected to achieve G_1. Suppose, the robot was intending to progress P_0 and failed, as the package was moved to location B by another robot. In this situation, instead of dropping the goal G_0, the robot could choose to progress plan P_1 which moves the robot to location B. After that, the robot is able to pick up the package, and thus recover from execution failure.

In order to avoid or reduce conflicts, or to exploit synergies, the agent must have the ability to reason about the progression of its intentions, i.e., which plans are used to achieve the (sub)goals, and in which order these plans should be executed. These choices and decisions can be difficult as there are very many different ways in which the agent may choose to accomplish its goals. These choices cannot be pre-determined and must be made at run time, as the environment is dynamic. However, most agent programming languages and platforms either use very simple approaches to make these decisions, or to allow the agent programmers to program a customised strategy tailored to a particular problem. For example, Jason [9] uses the intention selection function S_I to determine which intention to progress next. Programmers can customise S_I for different problems. Cutomising S_I may work well for problems where the programmers know how the agent should progress its intentions. However, for problems where the programmers do not know how to define S_I (i.e., S_I is not specified by the programmer), the agent will progress one step in each of its intentions in turn, which is known as the "Round-Robin" scheduling, and is likely to cause conflicts as the intentions are interleaved arbitrarily. Apart from

the approaches used in practical agent programming language and platforms, a number of approaches to reasoning about possible interactions between intentions have been proposed in the literature. For example, Thangarajah et al. [57, 56] describe an approach based on *summary information* that avoids conflicts by reasoning about necessary and possible pre-, in- and postconditions of different ways of achieving a goal. They give algorithms for computing summary information at compile time, and for dynamically updating it at run-time. They also present mechanisms to determine whether a newly adopted (sub)goal will definitely be safe to execute without conflicts, or will definitely result in conflicts, or may result in conflicts. If the goal cannot be executed safely, execution of the intention is deferred. The summary information based approach has also been used to reason about resource conflicts [60] and to exploit positive interactions (synergies) [58]. Shaw et al. developed two methods to reason about the conflicts and synergies between different goals. The first method uses Petri nets to represent and reason about the goals [54], while the second uses constraint satisfaction techniques to find an effective way to achieve the goals [55]. However these approaches are limited to interleaving intentions at the plan level, i.e., execution of another intention is only possible when a subgoal is posted and steps in different plans are not interleaved. In some cases, conflicts can only be resolved at action level. For example, consider an agent which has two plans P_1 and P_2. Plan P_1 consists of two actions: A_1 and A_2. Action A_1 has a precondition p and a postcondition r. Action A_2 has a precondition r and a postcondition $\neg q$. Plan P_2 also consists of two actions: A_3 and A_4. Action A_3 has a precondition q and a postcondition t. Action A_4 has a precondition t and a postcondition $\neg p$. In this situation,

the only way to successfully execute both plans is to interleave the actions in these plans, e.g., execute action A_1, then action A_3, then action A_2, and finally action A_4. Interleaving the execution of an agent's intention at the plan level also reduces the possibility of positive interactions (synergies), i.e., in cases where the plans cannot be merged, it may be possible to merge the actions in these plans. If intentions are merged at the plan level, then the agent cannot merge actions which have the same postcondition. In addition, their approaches have only been evaluated in a static environment, i.e., the environment only changes as a result of the agent's actions. Actions which have several possible effects are not considered. Waters et al. [66, 67] present a *coverage-based* approach to intention selection proposed by [59], in which the intention with the lowest coverage, i.e., the highest probability of becoming non-executable due to changes in the environment, is selected for execution. Unlike Thangarajah et al. and Shaw et al., they evaluate their approach in a dynamic environment; however intention selection is again limited to the plan level, and their experimental evaluation assumes that there are no conflicts between intentions.

Summary of contributions In this thesis, we provide a domain-independent approach to scheduling the progression of an agent's intentions. We use goal-plan trees, an abstract structure for representing the relationship between goals and the plans that can be used to achieve them, proposed by Thangarajah et al. [60, 57, 58, 56] to represent an agent's current intentions. Our approach is able to avoid conflicts, as well as to exploit synergies between an agent's intentions. In addition, our approach is also capable of scheduling with real-time constructs such as deadlines, and actions which may not achieve their

desired effects. The key idea of our approach is to use random sampling to estimate the utility of progressing each intentions, and to progress the intention which is most "promising". The most promising intention can be, for example the intention which has the largest utility value. Compared to existing approaches, our approach has several advantages. First, our approach is generic and flexible, and can be used to solve a wide range of problems. For different problems, the programmers don't need to rewrite the whole strategy like in most agent programming languages (e.g., S_I function in Jason). Instead, one can simply customise their requirements of the agent by customising the utility function in our approach. Second, our approach is able to schedule the progression of the agent's intentions at the level of individual actions. Problems which cannot be solved at the plan level may have a solution at the action level. Third, our approach is able to schedule actions which have several possible postconditions, while most of the existing approaches assume plans and actions always achieve their desired effects (postcondition). Finally, our approach is an anytime algorithm which means it can return a result at any moment in time.

We evaluate the performance of our approach in both synthetic domains and more realistic "real-world" domains. The synthetic domains involve scheduling a set of goal-plan trees which is randomly generated. We develop a way to generate sets of synthetic goal-plan trees, and use these synthetic goal-plan trees to evaluate the performance of our approach. The results show that our approach outperforms existing approaches to scheduling the progression of agent's intentions in both practical agent programming languages (e.g., Round-

Robin, First-In-First-Out scheduling), and in the theoretical literature (e.g., Summary-Information-based approach, Coverage-based approaches).

Outline of thesis The rest of the thesis is organised as follows. In Chapter 2, we give the general definition and characteristics of agent programming, followed by a detailed introduction to the Belief-Desire-Intention (BDI) agent architecture. This chapter also includes a brief survey of existing BDI agent programming languages and a detailed survey of previous approaches to reasoning about interactions between an agent's intentions.

In Chapter 3, we present the representations of goals, plans and actions and define an extended version of goal-plan tree [60, 57, 58, 56] which includes an explicit representation of actions. Goals, plans and actions are also extended to contain real-time construct including deadlines, preferred achievement time and estimated execution time, and we define fallible actions which have more than one possible postcondition. We end this chapter by precisely defining the intention progression problem, in which the intentions are represented using a set of goal-plan trees with their corresponding current step pointer.

In Chapter 4, we introduce Monte-Carlo Tree Search (MCTS) and its variant Single-Player Monte-Carlo Tree Search (SP-MCTS), and discuss the MCTS and SP-MCTS algorithm in detail. We then present one of the main contributions of the thesis: the *SAM* scheduler which uses (SP-)MCTS to schedule an agent's intentions.

In Chapter 5, we show how the *SAM* scheduler can be used to avoid conflicts between an agent's intentions. We evaluate the performance of our approach in both synthetic domains and a more realistic scenario based on the Miconic-10 elevator domain [40]. The synthetic domains involve scheduling

a set of synthetic goal-plan trees which was generated by a package called GenGPT. By controlling the input parameters of GenGPT, we can generate goal-plan trees of different shapes and properties. The experimental results suggest the *SAM* scheduler outperforms existing approaches to intention progression both in terms of goals achieved and the variance in goal achievement time.

To demonstrate the flexibility of our approach, *SAM* is used to schedule intentions which have deadlines and preferred achievement times in Chapter 6. Again, we evaluate the performance of *SAM* in both synthetic domains and a more realistic domain based on the classic Blocks World domain. The experimental results suggest that *SAM* outperforms a state-of-the-art approach to intention progression with deadlines, AgentSpeak(RT) [63, 62], both in terms of the number of goals achieved and the total tardiness in achieving these goals.

In Chapter 7, we consider positive interactions and use the *SAM* scheduler to exploit the synergies between agent's intentions. In the first part of this chapter, we focus on scheduling fallible actions (actions which have more than one possible postcondition), where synergies between intentions can be used to recover from execution failure. In the second part of this chapter, we focus on merging actions which have same postconditions to save execution steps. We evaluate the performance of *SAM* in both synthetic domains and more realistic domains based on the Miconic-10 elevator domain and the classic Blocks World domain. The experimental results indicate that the potential benefits of using *SAM* to exploit synergies between intentions is quite significant.

In Chapter 8, we discuss the computational overhead of *SAM*. We evaluate the performance of *SAM* with different computational budgets. The results suggest the *SAM* can achieve all given goals even when the computational budget is very small (i.e., 50 iterations and 5 simulations in each iteration), and only requires 9ms time to return a full schedule. If additional time is given, *SAM* can improve its performance by running additional simulations.

Finally, conclusions and possible future research areas of our work are presented in Chapter 9.

Chapter 2

Background

In this chapter, we give the general definition and characteristics of agent programming, followed by a detailed introduction to the Belief-Desire-Intention (BDI) agent architecture. Finally, at the end of this chapter, we give a brief survey of the existing BDI agent programming languages, and a detailed survey of previous approaches to reasoning about interactions between an agent's intentions.

2.1 Introduction to Agents

In Artificial Intelligence (AI) and Computer Science, the term "agent" is widely used but has no widely accepted definitions. In this thesis, we follow Woodridge and Jennings's definition of agents, where an agent is a computer system situated in some environment, and that is capable of autonomous actions in order to meet its design objectives [73]. The environment where an agent is situated can be the real world, or it may be a purely computational environment within computer systems. In both situations, the agent is able to

observe the environment via its sensors, and performs certain external actions using its actuators to change the state of the environment (see Figure 2.1), e.g., a robot vacuum cleaner could detect walls and obstacles via its sensors, and moves using its wheels.

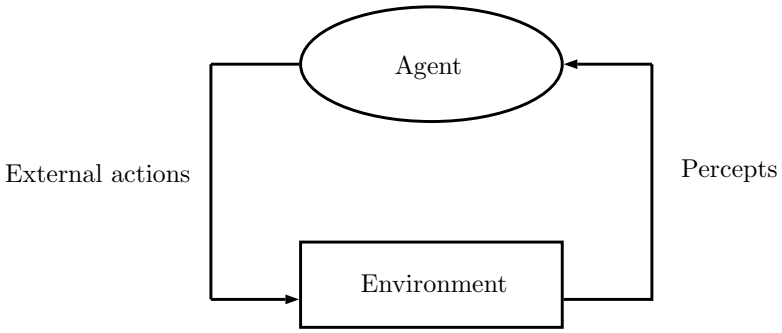


Fig. 2.1 Agent and environment.

In addition, Woodridge and Jennings also argued that an agent should have the following properties [73]:

- *autonomy*
- *social ability*
- *reactivity*
- *pro-activeness*

Autonomy means an agent has the ability to operate without the direct intervention of humans or other agents, and is able to determine which objectives to achieve next and how to achieve them. For example, unmanned aerial vehicles (UAVs) which are controlled by onboard computers are fully or intermittently autonomous. This kind of UAV is capable of autonomous actions and to accomplish its strategic objectives with minimal supervisory assistance. In contrast, computer programs such as text editors or spreadsheets

require human or other programs' command to work and have very little or no autonomy, i.e., they won't work unless you click on an icon or select an option in the menu.

Social ability means agents are able to interact with other agents or humans via some kind of communication languages or interface. It allows agents to exchange information, and to cooperate and coordinate activities with other agents, in order to accomplish their objectives.

Reactivity means agents have the ability to respond to the changes in the environment in a timely fashion. For example, a robot vacuum cleaner stops moving ahead if there is a wall in front of it. Being reactive is useful when plans (the ways to achieve an objective) or actions go wrong. Agents will respond to the failures and choose to execute an alternative plan.

Finally, pro-activeness means agents are able to exhibit goal-directed behaviour. If an agent has been given a particular objective, then we expect the agent to try to achieve this objective at any given opportunity. This property rules out entirely passive agents who never try to do anything.

2.2 The Belief-Desire-Intention Agent Architecture

In the previous section, we discussed *agent theories* [72, 73] , i.e., the definition and characteristics of an agent. In this section, we shift the emphasis from theory to practice, and focus on *agent architectures* which can be thought of as software engineering models of agents [72, 73]. To realise the properties of agents mentioned in the previous section, numerous agent architectures have been proposed including deliberative agent architectures, reactive agent

architectures and hybrid agent architectures [72, 73]. In this thesis, we focus on the Belief-Desire-Intention (BDI) agent architecture which is one of the most commonly used architectures for designing agents. Agents that are built using the BDI agent architecture are referred to as *BDI agents*. The origins of the BDI agent architecture lie in the philosophical theory of practical reasoning introduced by Bratman [10] (which we discuss later). The behaviour of BDI agents is described by using their mental attitudes including beliefs, desires and intentions.

Beliefs represent what the agent believes about the world or the environment, and usually includes a set of ground facts and some inference rules. For example, an agent may believe there is a wall in front of it. The agent's beliefs are not necessarily true, but are believed to be true by the agent, e.g., an agent may believe it will rain in an hour, which might be wrong. An agent's beliefs can change over time by *belief revision*. Katsuno et al. [38] identified two parts to belief revision: *update* and *revision*. Revision occurs when an agent realises that its beliefs are incorrect or incomplete compared to the real world. While, update is triggered by actual changes in the world which forces the agent to change its beliefs.

Desires are descriptions of states of affairs the agents want to bring about. Desires may be inconsistent, i.e., the agent may not be able to achieve all its desires. For example, the desires to arrive at the airport before 9:00 am and to stay in the office are inconsistent, as the agent cannot achieve (or satisfy) both of them. *Goals* are defined as the set of desires the agent decides to pursue. There are two different types of goals. One type is called *Achievement goals*,

and the other are called *Maintenance goals*. An achievement goal represents a state of the environment the agent wants to achieve [22] (e.g., arriving the airport before 9:00 am in the previous example). Once an achievement goal is achieved, the goal itself is dropped by the agent (with success). For example, if the agent arrived the airport at 8:30 am, then the goal to arrive the airport before 9:00 am should be dropped.

In contrast, a maintenance goal represents a particular state that must remain true [22] (i.e. a maintenance goal is initially true, and if it becomes false, the agent will make it true again). An example of a maintenance goal is to stay in the office (in our previous example). In this case, the agent shouldn't leave the office before 10:00 am. In this thesis, we focus on *achievement goals*, which represent the common natural understanding of the word "goal" [22].

Intentions are instantiated plans that the agent has committed to in order to achieve its goals. Each plan is composed of steps which are either *actions* that directly change the agent's environment or *subgoals* which are in turn achieved by other plans. When there are multiple plans available to achieve a goal, the agent has to choose which plan should be used to achieve this goal. For example, an agent could walk to the city centre or go to the city centre by bus.

In the BDI architecture, the model of decision making is termed *practical reasoning* [74]. While theoretical reasoning is directed towards an agent's beliefs, practical reasoning is reasoning towards the agent's actions. For example, if we believe all birds have wings, and a woodpecker is a bird, then we could conclude that the woodpecker has wings. The process of concluding that the woodpecker has wings is called theoretical reasoning which only

affects our beliefs about the world. However, the process of deciding whether to go to the airport by bus or by train is practical reasoning, as it is reasoning towards actions. In [11], Bratman argues practical reasoning can be thought of as the act of weighing multiple, conflicting considerations for and against conflicting choices, where the relevant considerations are provided by what the agent believes, desires, values and cares about. More precisely, practical reasoning consists of two distinct steps which are *deliberation* and *means-ends reasoning* [74]. The first step (deliberation) decides what state of affairs the agent wants to achieve, and the second step (means-ends reasoning) decides how the agent achieves this state of affairs. In the BDI architecture, the desires of an agent represents states of affairs the agents want to bring about, and the state of affairs that the agent commits to are intentions. Therefore, the process of deliberation involves deciding which goal to adopt (as intentions), and the process of means-ends reasoning involves deciding which means (plans) to use to achieve an adopted end. For example, consider a person who has a desire to go on a picnic, and another desire to go to the library. After deliberation, he may choose to go to the library as it is raining today. The next step is to decide how to achieve his selected desires (goals). He may go to the library by bus or by foot. The process of deciding whether to go to the library by bus or by foot is therefore the means-ends reasoning.

2.2.1 The Abstract BDI Interpreter

To build practical BDI implementations, Rao and Georgeff [47] define an abstract BDI interpreter which is used as the basis of the execution and deliber-

ation model for many BDI-based agent programming languages. The abstract BDI interpreter is shown in Algorithm 1.

Algorithm 1 BDI-Interpreter

```
1: Initialize-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);
```

The data structures B, G, I represent the mental state of the agent which are beliefs (B), goals (G) and intentions (I), and event-queue is an input queue of events. An event is a trigger to which the agent must respond. For example, a cleaning robot may generate a goal to clean a room if it detects garbage in the room. An event may be external or internal. An external event is generated outside of the agent (in the environment) and perceived by sensors. An internal event is generated by the agent program itself. The events are assumed to be atomic and are recognised after they have occurred. At the beginning of each cycle (which is also called agent's deliberation cycle), the agent examines the event-queue, and generates a set of *options* (plans) that the agent can pursue to achieve these events (line 3). The options represent the possible future execution (plans) in response to each event. The agent then deliberates on these options (whether to adopt these options) and updates its intentions (line 4-5), i.e., adds a subset of the options to its intentions. The agent then executes an action in one of its intentions (line 6). The action may be an internal action that manipulates agent's mental state or an external action that directly change the state of the environment. After that, any external events

that occurred during the cycle are added to the event-queue (line 7). The agent then updates its intentions by dropping all successful intentions (the goals which are achieved) (line 8), as well as the intentions that are impossible to achieve (failed) (line 9).

The abstract interpreter above provides a simple execution and deliberation model for building BDI-based agent systems. However, it cannot be directly used to build agent-based systems, as it leaves out certain details, e.g., how the set of intentions are executed. Consequently, a variety of BDI agent programming languages have been proposed based on the abstract interpreter. These languages provide data structures to encode beliefs, goals and intentions and specify the details of the activities such as generating new intentions, selecting plans to achieve a goal, and selecting an intention and progressing their interpreter. For example, there are several different ways to execute an agent's intention, including Round-Robin scheduling and First-In-First-Out scheduling (see Section 5.3). These execution strategies can also interact with other key features of practical BDI languages, such as failure handling (e.g., *RR*+ in Section 7.1.3), to improve the performance of agents. In the next section, we give a brief overview of well-known BDI agent programming languages which are implemented based on the abstract interpreter, and show how the intentions are selected and progressed in each of them.

2.3 BDI-based Agent Programming Languages and Platforms

AgentSpeak(L) Rao [45] proposed AgentSpeak(L), an abstract agent programming language based on a restricted first-order language with events and actions. An AgentSpeak(L) agent consists of a *belief base*, an *event base*, a *plan base* and an *intention base*. The belief base represents what the agent believes, and is encoded as a set of ground belief literals. The event base consists of a set of triggering events that occur during the execution of the agent. There are two types of triggering events. The first type relates to the addition of beliefs or goals and is denoted by $+e$, where e is a belief or a goal. The second type relates to the deletion of beliefs or goals, and is denoted by $-e$, where e is a belief or a goal. For example, $+!g$ is a triggering event which states that an achievement goal $!g$ is added. AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement goals are of the form $!g$, while test goals are $?g$, where g is a literal (as for beliefs). An achievement goal $!g$ represents that the agent wants to achieve a state where g is true. A test goal $?g$ states that the agent want to test whether the formula g is a belief. The plan base is a set of pre-defined plans which are used to achieve the goals. Each plan is of the form $+!g : \Phi \leftarrow P_1, \ldots, P_n$, where $+!g$ is a triggering event, Φ is a context-condition which must be true for the plan to begin execution. $P_1, \ldots, P_n$ is the body of the plan, which is a sequence of external actions, subgoals, test goals and belief updates. Finally, the intention base consists of a set of intentions that the agent committed to. Each intention is a stack of partially instantiated plans.

The operational semantics of AgentSpeak(L) is defined by a set of transition rules which describes how the agent generates intentions from its plan base and how these intentions are executed. Informally, at every deliberation cycle of an agent program, AgentSpeak(L) updates a list of events, which may be generated from perception of the environment (external), or from the execution of intentions (internal). The agent then selects one event from its event base. A plan which has the selected event as its triggering event and has its context condition holds in the current environment is then selected and fired from the plan base. If the selected event is an external event, the selected plan is used to create a new intention, which is then added to the intention base. If the selected event is an internal event, the selected plan is pushed on top of an existing intention which triggered the internal event. Finally, the agent randomly selects one of its intentions to execute. When an agent executes an intention, it executes the next step of the intention (i.e., the first goal or action of the top of the intention). If the next step in the selected intention is an external action, then the agent will perform the action in the environment. If the next step is a belief update action, then the agent will add/delete the beliefs to/from its belief base. If the next step is an achievement goal, an internal event (i.e., addition of the goal) will be generated and added to the event base. If the next step is a test goal, then the formula in the test goal will be evaluated against the belief base. The agent continues its deliberation cycle until there is no more events in the event base or there is no runnable intentions. The original operational semantics of AgentSpeak(L) has several limitations, e.g., it cannot handle execution failures, and is not able to avoid conflicts or to exploit synergies between intentions. Due to the omissions and limitations

in the semantics of AgentSpeak(L), several extensions of the languages were proposed.

Jason Jason [9] is a Java-based platform for developing multi-agent systems and an interpreter for an extension of the agent programming language AgentSpeak(L) [45].

The Jason interpreter (its deliberation cycle) is divided into 11 phases which are as follows.

1. Perceiving the Environment: In this phase, the `perceive` method is used to obtain percepts from the environment.

2. Updating the Belief Base: In this phase, the sensed percepts are then used to update agent's belief base. This is done by a *belief update function* which is implemented by the customisable method `buf`. In the default `buf` method everything in the current percepts are used to update the agent's belief base.

3. Receiving Communication from Other Agents: In this phase, the `checkMail` method is used to receive messages from other agents.

4. Selecting "Socially Acceptable" Messages: In this phase, the received messages are then selected by a method called `SocAcc` which implements the *social acceptance function*.

5. Select an Event: In this phase, a pending event is selected to progress from the event base by an agent-specific *event selection function* S_E. The default event selection function assumes all events have the same

priority (or importance), and simply selects the first event in the event list.

6. Retrieving all Relevant Plans: In this phase, all the relevant plans which have the selected event as their triggering event, are found in the plan base.

7. Determining the Applicable Plans: In this phase, all the relevant plans are checked to see if they are applicable with the current belief base. A relevant plan is applicable if its context condition holds in the current belief base of the agent.

8. Selecting One Applicable Plan: In this phase, one of the applicable plans is selected and added into the set of agent's intentions. The selection process is determined by an *applicable plan selection function* S_O. The default applicable plan selection function chooses an applicable plan based on the order in which they appear in the plan library.

9. Selecting an Intention for Further Execution: In this phase, one of the agent's intentions is then selected in this phase for execution. This is done by using an *intention selection function* S_I. The default intention selection function selects each of the intentions in turn, and when executed, only one step in the intention is executed. This is known as the "Round-Robin" scheduling. Round-Robin scheduling (RR) attempts to ensure 'fairness' between intentions by executing a fixed number of steps of each intention in turn. For example, the agent may iterate over its intentions, executing one step of the next intention in sequence at each deliberation cycle.

10. Executing One Step of an Intention: In this phase, one step of the selected intention is executed.

11. Before restarting the reasoning cycle, the set of intentions are updated, e.g., the finished intentions are removed from the set of intentions.

The Jason interpreter follows and extends the abstract BDI interpreter in [47]. It allows the programmer to implement their own strategies of practical reasoning by customising the event selection function S_E, the applicable plan selection function S_O and the intention selection function S_I. In addition, Jason also provides a programming construct called "atomic". If an instance of a plan is atomic and is selected by the intention selection function, then this intention will be selected for execution continuously until it is finished. The atomic constructs are used to avoid conflicts between intentions

As with any software system, a BDI agent program may fail. Another extension of Jason compared to AgentSpeak(L) is the ability to handle execution failure. When a plan fails to achieve a goal, the Jason interpreter generates a goal deletion event ($-!g$ indicating the plan for a corresponding achievement goal $+!g$ has failed). The plan for a goal deletion event (goal deletion plan) is used to define what should be done when a goal fails. One thing programmers may want to do in the goal deletion plan is to retry the failed goal event, i.e., to attempt to achieve the failed goal by using alternative plans. In some cases, it may be also necessary to "clean up" the effects of certain actions that are executed before the plan failed.

AgentSpeak(XL) In [8], Bordini et al. proposed an agent programming language called AgentSpeak(XL), which is an extension of AgentSpeak(L). The

basic idea is to use the TÆMS (Task Analysis, Environment Modelling, and Simulation) framework [28] together with Design-To-Criteria (DTC) scheduling [64] to schedule intentions for BDI agents. The TÆMS framework is used to analyse and quantitatively simulate the behaviour of single or multi-agent systems with respect to several objectives all of which must be achieved to some extent. The key representation in TÆMS is the local and nonlocal sets of activities which are called task structures. The task structures capture several important pieces of information including:

- The top-level goals/abstract-tasks that an agent intends to achieve.
- One or more possible ways (plans) to achieve the top-level goal, expressed as an abstraction hierarchy whose leaves are basic action instantiations, called methods.
- A precise, quantitative definition of the degree of achievement in terms of measurable characteristics such as solution quality, cost, and duration.
- Task relationships that indicate how basic actions or abstract task achievement affect task characteristics elsewhere in the task structure.

The quality of a task group (a set of top-level goals) depends on which subtasks are executed and when they are executed. For example, a task T may enable a method M in the sense that the quality of M cannot be accrued until T is completed; i.e., the earliest start time of M is the finish time of T. This relationship is called enables, and is a hard relationship in the TÆMS framework. In contrast, they also define soft relationships, e.g., when a task T_1 facilitates another task T_2, the quality of T_2 is affected by T_1, however the soft relation (i.e., facilitate relation) may not necessarily be observed.

DTC is used to reason about the quality, cost, and duration of interrelated methods, and constructs a set of satisfying schedules for achieving high-level goals. DTC decides which tasks to perform, how to perform them, and the order in which they should be performed, so as to satisfy hard constraints (e.g., (soft) deadlines) and maximise the agent's objective function.

Bordini et al. [8] incorporate the DTC scheduler into AgentSpeak(XL), and show the major changes to the AgentSpeak(L) language and its abstract interpreter. The extended language AgentSpeak(XL) is compared with AgentSpeak(L) using a factory plant robot scenario, and the results suggest that the AgentSpeak(XL) robot outperforms the AgentSpeak(L) robot. However, one limitation of their approach is they didn't consider the plan selection problem in their approach, and when there are multiple applicable plans, the first applicable plan will always be chosen to achieve the goal. For problems where the selection of plans will affect the performance of the agent, their approach may not work.

AgentSpeak(RT) AgentSpeak(RT) is a programming language for real-time BDI agents based on AgentSpeak(L)[63, 62]. It extends AgentSpeak(L) with deadlines and priorities. In AgentSpeak(RT), top-level goals may optionally specify either a hard or soft deadline, and/or a priority. A *hard deadline* specifies the time by which a goal must be achieved; achieving the goal after the deadline has no value. Goals which are not achieved by their (hard) deadline are dropped. For example, a paper must be submitted before the conference submission deadline, otherwise the goal to submit a paper to this conference should be dropped. On the other hand, a *soft deadline* specifies the time by which a goal should ideally be achieved; however achieving the

goal after the preferred achievement time still has value. For example, the agent may have a goal to visit the gym this week; however if it's not possible to go this week, going next week (or later) is acceptable. In addition to hard deadline and soft deadline, a goal also has a *priority* which specifies the relative importance of achieving the goal. Given the estimated execution time of plans, AgentSpeak(RT) schedules intentions so as to achieve a *priority-maximal* set of intentions by their deadlines with a specified level of confidence. Vikhorev et al. [63, 62] defined a set of intentions is priority-maximal if no more intentions can be added to the schedule such that the scheduled intentions are to remain feasible (executable by their hard deadline) at the specified confidence level, and intentions which are dropped are incompatible with some scheduled higher priority intention(s).

The syntax and operational semantics of AgentSpeak(RT) is based on AgentSpeak(L). The AgentSpeak(RT) architecture consists of five components which are a belief base, an event base, a plan library, an intention structure and an interpreter. The belief base contains beliefs which are represented as conjunctions of first order ground terms, e.g., at(agent, location1). Events in AgentSpeak(RT) are categorised as internal and external events. Internal events are caused by the execution of the agent's program, and external events come from the environment or other agents. To allow the specification of real-time tasks, external events may optionally specify its hard or soft deadline, and/or a priority. Plans in AgentSpeak(RT) specify the sequence of actions and subgoals an agent can perform in order to achieve a particular goal. A plan consists of its name, a triggering event, a context condition and its plan body. In order to reason about real-time, a plan also has an execution time profile

which specifies the estimated execution time a plan may take. Vikhorev et al. use the execution time profile to represent the execution time of a plan, as the execution time of a plan will typically be influenced by the characteristics of the environment in which the agent will operate. Execution time profiles can be derived from an analysis of the agent's actions, plans and environment, or using automated techniques, e.g., stochastic simulation. Similar to AgentSpeak(L), the intention structure in AgentSpeak(RT) contains plans that have been chosen to achieve the top-level goals. Each intention consists of a stack of partially executed plans, and its deadlines and priority.

The interpreter manipulates agent's belief base, events and intention structure, deliberates about which plan to select in response to belief and goal change events, and schedules and executes intentions [63, 62]. The agent's state is a tuple $\langle$B, E, I$\rangle$, where B is the belief base, E is a set of events, and I is an ordered set of intentions. In [63], the execution of the interpreter is formalised as a sequence of functions, which compute an agent's new state and execute actions based on the agent's current state. A function *evt* is used to generate a set of events based on the agent's percepts and to update the agent's belief base.[1] The second function opt takes an agent's current state as its input, and generates a set of applicable plans for each event in the event set. When there are multiple applicable plans for an event, a selection function S_O is used to choose one of these plans for the event (as in Jason). By default, it selects the first applicable plan. The third function *sched* is used to add the plans to the intention set and schedule the execution of them in deadline order using a scheduling algorithm. The algorithm schedules the execution of a set of

[1] Unlike Jason, AgentSpeak(RT) processes all events at each deliberation cycle, rather than one.

intentions considering their deadline and priority, and returns a set of intentions which is "maximally feasible". In a set of "maximally feasible" intentions, all intentions are serialised to avoid conflicts, and no more intentions can be added if the scheduled intentions are to remain feasible. Intentions which are incompatible with some scheduled higher priority intentions are dropped. Scheduling in AgentSpeak(RT) is pre-emptive such that the adoption of a new high-priority intention may prevent previously scheduled intentions with a lower priority being added to the new schedule. Finally, function *exec* is used to execute the first intention in the ordered set of intentions.

One limitation of AgentSpeak(RT) is that it has very limited ability to resolve conflicts between intentions. To avoid conflicts between intentions, AgentSpeak(RT), like Jason, uses *atomic plans* that cannot be interleaved with other atomic plans. A non-atomic plan π cannot have atomic sub-plans, i.e., none of the plans to achieve the subgoals in this plan is atomic. As a result, intentions in AgentSpeak(RT) are either atomic or non-atomic; if the root plan of an intention is atomic, the intention is atomic, otherwise the intention is non-atomic. (In [3], this restriction is shown to be necessary for the scheduling approach adopted by AgentSpeak(RT) to be tractable.) Intentions which do not contain an atomic plan (non-atomic intentions) are executed concurrently; the execution of intentions containing one or more atomic plans (atomic intentions) are serialised. However, in some cases, the assumption on atomic plans is too strong. It forces the agent to drop intentions which are feasible (i.e., can be achieved if they are interleaved).

3APL 3APL (An Abstract Agent Programming Language) [35, 25, 26] is a rule-based programming language for implementing cognitive agents that have

beliefs, goals, and plans as mental attitudes. In the first version of 3APL [35], goals are procedural, where a goal is a set of procedures P which are executed to achieve the goal. Inspired by Dribble [61], Dastani et al. [25] extended 3APL to include the declarative aspect of goals. In [25], reflective features are included to allow programmers to specify how agents should revise their goals and plans. In [26], Dastani et al. removed some features of 3APL (such as goal revision) and defined the deliberation cycle of 3APL (see Figure 2.2)

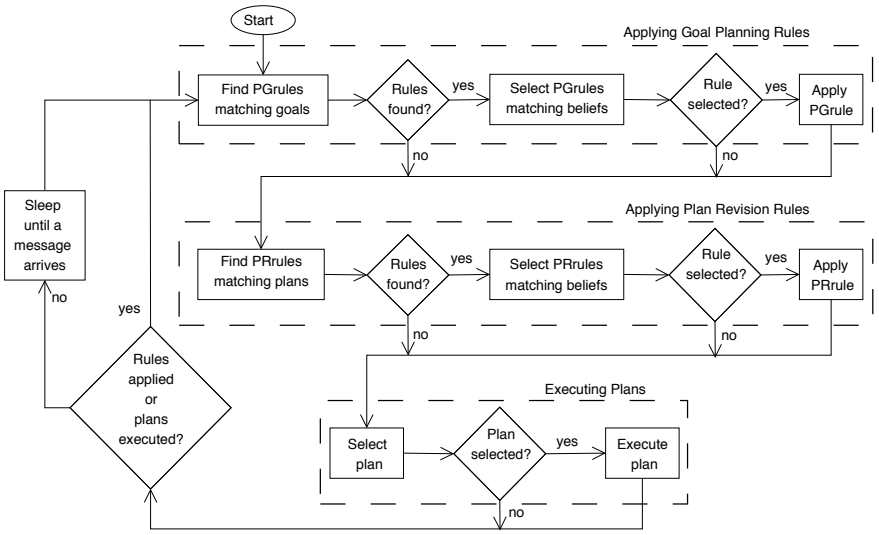


Fig. 2.2 Deliberation cycle in 3APL [26]

In what follows, we focus on the latest versio of 3APL as defined in [26]. A 3APL agent program consists of a capability base, a belief base, a goal base, a plan base, a goal planning rule base, a plan revision rule base, and the deliberation module. The capability base of an agent implements the mental actions that the agent can perform to update its beliefs. The belief base of an agent contains the set of information the agent believes about the world and itself, which is represented as a finite set of ground first-order atomic formulae and Horn clauses. The goal base denotes the desired state of the environment

the agent wants to achieve (which is represented as a finite set of first-order atomic formulas or conjunctions of formulas), and the plan base contains the plans the agent intends to perform which are sequences of external actions, belief update actions, abstract plans and test actions. In order to reason with goals and plans, 3APL provide two types of practical reasoning rules which are *goal planning rules (PG rules)* and *plan revision rules (PR rules)*. The goal planning rules are used to generate plans to achieve an agent's goals, and are of the form $\alpha \leftarrow \beta \mid \pi$, where α is a first order atomic formula representing a goal the agent wants to achieve, β is a query on agent's belief base indicating if a plan π can be applied. A plan π is built from basic actions that can be composed through program operators. Five types of basic actions are distinguished in [26], including mental actions (updating the belief base of an agent), communication actions (sending a message to another agent), external actions (changing the external environment), test actions (checking if a well-formed formula is derivable from the belief base) and abstract actions (abstract representation of a plan which can be instantiated with a more concrete plan during execution). There are three program operators defined in 3APL [26], including the sequential operator (;), the iteration operator (*while-do*), and the conditional choice operator (*if-then-else*)

The following is an example of goal planning rule in 3APL.

```
on(X, Y) ← on(X, Z) | Move(X, Z, Y)
```

This rule is used to have block X on the top of block Y. However, in order to apply this rule, block X must be on the top of block Z. If this rule is applied, a plan which has an action to move block X from block Z onto block Y is generated.

PR rules are of the form $\pi_a \leftarrow \beta \mid \pi_b$, and are used to revise the plans from the plan base. In a plan revision rule, π_a is a prefix of some plan, β is a query on the agent's belief base, and π_b is the revised plan. The following is an example of plan revision rule in 3APL.

`Move(X, Z, Y)` $\leftarrow$ `on(W, X) | Move(W, X, table) ; Move(X, Z, Y)`

This plan revision rule is used if the agent cannot move block X from block Z onto block Y as (the agent believes) there is a block W on the top of block X. If this rule is applied, a revised plan which first moves block W to the table, then moves block X from block Z onto block Y is generated.

The deliberation module is used to implement an agent's deliberation process. The deliberation cycle of 3APL in Figure 2.2 consist of 4 main steps.

Step 1: The agent receives the perception from the environment and updates its belief base and goal base.

Step 2: The agent then applies a goal planning rule to one of its goals, and adds the resulting plan to the plan base.

Step 3: A plan revision rule is applied to revise one of its intentions.

Step 4: One of its intentions in the plan base is selected for execution.

There are several decisions to be made in an agent's deliberation process including which goal planning rule should be applied in step 2, which plan revision rule should be used in step 3 and which intention should be executed in step 4. Hindriks et al. introduced a meta-language on top of basic 3APL to make these decisions explicitly [35]. This meta-language was then extended

by Dastani et al. [24] to include planning, tests, and different types of selection functions, such that it can be used to select plans and rules in 3APL. Using this meta-language, one can program the deliberation process of an agent, e.g., by defining the ordering on goals or rules. By default, 3APL always selects the first applicable rule (or plan). Additionally, 3APL [26] also allows basic failure handling by specifying plan revision rules (called failure rule in [35]). Plan revision rules are used to repair the current plan so that the intended goal can be achieved.

2APL 2APL (A Practical Agent Programming Language) [23] extends 3APL for implementing multi-agent systems. It provides three types of practical reasoning rules which are *planning goal rules (PG-rules)*, *procedure call rules (PC-rules)* and *plan repair rules (PR-rules)*. Planning goal rules are the same as the goal planning rules of 3APL. They are used to generate plans when the agent has certain goals and beliefs. Procedure call rules can be used in various ways, including executing abstract actions (procedure calls in imperative programming languages), responding to messages and handling external events. Procedure call rules have a similar form to the goal planning rules except that the head of a procedure call rule is not a goal. Instead, the head of a procedure call rule can be an abstract action, a message, or an external event. Plan repair rules have the same form as plan revision rules in 3APL. However, unlike plan revision rules, plan repair rules can only be applied if a plan fails. By applying a plan repair rule, the failed plan is replaced by a new plan.

Figure 2.3 shows the deliberation cycle of a 2APL agent. Each step in the deliberation cycle is specified by a set of transition rules representing

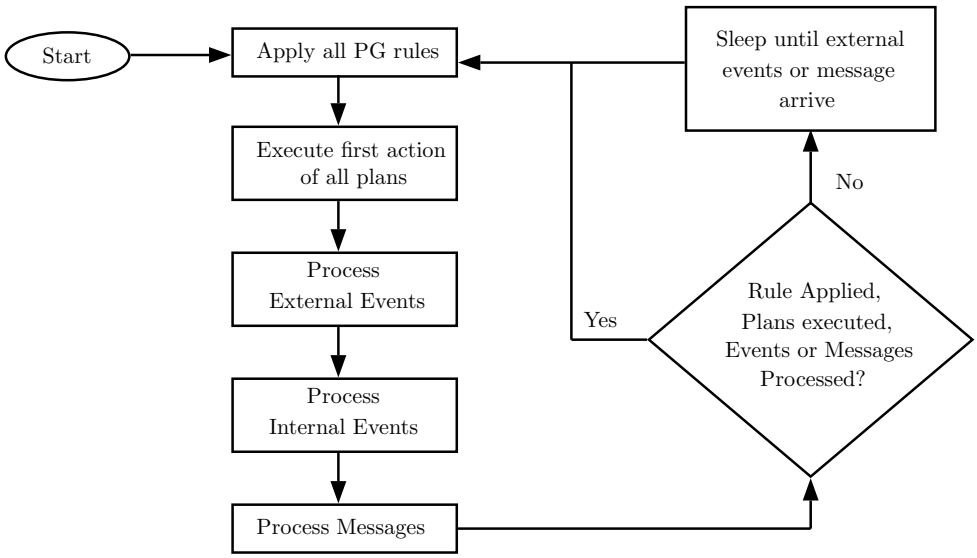


Fig. 2.3 Deliberation cycle in 2APL [23]

which transition a 2APL agent can make at each moment in time. At the beginning of each deliberation cycle, the 2APL agent applies all the applicable PG-rules (each rule only once) to generate plans to achieve its goals. The agent then executes the first action of all its intended plans. This is to make the deliberation process fair such that all the plans get a chance to be executed. The next steps are to process all external events (from the environment), and then all internal events indicating the failure of plans, and finally all the messages received from other agents.

External events, internal events and received messages are handled by applying the first applicable PC-rule, PR-rule, PC-rule to them respectively. After these steps, the agent sleeps if there is no rule could be applied, no plan could be executed, and no event could be processed. Otherwise, the agent will do a new cycle of these deliberation steps.

GOAL The Goal-Oriented Agent Language *GOAL* was proposed by Hindriks et al. [36] to support declarative goals which were not included in

AgentSpeak(L) [45] and the first version of 3APL [35]. A declarative goal describes a state that an agent wants to achieve, rather than a plan that needs to be performed. A GOAL agent selects actions to achieve its goals on the basis of its current mental state. A mental state is a pair $< \sigma, \gamma >$, where σ is a set of beliefs and γ is a set of goals. Both the beliefs σ and the goals γ are represented using a propositional language $\mathcal{L}$. The goals in γ must be consistent, and the mental state of an agent must satisfy that: for all $\phi \in \gamma$, $\sigma \not\models \phi$. That is, an agent cannot have a goal which the agent already believes. In addition to the mental state, a GOAL agent also has its *capabilities* (*Bcap*). The capabilities of an agent is a set of *basic actions* which are used to update the agent's beliefs σ. However, the capabilities do not change the goals of an agent directly. To modify an agent's goals, two additional actions *adopt* and *drop* are introduced, which are used to adopt a new goal and to drop a failed goal respectively. An agent's capabilities decide what the agent is able to do, but do not specify when an agent should perform these basic actions. A conditional action is a pair $\varphi \rightarrow do(\alpha)$, where $\alpha \in Bcap \cup \{adopt(\phi), drop(\phi) | \phi \in \mathcal{L}\}$, and φ is a mental state condition which specifies when the action α can be executed. That is, if the mental condition φ holds, then the agent may consider executing the basic action α. A Goal agent is represented by a triple $< \Pi, \sigma_0, \gamma_0 >$, where Π is a non-empty set of conditional actions, σ_0 is a set of initial beliefs, and γ_0 is a set of initial goals.

GOAL agents implement a blind commitment strategy [46] as their default commitment strategy, i.e., an agent drops a goal if and only if it believes the goal has been achieved. However, it is only a default strategy and programmers may overwrite the default strategy by means of the drop action.

JACK JACK Agent Language[68] is a BDI agent development platform for building intelligent agents as components of Java applications. A JACK agent consists of its plan library, event-goals, beliefs and a Java class to manipulate its resources. Plans in JACK are similar to the plans in Jason, where each plan has a context condition which determines its applicability and a plan body which contains subgoals. However, the actions in JACK are arbitrary Java code. The event-goals may be external or internal (those set by the agent itself), and the beliefs are represented using a database.

The general execution model of JACK to handle an event is shown in Figure 2.

Algorithm 2 Event handling in JACK

1: Event posted.
2: Determine the set of relevant plans.
3: Determine applicable plans
4: Select a plan and run its body
5: If plan fails, go to step 4

When a goal-event is posted, the agent retrieves all its the relevant plans from the plan library (each goal event will normally have a number of plans to handle it). For these relevant plans, some of them will be applicable. A relevant plan is applicable if its context condition is true. If there is no applicable plan, then the goal fails and a failure handling event is triggered. Otherwise, one of the applicable plans is selected for execution. In JACK, there are three different mechanisms for choosing plans. By default, the plan is chosen based on the order in which they appear in the plan library. Programmers can also assign a numerical priority to each of these plans and select the plan with highest priority, or use a meta-plan to decide which plan to select. The selected intention is then added to the set of agent's intentions. JACK provides

two mechanisms to execute its intentions. Programmers can choose to either execute its intentions in a Round-Robin fashion, which does a fixed number of steps on each intention in turn, or a First-In-First-Out (FIFO) queue, which processes each intention in the order received, moving it to the back of the queue if for some reason it fails.

When a plan fails and a failure handling event is triggered, an alternative plan will be used to handle it. If all plans have failed and the goal event is still not handled, the plan which posted this goal event fails (if this goal event is posted from a plan) and its triggering event is re-posted in an attempt to find an alternative plan. If the goal event is not posted from a plan and all its applicable plans failed to handle it, then the goal event is discarded.

CAN The Conceptual Agent Notation (CAN) BDI agent programming language was first introduced by Winikoff et al. [70] to provide a framework for goals which integrate both declarative and procedural views.

A CAN agent program $\prod$ consists of a collection of plan clauses $e : c_1 \leftarrow c_2 : P$ where e is an event, c_i are context conditions, and P is the plan body. Beliefs (b) are represented using first-order terms. Winikoff et al. assumed that a CAN agent has operations to check whether a condition holds in a belief set, to add a belief to a belief set, and to delete a belief from a belief set. The plan body P is described by the following grammar:

$$
\begin{aligned}
P \quad ::= \quad & true \mid fail \mid ex(X) \\
\mid \quad & act \mid +b \mid -b \mid ?c \mid !e \\
\mid \quad & P;P \mid P \| P \mid Goal(s,P,f) \\
\mid \quad & P \triangleright P \mid B : P, \ldots, B : P
\end{aligned}
$$

Where $ex(X)$ represents an exception, *act* is a primitive action, $+b$ and $-b$ are operations to add and delete beliefs, $?c$ is a test condition and $!e$ is an event. Winikoff et al. defined a new construct *Goal(s, P, f)*, which means achieving s using P; failing if f becomes true. Both s and f (i.e., the declarative aspects of a goal) are logical formulae over an agent's beliefs, and are required to be exclusive of each other such that no world state satisfies s and f at the same time. P is a set of guarded plans of the form $C_1 : P_1, \ldots, C_n : P_n$ (i.e., the procedural aspects of a goal), where each P_i is a plan to achieve s and C_i is its corresponding context-condition (which must be true for the plan P_i to begin execution). $P;P$, $P\|P$ and $Goal(s,P,f)$ are therefore compounds which represent sequencing, parallelism and a goal respectively. Winikoff et al. also defined the auxiliary compound $B : P, \ldots, B : P$ where an event B matches a set of guarded alternatives P, and $P \triangleright P$ which means executing the first plan and then the second only if the first plan fails. The detailed operational semantics can be found in [70]. Winikoff et al. also pointed out that different ways of pursuing multiple goals in parallel might cause potential conflicts and positive interactions. However, they didn't incorporate any effective approach of how to pursue multiple goals in parallel into CAN [70]. When there are more than one applicable plans to achieve a goal, the plan is selected non-deterministically.

In a later version of CAN, CAN^A [50], the plan selection rule *Sel* is implemented, which applies the first applicable plan of a goal and keeps the remaining non-chosen plans as "backup". Once an applicable plan is selected, it must be carried out to completion, if possible. If the selected plan is unable to execute further, the "backup" plans will be tried to achieve the goal. If all the "backup" plans fail to achieve the goal, then the failure will be propagated

to a higher level goal, and "backup" plans to achieve the higher level goal will be tried. In an another version of CAN, CANPlan [50], Hierarchical Task Network (HTN) planning is integrated into the CAN programming language to provide on-demand planning capabilities. The planning mechanism is used for intelligent plan selection (i.e., which plan should be applied to achieve a goal). However, in CANPlan the approaches to select which intentions to progress at each deliberation cycle are still not provided.

Meta-APL Meta-APL is a BDI agent programming language proposed by Doan et al. [30, 31] which allows both agent programs and their associated deliberation strategy to be encoded in the same programming language. Meta-APL combines both logical programming and imperative programming, i.e., an agents' mental states can be represented using a logical language, and the agent's intentions can be specified and executed in an imperative fashion.

A meta-APL agent consists of two parts: a mental state and a plan state. The mental state of an agent represents its beliefs about the environment and itself, the goals that the agent wants to achieve, and the events the agent needs to react to. The plan state of an agent is a set of plans which represents the agent's current intentions. Similar to the plans in Jason and Jack, and the goal planning rules in 3APL (and planning goal rules in 2APL), the plans in meta-APL are generated by applying *object-level rules*. An agent's mental state and plan state are manipulated by meta actions which are organised into meta rules. Each meta rule has a condition that indicates when the rule can be applied, and a sequence of meta actions which are used to update the mental state and the plan state and enabling procedural reflection in meta-APL.

Meta-APL allows object-level rules and meta rules to be grouped into a list of rule sets. Programmers can easily define the order in which rules are applied. For example, suppose R_1 are meta plans which are used to update an agent's beliefs, R_2 is a set of object-level rules for generating plans, and R_3 are meta rules to select intentions for execution. Programmers may write these rules in the following order: R_1, R_2, and then R_3. It means an agent will first update its beliefs by applying R_1, then apply object-level rules in R_2 to generate plans to achieve its goals, and finally select intentions for progressing according to the meta rules R_3.

The core deliberation cycle of meta-APL consists of three phases. In the first phase, the agent updates its mental state with its perceptions of the environment and the messages received from other agents. In the second phase, the sets of rules (both objective-level rules and meta rules) comprising an agent program are processed sequentially. Programmers can customise the order of the rule sets and the set of meta-plans to customise the strategy of progressing intentions, i.e., which objective-level rule should be applied first, and which intentions should progressed in the current deliberation cycle.

Summary This section gives a brief overview of the well-known BDI agent programming languages. AgentSpeak(L), Jason, JACK, 3APL and 2APL allow programmers to customise their own strategy to progress intentions (e.g., the intention selection function S_I in Jason), and use very simple approaches (e.g., First-In-First-Out or Round-Robin) as their default strategies to progress intentions. When an agent has multiple applicable plans to achieve a goal, the plan is either chosen nondeterministically or in the order in which they appear in the plan library. These approaches in general do not have abilities

to avoid conflicts or exploit synergies between intentions. Meta-APL has no default strategy to progress intentions, but also allows programmers to customise their own strategy to progress intentions. AgentSpeak(RT) extends AgentSpeak(L) with deadlines and priorities, and schedules intentions so as to achieve a priority-maximal set of intentions by their deadlines with a specified level of confidence. However, AgentSpeak(RT) has very limited ability to resolve conflicts between intentions, and has no ability to exploit synergies between intentions. AgentSpeak(XL) uses the TÆMS (Task Analysis, Environment Modelling, and Simulation) framework together with Design-To-Criteria scheduling to schedule intentions in BDI agent. The TÆMS framework is used to analyse and quantitatively simulate the behaviour of single or multi-agent systems with respect to several objectives all of which must be achieved to some extent, and the DTC is used to reason about the quality, cost, and duration of interrelated methods, and constructs a set of satisfying schedules for achieving high-level goals. However, AgentSpeak(XL) has a very simple approach to select plans. When there are multiple applicable plans, the first applicable plan in the plan library will always be chosen to achieve the goal. Moreover, the approach to schedule intentions in AgentSpeak(XL) also doesn't consider the problem of exploiting synergies between intentions. The *GOAL* agent programming language was proposed to support declarative goals. It provides operational semantics for selecting and executing actions and assume the selection of actions is weakly fair. However, it also does not have abilities to avoid conflicts or exploit synergies between intentions. The *CAN* agent programming language provides a framework for goals which integrates both declarative and procedural views. In a later version of CAN, CANPlan, the

HTN planner is used to decide which plan should be used to achieve a goal. However, in CANPlan the approaches to select which intentions to progress at each deliberation cycle are still not provided.

In state of the art BDI agent programming languages, there is no general approach that can avoid conflicts and exploit synergies between intentions in all problem domains. In the next section, we will explore the approaches proposed to reason about the interactions between intentions of BDI agents in the literature.

2.4 Approaches to Reasoning About Interactions Between Intentions

In this section, we introduce the notion of *goal-plan trees* which have been used to represent the relationship between goals and plans in BDI-based agent systems. We then review approaches to reasoning about interactions that have been developed based on the goal-plan trees structure.

2.4.1 Goal-Plan Trees

Thangarajah et al. [60, 57, 58, 56] developed the notion of *goal-plan trees* (GPT) to represent the relationship between goals and plans for BDI agents. The root of a goal-plan tree is a top-level goal (goal-node), and its children are the plans that can be used to achieve the goal (plan-nodes). Plans may in turn contain subgoals (goal nodes), giving rise to a tree structure representing all possible ways an agent can achieve the top-level goal.

In addition to defining the relationships between goals and plans, goal-plan trees are also useful in two ways. First, they can be used to record information about the conditions necessary to achieve a (sub)goal or successfully execute a plan in the form of pre-, and postconditions associated with goal and plan nodes. Preconditions are conditions that need to be true for a plan to begin execution. In-conditions are conditions that need to remain true during the execution of a plan or the achievement of a goal. Postconditions are conditions that are made true by executing a plan or achieving a goal. A definite condition must hold for all executions of a plan or for all ways of achieving a goal, while a potential condition must hold for some executions of a plan or some ways of achieving a goal.[2]. Second, they can be used to define several notions that characterise the robustness of an agent program in a given environment (and hence the likelihood that a particular interleaving of two goal plan trees will result in interactions), including coverage and overlap. Thangarajah et al. [59] defined the *coverage* of an intention as the percentage of world states for which there is some applicable plan for any subgoal within an intention, and the *overlap* of an intention as the percentage of world states in which two or more plans for a goal are applicable.

Figure A.1 shows a simple goal-plan tree. The top-level goal, G_0, is to buy milk. There is a single plan, P_0, to achieve G_0. P_0 consists of two subgoals, G_1 and G_2. The subgoal G_1 can be achieved by either walking to the shop (plan P_1) or driving to the shop (plan P_2). There is a single plan, P_3, to achieve the subgoal G_2. The pre- and postconditions of plans are shown in the figure which specifies the conditions that must be true for the plan to begin execution and

[2]The detailed definition of pre- and postcondition can be found in Chapter 3

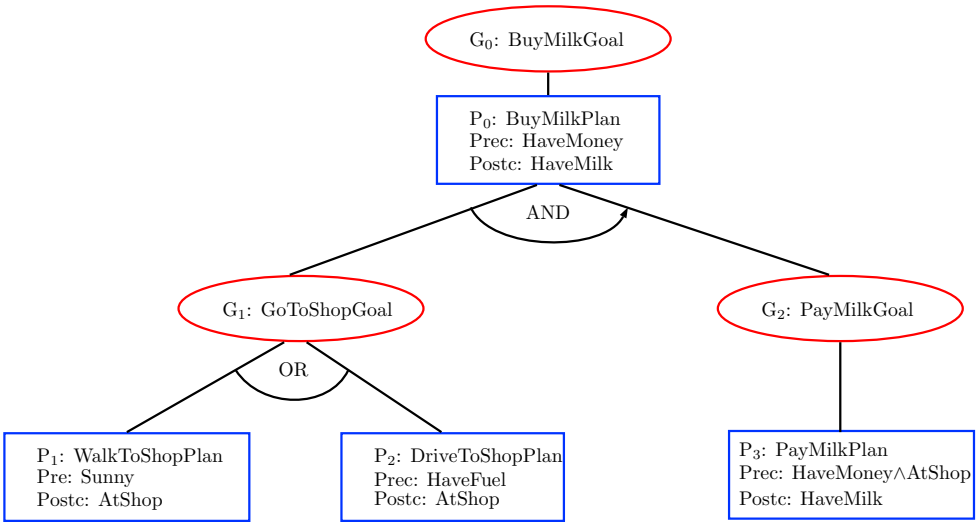


Fig. 2.4 An example of goal-plan tree

the changes to the environment (and the agent itself) respectively; for example, the plan P_1, *WalkToShopPlan*, has precondition that the weather is sunny, and postcondition that the agent is at the shop.

2.4.2 Summary-Information-Based Approach

Thangarajah et al. [60, 57, 58, 56] defined the *summary-information-based approach (SI)*, in which three types of interactions between goals are defined, including resource conflicts, negative interactions (which are caused by breaking the dependency links between plans) and positive interactions (synergies).

To reason about resources, Thangarajah, Winikoff, Padgham and Fischer [60] defined *necessary* and *possible* resources.[3] A necessary resource will definitely be required in every way that an agent achieves a goal. That is, no matter which plan the agent chooses to achieve the goal, a necessary resource is always required. Conversely, a possible resource is not always necessary, but may be required depending on the selection of plans. Necessary

[3]Thangarajah et al. are referring to both reusable and consumable resources

and possible resources of a goal are derived by combining the necessary and possible resources of all relevant plans for that goal. Similarly, necessary and possible resources of a plan are derived by combining the necessary and possible resources of the subgoals within that plan. These summarised resources requirements (i.e., necessary and possible resources for each goal and plan) can then be used to avoid resource conflicts by choosing suitable plans or appropriately ordering plan execution. Thangarajah et al. introduced an algorithm to compute if it is safe to add a new goal to the existing set of goals by checking each type of resource to see if adding a new goal would cause conflicts with any of the existing goals. The empirical results of their approach are given in [60]. Compared to the default intention progression approach in JACK, their approach shows an improvement in performance regarding the number of goals successfully achieved, with only a half second time increase in the computation cost.

Thangarajah, Padgham and Winikoff also discussed how to detect and avoid negative interactions by using summary information [57]. They defined *definite* and *potential* pre-, in- and postconditions for plans and goals. Definite and potential pre-, in- and postconditions of plans and goals are stored as summary information which are used to monitor the causal links between plans. A *causal link* is a dependency relation between two plans where the postcondition of the first plan makes the precondition of the other true. The postcondition of the first plan which is required by the second plan (as its precondition) is called the *preparatory-effect (p-effect)*. The plan that is dependent on the preparatory-effect is termed as the *dependent-plan*. The link between the dependent-plan and its corresponding preparatory-effect is called the *dependency-link*.

Summary information of $p-effect$ and in-conditions of each goal-plan tree is used to reason about conflicts between current goals and a new goal which may be adopted. They provided an algorithm for computing summary information at compile time, and for dynamically updating it at run-time. They also presented mechanisms to determine whether a newly adopted (sub)goal will definitely be safe to execute without conflicts (the new goal can be pursued in parallel with existing goals), or will definitely result in conflicts (pursuit of the new goal or existing goals must be deferred), or may result in conflicts (the goals can be pursued in parallel, but the agent needs to schedule conflicting steps to avoid the conflict). If the goal cannot be executed safely, execution of the intention is deferred. Their approach to resolve conflicts is similar to these used by Clement and Durfee [17–19], Clement and Durfee and Barrett [20]. The empirical results of their approach are given in [57]. SI showed an improvement in the number of goals successfully achieved compared to the default approach in JACK.

Thangarajah, Padgham and Winikoff also discussed how to exploit positive interactions between intentions [58]. When two or more plans have the same postcondition (in different intentions), it may be possible to merge these plans to improve the performance of the agent. To reason about positive interactions, they again use summary information to derive definite and potential postconditions of plans and goals. An algorithm to decide if it is feasible to merge plans was introduced in [58]. The empirical results of this approach were given in [58]. Compared to the approach without any reasoning (i.e., the default approach in JACK), their approach shows the expected reduction in the number of plans used to achieve the same set of goals.

However, the summary-information-based approach has several limitations. One limitation of their work is that they didn't consider interactions between intentions at the action level. All conflicts and positive interactions are resolved and exploited at the plan level. In some cases, conflicts can only be resolved at the action level. If intentions are merged at the plan level, the agent also cannot merge actions which have the same postcondition. Another limitation of their work is that they didn't show how to reason about all three types of interactions they discussed in conjunction. All results are given for each individual type of interactions between intentions.

2.4.3 Coverage-Based Approach

In [59], Thangarajah et al. presented an approach to intention selection based on the notion of coverage and overlap of intentions. They defined the *coverage* of an intention as the percentage of world states for which there is some applicable plan for any subgoal within an intention, and the *overlap* of an intention as the percentage of world states in which two or more plans for a goal are applicable. For example, consider a goal G_1 which can be achieved by two plans: P_1 and P_2. The preconditions of P_1 are $p \wedge q$ and the preconditions of P_2 are $p \wedge r$. In this situation, the coverage of G_1 is 0.375, as G_1 has at least one applicable plan if p is true and one of q and r is true (i.e., $p \wedge q \wedge \neg r$, $p \wedge \neg q \wedge r$ and $p \wedge q \wedge r$). The overlap of G_1 is 0.125, as G_1 has two applicable plans only when all p, q and r are true. Both coverage and overlap are calculated using a model counting solver reasoning over the space of the context conditions [59].

Coverage-based intention selection (C) is based on the notions of *coverage* proposed by Thangarajah et al. [59], and was implemented and evaluated by

Waters et al. in [66, 67]. C prioritises the progression of intentions with lowest coverage, as they are more likely to fail due to changes in the environment and the progression of other intentions. Waters et al. implemented two versions of C in [66, 67], which are called c0 and c1. c0 executes the progressable intention with lowest coverage and maintains focus as long as possible. Only when the current intention is finished or blocked will c0 recalculate coverage for all intentions and change focus to the progressable intention with lowest coverage. In c1, whenever a subgoal is posted, the coverage of each intention is recalculated and its progressibility determined. As a result, c1 will change focus to a lower coverage progressable intention, even if the current intention is progressable. Like SI, both c0 and c1 interleave intentions at the plan level. However, SI didn't consider the problem of avoiding resource conflicts and cannot exploit synergies between intentions.

2.4.4 Petri Nets Approach and Constraints-Based Model

Shaw et al. developed two methods to reason about the interactions between different goal-plan trees representing an agent's intentions. The first method uses Petri nets to represent and reason about the goals [54], while the second uses constraint satisfaction techniques to find an effective way to achieve the goals [55].

Petri nets are mathematical models used for describing and studying concurrent systems. Petri nets consist of *places*, *transitions* and *arcs*. Places are connected by arcs to transitions. The places from which an arc runs to a transition are called the input places of the transition, while the places to which arcs run from a transition are called the output places of the transition. Places

in a Petri net may contain *tokens* which is a discrete number or marks. The arcs can be inscribed with weightings indicating the number of tokens transferred at a time along that arc. Transitions can only fire when there are sufficient tokens, which acts as precondition for the transition. When the transition fires, it consumes the required input tokens, and creates tokens in its output places (as postcondition). Unless an execution policy is defined, Petri nets are executed non-deterministically, i.e. when there is more than one fireable transition, any one of them may fire.

Shaw et al. developed a Petri nets model to represent an agent's goals and plans, and to represent the same problems as expressed by goal-plan trees in the work by Thangarajah et al [60, 57, 58, 56]. Their Petri nets model allows an agent to handle both positive and negative interactions between multiple goal-plan trees, as well as reasoning about the limited availability of consumable resources. However, the Java-based simulator which was used to develop and run the Petri nets was unable to load the Petri nets when the number of goals being evaluated is large (e.g., 50 goals) [53].

Their second method is based on constraint satisfaction techniques. Constraint satisfaction attempts to find a solution which assigns values to a set of variables in the domain, such that the assignment of the values satisfies a number of constraints. A Constraint Satisfaction Problem (CSP) is represented as a tuple $< X, D, C >$, where $X = x_1, \ldots, x_n$ is a set of variables, D is a non-empty finite domain of values, C is a set of constraints over the variables. A solution to a CSP is a complete assignment of variables where each variable is assigned with a value and all constraints are satisfied [48]. The basic idea of their approach is to use constraints to represent goal-plan

trees (and their associated pre- and postconditions) and model the execution of goal-plan trees as a constraint satisfaction problem. Like the Petri nets approach, the Constraints-based model allows an agent to handle both positive and negative interactions between goal-plan trees, as well as reasoning about the limited availability of consumable resources. Compared to the Petri nets model, the performance of Constraints-based model was impressive with its ability to reason about large number of goals. However, the downside is that the Constraints-based model had a very large computational overhead if the number of goal were large (e.g., the Constraints-based model took over 9 hours to reason about 75 goals) [53].

They evaluated both of their approaches and showed that their approaches have a significant improvement compared to the approach without any reasoning ability. However, similar to SI and C, their approach also interleaves the execution of intentions at the level of individual plans. Thus, problems which can only be solved at the action level cannot be solved by their approaches. Moreover, their approaches have not been used to schedule intentions which have deadlines (i.e., they assume goals do not have deadlines), or contain fallible actions (which have more than one possible postconditions). Adding real-time constructs (e.g., deadlines) or fallible actions into their model will hugely affect the performance of their approach. For example, adding deadlines will increase the number of constraints in the CSP model.

2.5 Summary

In this Chapter, we reviewed the representative BDI agent programming languages to see how intentions are progressed in each of them, and provide a

detailed survey of previous approaches to reasoning about interactions between an agent's intentions. As mentioned above, there is no general approach implemented in the state of the art BDI agent programming languages that can avoid conflicts and exploit synergies between intentions. There are several approaches proposed in the literature to reason about interactions between intentions. However, they have certain limitations. The summary-information-based approach (SI) doesn't consider the interactions between intentions at the action level. All the conflicts and positive interactions are resolved/exploited at the level of individual plans. In some cases, conflicts can only be resolved at the action level. Interleaving the execution of an agent's intentions at the plan level also reduces the number of positive interactions (synergies). Another limitation of SI is that it hasn't been used to reason about all the three types of interactions (i.e., resource conflicts, negative interaction and positive interaction) in conjunction. All results are given for each individual type of interactions between intentions. It may be possible that different types of reasoning in their approaches will interference with each other. Like SI, the coverage-based approach (C) also schedules intentions at the plan level. Moreover, C doesn't have the ability to resolve resource conflicts and to exploit synergies between intentions. Like SI and C, the Petri Nets Model and Constraints-Based Model also schedule intentions at the plan level. The Java-based simulator which was used to develop and run the Petri nets was unable to load the Petri nets when the number of goals being evaluated is large, and the Constraints-based model takes very long to complete its reason when the number of goals are large.

Chapter 3

Reasoning About Goal-Plan Trees

In this chapter, we first introduce data structures to represent goals, plans and actions. We then extend the goal-plan tree structure proposed by Thangarajah et al. [60, 57, 58, 56] to include an explicit representation of actions. We define fallible actions which have more than one possible postcondition. Goals are extended to contain real-time constructs including deadlines and preferred achievement time. We end this chapter by precisely defining the intention progression problem, in which the intentions are represented using a set of goal-plan trees and their corresponding current step pointers.

3.1 Beliefs, Goals, Plans and Actions

We consider BDI agent programming languages in which there is an explicit representation of beliefs, goals, plans and actions. For simplicity, we assume that the agent's beliefs are represented by a set of literals Φ^+ and its negation Φ^-. (In practice, beliefs in BDI agent programming languages are first order ground terms, so this assumption is really without loss of generality. We use

Φ to represents the set of literals and its negation.) Actions may have pre- and postconditions. Pre- and postconditions specify respectively the states of the environment which must hold immediately before an action is executed, and which are brought about by executing an action. Plans may also have preconditions [1] which indicate the situation in which the plan is applicable. More precisely, we define precondition and postcondition as follows.

Precondition: a precondition is a set of literals $\phi = pre(\eta)$, $\phi \subseteq \Phi$ that must be true for η to begin execution (where η is an action or a plan). The precondition of an action or a plan does not need to be true once the action or plan has begun execution. For example, a robot may have an action to pick up a block which has a precondition that the robot is not currently holding anything.

Postcondition: a postcondition (or effect) is a set of literals $\psi = post(A)$, $\psi \subseteq \Phi$ that are or may be made true by executing A (where A is an action). For example, an action to walk to the library has a postcondition that the agent is at the library.

Goals: each goal G is associated with a set of plans $P_1, \ldots, P_n$ that achieve G. To allow reasoning about real-time agency, we extend the notion of goal found in languages such as Jason [9] and 2APL [23], by introducing two temporal constructs: a preferred achievement time (soft deadline) p, and a deadline (hard deadline) d, where $p \leq d \leq \infty$. The *preferred achievement time* specifies the time by which a goal should ideally be achieved; however achieving the goal

[1] The precondition of a plan is also called context condition in languages including JACK[68], dMARS [29] and Jason [9]

after the preferred achievement time still has value. The *deadline* specifies the time by which a goal must be achieved; achieving the goal after the deadline has no value.

Plans are the means an agent can use to achieve its goals. An agent may have more than one plan to achieve a goal, and for each of its goals there must be at least one plan to achieve it. Each plan consists of a head and a body. The head of a plan consists of a trigger (an event template) and a precondition. The trigger of a plan is either a goal or a change in the agent's beliefs. The precondition of a plan specifies the conditions that must be true for the plan to begin execution.[2]

The body of a plan is composed of execution steps which are either basic actions that directly change the state of the environment (or the agent) or subgoals which are in turn achieved by other plans. For example, a plan to buy milk consists of a sequence of two subgoals: go to the supermarket and buy milk.

Actions are the basic steps an agent can perform in order to change the environment. Each action A is specified by its precondition and one or more postconditions. Actions may be *fallible* or *infallible* [77]. An *infallible* action has a single postcondition, while the execution of a *fallible* action results in one of a set of possible postconditions.

We denote the postconditions of a fallible action by $[\psi_0, \ldots, \psi_n]$, where each ψ_i is a possible postcondition of the action. Some postconditions may be more likely than others. If the probability distribution over possible post-

[2]We assume the precondition of a plan is derived from the preconditions of the actions in the plan.

conditions is known, we denote the possible postcondition of an action by $[(\psi_0, p_0), \ldots, (\psi_n, p_n)]$ where ψ_i is a possible postcondition of the action and p_i is the corresponding probability of its occurring. We further require that the sum of these probabilities equals to 1, i.e., $\Sigma(p_0, \ldots, p_n) = 1$. For each fallible action, we assume there is a single *intended* postcondition that the agent programmer assumes (or hopes) will occur when the action is executed. All other (unintended) postconditions correspond to anticipated failure modes of the action. For example, the intended postcondition of a fallible action may establish a precondition of a subsequent action in the plan, while the other outcome(s) do not.

Each action A also has an *estimated execution time*, $e(A, \Phi)$, which is a function of the agent's beliefs Φ. The estimated execution time may be specified by the programmer, e.g., the expected execution time of a 'move' action may be based on the distance to be moved, or learnt from experience, e.g., the agent may learn that traversing a hallway during a busy period takes longer than in the middle of the night.

3.2 Goal-Plan Trees With Actions

As in [56–58, 60], we use the notion of *goal-plan trees* to represent the relations between goals, plans and actions, and to reason about the interactions between intentions. As discussed in Section 2.4.1, the root of a goal-plan tree is a top-level goal (goal-node), and its children are the plans (plan-nodes) that can be used to achieve the goal. In general there are several alternative plans to achieve a goal, hence, the child plans are viewed as "OR" nodes. On the other hand, plan execution involves achieving all the subgoals in the plan.

Thus, the children of a plan-node are viewed as "AND" nodes. In [56–58, 60], there are no explicit representation of actions, and hence the relationship between actions are not represented. To reason about the progression of an agent's intentions at the action level, we extend the definition of goal-plan trees to include an explicit representation of actions (action-nodes) in plans in addition to subgoals [77, 76]. Figure 3.1 is the Backus-Naur Form (BNF) of the goal-plan tree used in this thesis.

$$
\begin{array}{rcl}
\langle \text{GoalType} \rangle & ::= & \langle \text{GoalTypeName} \rangle\ \langle \text{Plans} \rangle\ \langle \text{Deadline} \rangle\ \langle \text{PreferredTime} \rangle \\
\langle \text{GoalTypeName} \rangle & ::= & \langle \text{Label} \rangle \\
\langle \text{Plans} \rangle & ::= & \langle \text{PlanTypeName} \rangle\ (,\ \langle \text{PlanTypeName} \rangle)^{*} \\
\langle \text{PlanTypeName} \rangle & ::= & \langle \text{Label} \rangle \\
\langle \text{PlanType} \rangle & ::= & \langle \text{PlanTypeName} \rangle\ \langle \text{Precondition} \rangle\ \langle \text{PlanBody} \rangle \\
\langle \text{PlanBody} \rangle & ::= & \langle \text{ExecutionStep} \rangle\ (;\ \langle \text{ExecutionStep} \rangle)^{*} \\
\langle \text{ExecutionStep} \rangle & ::= & \langle \text{ActionTypeName} \rangle \mid \langle \text{GoalTypeName} \rangle \\
\langle \text{ActionType} \rangle & ::= & \langle \text{FallibleActionType} \rangle \mid \langle \text{InfallibleActionType} \rangle \\
\langle \text{FallibleActionType} \rangle & ::= & \langle \text{ActionTypeName} \rangle\ \langle \text{Precondition} \rangle\ \langle \text{Postconditions} \rangle \\
 & & \langle \text{EstimatedExecutionTime} \rangle \\
\langle \text{InFallibleActionType} \rangle & ::= & \langle \text{ActionTypeName} \rangle\ \langle \text{Precondition} \rangle\ \langle \text{Postcondition} \rangle \\
 & & \langle \text{EstimatedExecutionTime} \rangle \\
\langle \text{ActionTypeName} \rangle & ::= & \langle \text{Label} \rangle \\
 & & \\
\langle \text{Postconditions} \rangle & ::= & (\langle \text{Postcondition} \rangle, \langle \text{Probability} \rangle) \\
 & & (,\ (\langle \text{Postcondition} \rangle, \langle \text{Probability} \rangle))^{+} \\
\langle \text{Precondition} \rangle & ::= & \varepsilon \mid \langle \text{Condition} \rangle\ (,\ \langle \text{Condition} \rangle)^{*} \\
\langle \text{Postcondition} \rangle & ::= & \varepsilon \mid \langle \text{Condition} \rangle\ (,\ \langle \text{Condition} \rangle)^{*} \\
 & & \\
\langle \text{GoalInstance} \rangle & ::= & \langle \text{InstanceName} \rangle\ \langle \text{GoalType} \rangle \\
\langle \text{PlanInstance} \rangle & ::= & \langle \text{InstanceName} \rangle\ \langle \text{PlanType} \rangle \\
\langle \text{ActionInstance} \rangle & ::= & \langle \text{InstanceName} \rangle\ \langle \text{ActionType} \rangle \\
\langle \text{InstanceName} \rangle & ::= & \langle \text{Label} \rangle \\
 & & \\
\langle \text{Condition} \rangle & ::= & \langle \text{Literal} \rangle \mid \textit{NOT}\ \langle \text{Literal} \rangle \\
\langle \text{Literal} \rangle & ::= & \textit{string} \\
\langle \text{Label} \rangle & ::= & \textit{string} \\
\langle \text{Probability} \rangle & ::= & \textit{double} \in [0, 1] \\
\langle \text{EstimatedExecutionTime} \rangle & ::= & \textit{double} \in [0, \infty] \\
\langle \text{Deadline} \rangle & ::= & \textit{double} \in (0, \infty] \\
\langle \text{PreferredTime} \rangle & ::= & \textit{double} \in (0, \infty]
\end{array}
$$

Fig. 3.1 BNF of goal-plan tree with actions

GoalTypeName, *PlanTypeName* and *ActionTypeName* are labels that indicate the type of goal, plan and action respectively. A *GoalType* is a template for a goal which represents a particular type of goal. Similarly, a *PlanType* is a template for a plan, and a *ActionType* is a template for an action. An *ActionType* can be used to represent a faillible action (*FallibleActionType*) or an infallible action (*InfallibleActionType*). All the *GoalType*, *PlanType* and *ActionType* are created at compile-time from the set of agent's plans. *InstanceName* is an unique label that is attached to each goal, plan and action instance, as there can be multiple instances of the same goal/plan/action type. All the *InstanceNames* of goals, plans and actions are assigned at run-time. When an agent chooses to pursue a top-level goal, a new instance of the (corresponding) goal-plan tree is created which contains instances of all the goals, plans and actions in the tree.

In [56–58, 60], goal-plan trees and summary information are preliminarily designed to improve the performance of JACK agents. Therefore, as in JACK, plans and goals have *in-conditions* which specify the states of the environment which must hold for the duration of the execution of the plan or the achievement of the goal. Sub-goals may be combined in sequence or in parallel with each other. A parallel operator $\|$ is used to denote sub-goals executed in parallel. In this thesis, for simplicity we assume plans and goals do not have in-conditions, and actions and subgoals are executed sequentially.[3]

Figure 3.2 shows a simple goal-plan tree template for a delivery goal in the elevator domain [40]. *Deadline* represents the deadline for each goal, $Preferred$ is the preferred achievement time for each goal, and $EstimatedTime$

[3]Note, not all BDI agent programming languages have the notion of in-condition and parallel composition, e.g., Jason.

represents the estimated execution time for each action. G_i, P_i, A_i are instance names, $DeliveryGoal(p, f_1, f_0)$, $DeliveryPlan(p, f_1, f_0)$ and so on are type names. The top level goal, G_0, is to deliver a passenger p from floor f_1 to floor f_0. There is a single plan, P_0, to achieve G_0. P_0 consists of two subgoals, G_1 and G_2 and two actions, A_0 and A_1. Each subgoal may be achieved by two subplans (P_1 and P_2 for G_1, and P_3 and P_4 for G_2). The precondition of plans, and the pre- and postcondition of actions are shown in the figure; for example, the infallible action A_0, $Board(p, f_0)$, has precondition that both the $lift$ and passengers p are at floor f_0, and postcondition that the passengers are in the lift.

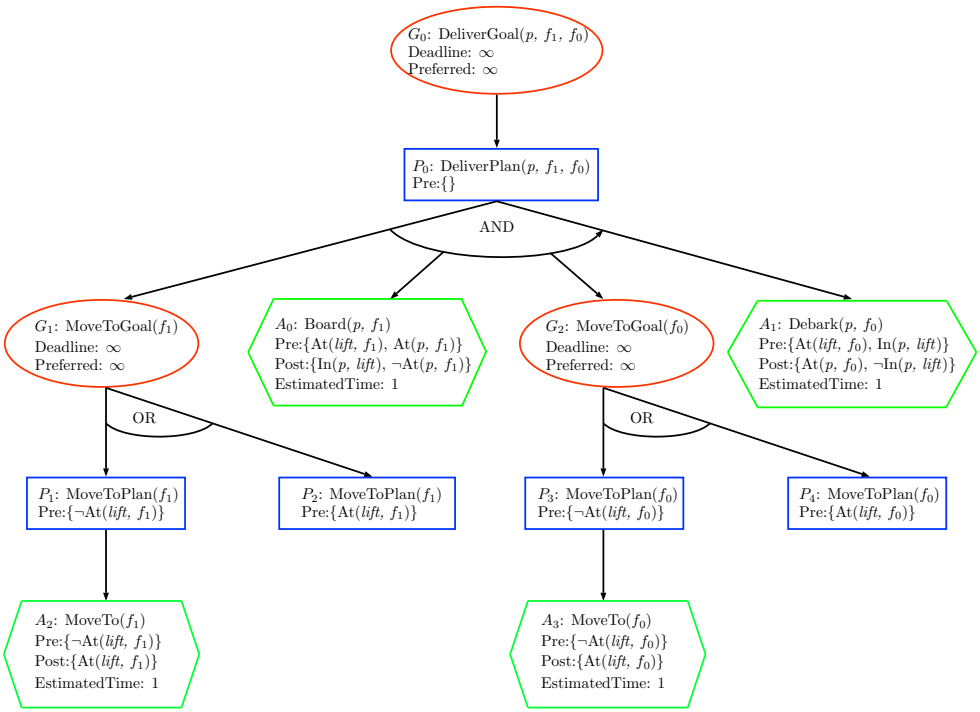


Fig. 3.2 An example of goal-plan tree.

3.3 The Intention Progression Problem

A BDI agent usually has more than one plan to achieve a given goal. Plans are applicable in different situations depending on their preconditions. When there are multiple applicable plans to achieve a particular goal, the *plan selection* problem arises, i.e., which plan should the agent apply to achieve the goal. Plan selection is important as it may have implications for which other goals the agent can achieve. For example, suppose an agent has a goal to go to the supermarket and another goal to buy milk. The agent can go to the supermarket either by taxi or on foot. The choice of the plan to achieve its first goal (go to the supermarket) may affect the achievement of its second goal (buy milk), e.g., if it goes to the supermarket by taxi, then it may not have enough money to buy the milk, or if it goes to the supermarket on foot, the supermarket may close before it arrives.

When an agent has more than one intention, the choice of which intention to execute next is known as the *intention selection* problem.[4] Intention selection is also important as it may affect the achievement of an agent's goals. For example, suppose a Mars Rover has a goal to perform a rock experiment at location A and another goal to recharge its battery at location B. The selection of which intention to execute next may affect the achievement of the goals, e.g., if the Mars Rover executes its first intention (i.e., move to location A), then it may fail to finish its second intention as the current battery level is not enough for it to move to location B.

[4]Note, the intention selection problem does not include the problem of which plan should be used to achieve a (sub)goal if the next step in an intention is a (sub)goal.

We focus on solving the *intention progression problem* at run time, which takes both the plan selection and the intention selection problem into account. In order to provide a generic approach that can be integrated into most BDI agent programming languages, we use goal-plan trees to represent an agent's top-level goals (and their hierarchies below). The progression of an intention to achieve a goal amounts to choosing a path through the corresponding goal-plan tree. Each path corresponds to a different way of achieving the top-level goal forming the root of the tree. The path specifies a sequence of plans, actions, subgoals and sub-plans that, if executed successfully, will achieve the top-level goal. The execution of an agent program thus corresponds to an interleaving of paths through each of the goal-plan trees corresponding to the agent's current intentions.

Different interleavings of the execution steps in goal-plan trees represent different ways of progressing an agent's intentions. Depending on the problem domain, the requirements of how the agent should progress its intentions may be different. For example, in some cases, programmers might require the agent to achieve all its goals as soon as possible, i.e., the agent is aiming to select the least time-consuming plans to achieve its goals and may also merge some of the execution steps in the plans in different intentions if possible. In some other cases, programmers might require the agent to interleave its intentions such that the achievement of its goals are "fair", i.e., the agent is aiming to share its focus of attention across its active intentions. We are aiming to provide a generic approach to intention progression, such that programmers can customise the approach for different problem domains. We assume a function f_μ takes the current intention of an agent and the environment as

input, and returns a utility value representing the performance of the agent. By defining the function f_μ, programmers can customise their requirements for the agent. For example, if the programmer defines a function f_μ which returns the number of goals achieved, then the more goals the agent achieved, the better the agent's performance is. That is, the agent is aiming to achieve as many goals as possible.

We define the *intention progression problem* as the problem of choosing an interleaving of paths through each goal-plan tree representing the agent's current intention, such that the interleaving leads to a state which has the largest utility value with respect to the function f_μ. To make the problem precise, we first introduce the notion of the *current step* of a goal-plan tree. The current step of a goal-plan tree t_i is a pointer to the current step in t_i, which may be a primitive action or a subgoal. The path from the root of the goal-plan tree to the current step represents the choices made (and steps executed) so far in achieving the top-level goal. When an agent adopts an intention for a new top-level goal G_i, the current step of the goal-plan tree for G_i is initially set to G_i itself. If the current step is a subgoal G, advancing the current step involves choosing a plan P for G, and setting the current step to be the first step in P. If the current step is an action A, advancing the current step involves returning the action A, and setting the current step to be the action or subgoal following A in the same plan P. If there is no such action and subgoal, then the current step is set to be the action or subgoal following the parent goal of plan P.

The intention progression problem can now be stated as: given a set of goal-plan trees with their corresponding current step pointers, representing the current intentions of the agent $\{t_1, \ldots, t_n\}$, a set of condition variables Env

representing the current state of the environment and the agent, and an function f_μ, return a next step of a goal-plan tree t_i to be executed at each deliberation cycle. By executing these steps in each deliberation cycle, a final state s is returned. We further require that the final state s has the largest utility value with respect to the function f_μ. That is, there is no state $s^{'}$ which has a larger utility value than s can be generated from the initial state.

Chapter 4

The *SAM* Scheduler

In this chapter, we briefly introduce Monte-Carlo Tree Search (MCTS) and its variant Single-Player Monte-Carlo Tree Search (SP-MCTS), and discuss the MCTS and SP-MCTS algorithms in detail. We then present one of the main contributions of the thesis: the *SAM* scheduler, which uses (SP-)MCTS to schedule an agent's intentions.

4.1 Monte-Carlo Methods

Monte-Carlo methods are a class of computational algorithms that rely on repeated random sampling to obtain approximated values, and have been used in a wide range of research areas including physics (e.g., simulations), mathematics (e.g., computation of integral functions) and computer science (e.g., games).

Monte-Carlo methods are mostly used to make simulations or to compute values in given problem domains. For example, to know the percentage of people who don't like green peppers, we may need to ask every living person

on the Earth, which is not practically possible. Instead, we could calculate an approximation of the true value by random sampling, e.g., ask a random amount of people if they like green peppers or not and calculate the percentage. With a significant number of tries, the average values we are calculating will give an approximation of the true value we are searching for. To give a good approximation, the number of tries should be large enough that we can trust the results, and we also require the choice of the samplings are unbiased.

Abramson [1] was the first to propose a way to use Monte-Carlo methods for playing games, with his expected outcome model. When applied to games, Monte Carlo methods randomly simulate the results of the game for each move that is possible at the current state. Then, the move which produced the highest average game scores amongst its samples is selected as the best move. Compared to standard tree-search methods, Monte-Carlo methods have several advantages. First, Monte-Carlo methods only need to know the rules of the game and how to calculate the results of the game. No additional information is required. In contrast, tree-search methods may require additional domain-specific knowledge such as the estimated value of a mid-game position. Second, Monte-Carlo methods work well with games that have random or unknown elements, while the performance of tree-search methods are usually significantly affected by the random or unknown elements in the game, as these elements may increase the depth of the search tree. Finally, it is usually very computational expensive for tree-search methods to reach appropriate search depths in games that have a large branching factor. In contrast, Monte-Carlo methods can easily cope with large branching factors by increasing the amount of sampling.

However, Monte-Carlo methods also have disadvantages compared to tree-search methods. One disadvantage of Monte-Carlo methods is that they lack tactical insight. Tree-search methods are able to recognise a move which leads to a certain win if the game is played in a particular way. Monte-Carlo methods are unable to recognise such move, as they only calculate scores for the next move and the score of this move is comprised of good and bad simulation results.

4.2 Multi-Armed Bandit Problem

The multi-armed bandit problem (also known as N-armed bandit problem) is a problem in which a gambler needs to choose amongst N actions (i.e., the N arms of a multi-armed bandit slot machine), in order to maximise the cumulative reward by consistently taking the optimal action [37]. With an unknown reward distribution, the choice of action is difficult. The potential values of executing each action can only be estimated using the past observations. Choosing the action with the highest mean award (so far) may not result in the highest overall reward, as the action which has the highest overall reward might have not been discovered yet. Therefore, we need to balance the exploitation of the actions that are currently believed to be optimal with exploration of other actions which may turn out to be optimal in the next few runs. To solve this problem, regret is defined as the expected loss due to not performing the best action. One possible way to maximise the cumulative reward in multi-armed bandit problem is to minimise the player's regret.

$$R_N = \mu^\star n - \mu_j \sum_{j=1}^{N} \mathbb{E}[T_j(n)] \tag{4.1}$$

In equation 4.1, R_N is the regret after n plays. Each play involves choosing one of the N actions (arms) and playing it. $\mu^\star$ is the best possible expected value, μ^j is the value of playing the j^{th} action, N is the total number of actions and $\mathbb{E}[T_j(n)]$ represents the expected number of plays for the j^{th} action in the first n plays (i.e., the number of times the j^{th} action has been played in the first n plays). It is important that one should not miss the optimal arm due to temporarily promising rewards from a sub-optimal arm. Thus, the *upper confidence bound(UCB)* that any given arm will be optimal is useful.

The simplest UCB policy proposed by Auer et al. [6] is called UCB1, which can work without any prior knowledge regarding the reward distributions. The formula to calculate the UCB values for the j^{th} action is given in Equation 4.2. The action which has the highest UCB1 value will be chosen and played in the $(n+1)^{th}$ play.

$$UCB1 = \overline{X}_j + \sqrt{\frac{2\ln N}{N_j}} \tag{4.2}$$

Where $\overline{X}_j$ is the average reward from j^{th} action, N_j is the number of times the j^{th} action was played and N is the total number of plays. In this formula, the term $\overline{X}_j$ encourages the exploitation of high reward actions, while the term $\sqrt{\frac{2\ln N}{N_j}}$ encourages exploration of less visited actions. It is obvious that $N_j = 0$ yields a UCB1 value of ∞. This means that untried actions are assigned the largest value, which ensures that all actions are considered at least once before other actions are exploited further.

4.3 Monte-Carlo Tree Search

Due to the lack of tactical insight of Monte-Carlo methods, Monte-Carlo Tree Search (MCTS) [15, 39, 14] was proposed which combines the advantage of both Monte-Carlo methods and tree-search methods. MCTS is a method for making decisions in a given domain by running stochastic simulations in the decision space and iteratively building a search tree using the results of the simulations. It was originally developed for game playing (e.g., Go [15]), and also has been successfully applied in problems where the domain can be represented as trees of sequential decisions (e.g. planning problems).

Monte-Carlo Tree Search is built on two fundamental concepts: random simulation can be used to estimate the approximate value of executing an action (Monte Carlo methods), and these estimation values can be used to adjust the policy towards a best-first strategy (multi-armed bandit problem). The algorithm iteratively builds a game tree based on the simulation results until some predefined computational (e.g., time, memory or number of iterations) budget is reached. The algorithm then halts and the best performing action (or move) is returned. Each node in the search tree represents a state in the problem domain. In addition, each node also contains a record of the number of times it has been visited and the total simulation value of the node. The edges to child nodes represent actions leading to subsequent states. Each iteration of the algorithm consists of 4 phases: *selection*, *expansion*, *simulation* and *back-propagation* [14]. Figure 4.1 shows one iteration of the basic MCTS algorithm.

Selection: In the selection phase, a leaf node n_e in the search tree is selected for expansion. A node may be expanded if it represents a non-terminal state (a

state in which it is possible to perform a MCTS action leading to a subsequent state in the problem domain). The problem of selecting a node for expansion is similar to the multi-armed bandit problem: the value of a child node is the expected reward value approximated by random simulations, and hence these rewards correspond to random variables with unknown distributions. The policy to select a leaf node needs to control the balance between exploration and exploitation. Therefore, UCB1 (which is also called Upper Confidence Bound for Trees (UCT) in MCTS) is a popular algorithm to select leaf nodes in the selection phase [39].

Expansion: In the expansion phase, the node selected in the selection phase n_e is expanded by adding child nodes representing the states after executing an MCTS action from n_e. The edges from n_e to its child nodes represent executing one MCTS action from node n_e.

Simulation: In the simulation phase, simulations are run from one of the newly added child nodes n_s to produce estimated values for this node. It is similar to the simulations performed during standard Monte-Carlo methods. By default, the simulation is run randomly (unbiased), which means MCTS works domain independently. However, domain-specific knowledge can also be integrated into the simulation policy to improve the performance of the algorithm (e.g., ignore actions and states which won't be tried in reality.)

Back-propagation: Finally, the simulation values are back-propagated from node n_s to all nodes on the path to the root node.

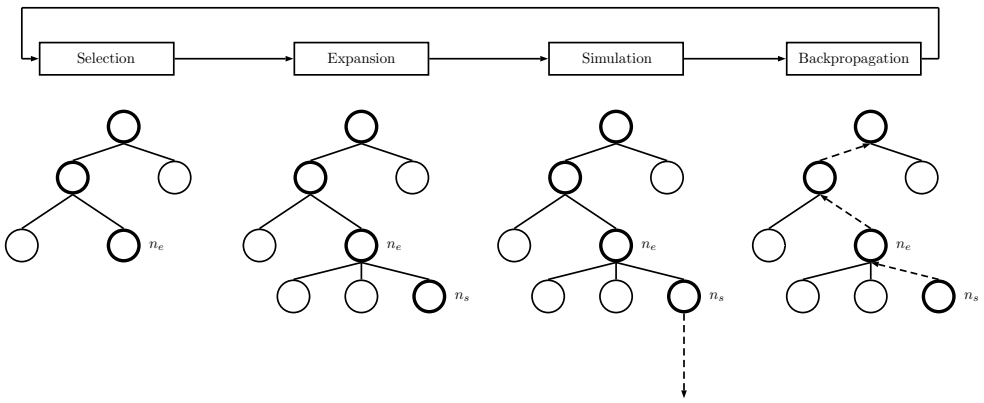


Fig. 4.1 One iteration of MCTS

As soon as the predefined computational budget is reached, the action with best performance is returned. Schadd [51] proposed four criteria for selecting the best action, based on the work of Chaslot et al. [14].

1. Max child: select the root child with the highest simulation value.
2. Robust child: select the most visited child of the root node.
3. Max-Robust child: select the child of the root node with both highest visit count and the highest simulation value. If there is no such child node, then additional iterations will be run until there is an acceptable child of the root node.
4. Secure child: select the child which maximises a lower confidence bound.

For different problems, the best action returned by these four criteria may be slightly different. Programmers can choose the most suitable criteria for different problems.

MCTS has two main characteristics which make it a suitable choice for a variety of domains [12].

Aheuristic Perhaps the most significant benefit of MCTS is it can work without domain-specific knowledge, which means MCTS can be applied to any domain that can be modelled as a search tree. In games like chess, where reliable heuristics were already proposed, minimax search (in which the players maximise their reward while the opponent seeks to minimise it) performs admirably well. However, the performance of minimax significantly depends on the heuristic used to evaluate the tree nodes. In games such as Go, where useful heuristics are much more difficult to formulate, the performance of minimax degrades significantly. In contrast, MCTS is based on standard Monte-Carlo methods which do not require any domain-specific knowledge.

As discussed, MCTS can be applied without domain-specific knowledge. However, using domain-specific knowledge may improve its performance. For example, 3×3 patterns for detecting cut moves are used as a heuristic to improve the playing strength of the MCTS-based Go program *MoGo* [65]. There are trade-offs to consider when decides when to use domain-specific knowledge and how to use it. One advantage of uniform random simulation is speed. Without domain-specific knowledge, the algorithm can perform more simulations. With domain-specific knowledge, the algorithm may reduce the number of possible simulations, but may also reduce the variance of simulation results.

Anytime Another important characteristic of MCTS is that it is an anytime algorithm. An anytime algorithm can return a result at any moment in time. This is because MCTS back-propagates the simulation results in the back-propagation phase immediately, such that all values in all nodes are always

up-to-date. Allowing the algorithm to run additional iterations often improves the performance of the algorithm.

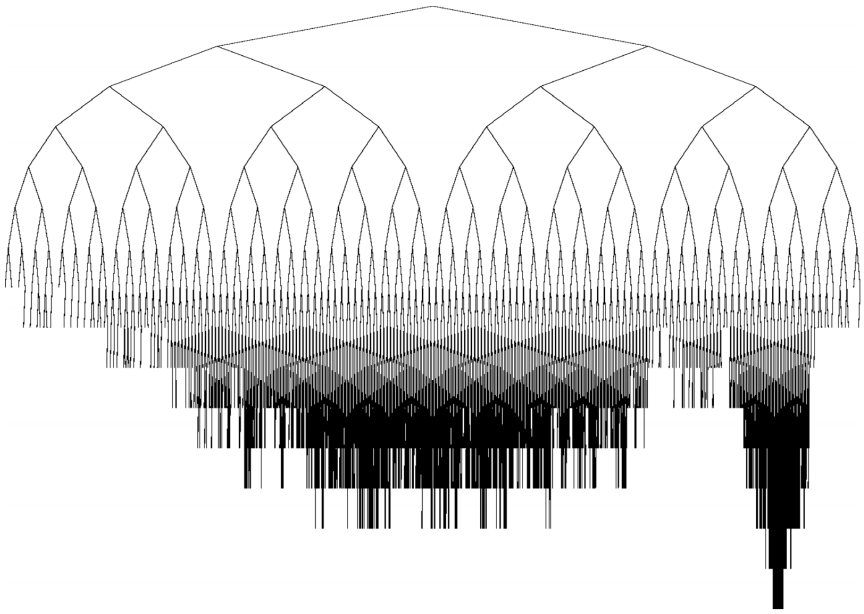


Fig. 4.2 Asymmetric search tree [21]

4.3.1 Single-Player Monte-Carlo Tree Search

MCTS was originally developed for solving two-player combinatorial games, and has been applied rarely in single-player puzzles (games). Single-player puzzles have two essential differences compared to two-player combinatorial games. The first difference is the range of result values. Typically, in a two-player game, the result of the game is denoted by *win*, *draw* and *loss*, i.e. 1, 0, -1 or 1, 0.5, 0. The average simulation results of a node in the search tree always stay within the range [-1, 1] or [0, 1]. In contrast, in a puzzle (or a planning problem), the player (or the agent) usually has an arbitrary score which is not in a preset interval. For example, in points-based puzzles, e.g.,

SameGame [52], the result of the puzzle is represented using the scores in the game. The UCB1 formula that is used to calculate the UCB value for each node requires the reward value (i.e., the simulation value) of a node within the range [0, 1]. Therefore, in order to solve single-player puzzles we need to scale the scores of puzzles to the range [0, 1].

The second difference is that puzzles do not have any uncertainties on the opponent's play. It means that the line of play doesn't need to consider the opponent's move. For example, when playing Go, we need to consider the possible moves from our opponent. While, playing single-player puzzles like SameGame you don't need to worry about opponent's move, all the moves are made by the player.

In order to solve single-player games (puzzles), Schadd et al. [52] introduced a variant of MCTS, called Single-Player Monte Carlo Tree Search (SP-MCTS). It has been used successfully to solve not only single-player games [52], but also reentrant scheduling problems [41]. SP-MCTS differs from MCTS in how to select a child node in the selection phase, and in which action to return when the algorithm terminates. Instead of using UCT to select a leaf node in the selection phase, Schadd et al. [52] proposed a modified version of UCT (which we denote UCT^{SP}) for selecting a leaf node in solving single-player games (see Equation 4.3). This formula is used to calculate the modified UCB value for a node n_j, which is a child node of node n_i.

$$UCT^{SP} = \overline{X}_j + C\sqrt{\frac{2\ln N}{N_j}} + \sqrt{\frac{\sum x^2 - N_j\overline{X}_j^2 + D}{N_j}} \tag{4.3}$$

In Equation 4.3, $\overline{X}_j$ is the average simulation result of the child node n_j, N_j is the number of times node n_j was visited and N is the number of times

the parent node of n_j (i.e., n_i)) was visited. The first two terms in UCT^{sp} are similar to those in Equation 4.2. The first term $\overline{X}_j$ is used to encourage the exploitation of high reward (MCTS) nodes, the second term $C\sqrt{\frac{2\ln N}{N_j}}$ encourages the exploration of less visited nodes. For single-player puzzles, a third term, representing a possible deviation of the child node's simulation value is added. $\sum x^2$ is the the sum of the squared simulation results so far achieved in the node n_j, and $N_j\overline{X}_j^2$ is the expected simulation results in n_j. $\sqrt{\frac{\sum x^2 - N_j\overline{X}_j^2}{N_j}}$ is the standard deviation of the simulation values of node n_j. The constant D is used to inflate the standard deviation for infrequently visited nodes, so that the rewards for such nodes are considered to be less certain. All together, the term $\sqrt{\frac{\sum x^2 - N_j\overline{X}_j^2 + D}{N_j}}$ favours the nodes with large deviation values. That is, instead of selecting the node which has all its simulations of medium strength, SP-MCTS favours the node which has the best simulation results. Compared to UCB1 (see Equation 4.2), another constant C is also required, as the value of the game is not within the preset interval [0, 1] anymore. Changing the value of C and D changes the balance between exploitation and exploration for a particular problem. In [52], Schadd et al. present three different pairs of (C, D) values. In this thesis, we will use the values (0.1, 32) which has best performance when the number of nodes generated in the search tree is less than 10^5 [52].

Compared to MCTS, SP-MCTS also differs in the ways the algorithm selects actions when it terminates. Unlike two-player games, single-player puzzles do not have any uncertainty on the opponent's play. When the algorithm terminates, the actions which leads to the best simulation result is selected.

4.4 The *SAM* Scheduler

In Chapter 3, we defined the intention progression problem as the problem of deciding which step to progress (intention selection) and possibly how to progress it (plan selection) in each deliberation cycle. When all the actions in the problem domain are infallible, the intention progression problem is similar to single player puzzles (i.e., all the plays and moves are made by the player (or the agent) and there is no uncertainty from the opponents' move), and thus can be solved by using SP-MCTS. However, when there are fallible actions in the problem domain, the agent cannot control which postcondition of a fallible action will be achieved. We therefore consider the intention progression problem with fallible actions as a two-player game, and the execution of fallible actions are treated as an opponent's move (i.e., the agent knows which actions are fallible and what the possible postconditions are).

In this section, we propose the *SAM* scheduler (see Algorithm 3) based on Monte-Carlo Tree Search and Single-Player Monte Carlo Tree Search to schedule the progression of an agent's intentions. The *SAM* scheduler decides which intention should be progressed and how to progress it in each deliberation cycle. In order for *SAM* to work, the programmer needs to define the precondition of each plan[1], and the pre- and postcondition of each action that are relevant to the problem domain. For example, in an agent program, we may define the postcondition of an action *TaxiToBeach* as *AtBeach* and *TaxiFarePaid*, which contains all the postconditions of *TaxiToBeach* that are relevant to the problem, and ignores the irrelevant postconditions such as the

[1] We assume the precondition of a plan is derived from the actions in the plan.

location of the taxi. To better understand the algorithm of the *SAM* scheduler and how the *SAM* scheduler interacts with the goal-plan trees, we define an abstract data type for goal-plan trees, and all the operations that can be performed on each goal-plan tree are shown as follows:

- `node getCurrentStep();`

  This operation returns the current step in the goal-plan tree which might be a goal or an action. Initially (i.e., when the instance of the goal-plan tree for this top-level goal is generated), the current step is set to be the top-level goal of the goal-plan tree.

- `int getStartTime();`

  This operation returns the time[2] the goal-plan tree is generated.

- `int getCompletionTime();`

  This operation returns the time when the top-level goal of the goal-plan tree is achieved. If the top-level goal has not been achieved, -1 is returned.

- `action getCurrentAction();`

  This operation returns the action that is currently executing in the goal-plan tree. If there is no such action, null is returned.

- `int getActionStartTime();`

  This operation returns the start time of the action that is currently executing in the goal-plan tree.

- `int getActionEstimatedTime(action a);`

  This operation returns the estimated execution time of the action `a`.

[2]For simplicity, the time is represented as the deliberation cycle number.

- `int getDeadline();`

  This operation returns the deadline of the goal-plan tree.

- `int getPreferredTime();`

  This operation returns the preferred achievement time of the goal-plan tree.

- `boolean executable(Env env);`

  This operation returns true, if the next step in the goal-plan tree is executable. If the current step is an action, and its precondition holds in the current environment, and there is no action that is currently executing in the goal-plan tree, then this operation returns true. If the current step is a subgoal, and there is at least one plan can be applied to achieve it (i.e., the precondition of the plan holds in the current environment), and there is no action that is currently executing in the goal-plan tree, then this operation returns true. Otherwise, this operation returns false.

- `ArrayList<plan> getApplicablePlans(Env env);`

  If the current step is a subgoal, this operation returns the set of plans that are applicable in the current environment. Otherwise, this operation returns an empty set.

- `action getCompletedAction();`

  This operation returns the actions that finish its execution in next deliberation cycle.

- `action progress(plan` *P*`/action` *A*`);`

  This operation advances one step in the goal-plan tree. If the current step is a subgoal *G* and a plan *P* is selected to achieve it in the next

deliberation cycle, then the first action or subgoal in P is set to be the current step of the goal-plan tree. If the current step is an action A and A is selected for execution in next deliberation cycle, then the action that is currently executing is set to be A, and the start time of A is set to be the current deliberation cycle. The action or subgoal following A in the same plan P will be set to be the current step. If there is no such action and subgoal, then the current step is set to be the action or subgoal following the parent goal of the plan P. If there is an action in the goal-plan tree that is currently executing and is expected to finish its execution in the next deliberation cycle, then the action that is currently executing is set to *null* and the start time of this action is set to -1. If A is the last action in the goal-plan tree, then the completion time of the goal-plan tree is set to be the next deliberation cycle.

- `boolean progressMerge();`

  This operation is used when the current step of the goal-plan tree (an action A) is merged with other actions. The action or subgoal following A in the same plan P will be set to be the current step, and the total number of merged steps in this goal-plan tree is increased by one.

We also defined an abstract data type for MCTS nodes, and all the operations that can be performed on each MCTS node are shown as follows:

- `ArrayList<gpt> getGPTs();`

  This operation returns the set of goal-plan trees in the node.

- `Env getEnv();`

  This operation returns the current state of the environment in the node.

- `int getVisited();`

  This operation returns the number of times this node has been visited.

- `double getValue();`

  This operation returns the total simulation value of the node.

- `double getBestValue();`

  This operation returns the best simulation value of this node.

- `ArrayList<node> getChildren();`

  This operation returns the child nodes of this node.

- `node getParent();`

  This operation returns the parent node of this node. If the node is the root node, this operation returns *null*.

- `int getCompletedGoals();`

  This operation returns the number of top-level goals which are already achieved.

- `int getMerged();`

  This operation returns the number of actions that are not performed (saved) due to the synergies between intentions.

- `step getPreviousStep();`

  This operation returns the step which leads the parent node of this node to the current node.

- `gpt getPreviousGPT();`

  This operation returns the goal-plan tree progressed in the previous cycle (for this node).

- `void visitedInc(int n);`

  This operation increases the number of times the node has been visited by `n`.

- `void valueInc(double n);`

  This operation increases the total simulation value of this node by `n`.

- `void setBestValue(double n);`

  This operation is used to set the best simulation value of this node.

- `void setChildren(ArrayList<node>);`

  This operation is used to set the child nodes of this node.

- `void setParent(node);`

  This operation is used to set the parent nodes of this node.

Algorithm 3 Return a step to execute at current deliberation cycle

1: $I \leftarrow \{t_1, \ldots, t_n\}$
2: **procedure** $SAM(I, Env, f_\mu, f_a, f_b, \alpha, \beta)$
3: $n_0 \leftarrow node(I, Env)$
4: **for** $i \leftarrow 1, \alpha$ **do**
5: $n_e \leftarrow$ SELECT(n_0, f_a)
6: EXPAND(n_e)
7: $n_s \leftarrow random(n_e$.`getChildren()`$)$
8: $V \leftarrow \emptyset$
9: **for** $j \leftarrow 1, \beta$ **do**
10: $n_t \leftarrow$ SIMULATE(n_s)
11: $V \leftarrow V \cup f_\mu(n_t)$
12: $V_{best} \leftarrow max(V)$
13: BACKUP(n_s, V_{best})
14: **return** BEST-NEXT-STEP(n_0, f_b)

The *SAM* scheduler takes seven parameters as input: a set of goal-plan trees $\{t_1, \ldots, t_n\}$ corresponding to the agent's current intentions I, the current state of the environment and the agent Env, three functions f_μ, f_a and f_b, the

number of iterations to be performed α, and the number of simulations to be performed at each iteration β. The parameters f_μ, f_a, f_b, α and β allow the agent developer to configure *SAM* for a particular application. f_μ is a function which returns a utility value representing the performance of the agent (see Chapter 3). f_a is a function to calculate the UCB value for a node in the selection phase. Given two nodes n_i and n_j, function f_b returns true if node n_i is more promising than node n_j. Otherwise, f_b returns false. Function f_b is used to select the best child node of the root node in the search tree, when the algorithm terminates.

α and β together represent the computational budget of the algorithm, i.e. how many iterations should be run by the scheduler and how many simulations should be run at each iteration. Finally, the *SAM* scheduler returns which current step pointer to advance and how to advance it (i.e., if the current step is a subgoal).

The scheduling algorithm starts with the root node of the search tree n_0 which represents an agent's current intentions I and the current state of the environment and the agent Env, and iteratively runs α iterations to build a Monte-Carlo search tree. The steps in each iteration correspond to 4 phases in Monte-Carlo Tree Search, which are selection, expansion, simulation and back-propagation. Line 5 in Algorithm 3 corresponds to the *selection* phase in (SP-)MCTS, where a leaf node, n_e is selected for expansion. The function SELECT (see Algorithm 4) requires two parameters as input: a node n, and the function f_a, and returns a leaf node in the search tree which has the largest f_a value compared to its siblings. The function SELECT recursively checks if n is a leaf node. If it is, then n is returned. If it is not, f_a is called to calculate

the UCB values for all its child nodes, and n is assigned to the child node which has the largest UCB value. *head* takes a list as input and returns the first element in the list. Conversely, *tail* is called to return the list of remaining elements.

Algorithm 4 Return the selected leaf node

```
1: function SELECT(n, f_a)
2:     if n.getChildren() ≠ ∅ then
3:         n' ← head(n.getChildren())
4:         for each n'' ∈ tail(n.getChildren()) do
5:             if f_a(n'') > f_a(n') then
6:                 n' ← n''
7:         SELECT(n', f_a)
8:     return n
```

Line 6 in Algorithm 3 corresponds to the expansion phase in (SP-)MCTS. The function EXPAND generates all the child nodes of the selected node n_e. Each child represents the environment state (and the current step of each goal-plan tree) resulting from progressing one step in one intention in n_e. For a given node n, function EXPAND (shown in Algorithm 5) checks the current step (`getCurrentStep()`) of each goal-plan tree t_i. If the current step of t_i (ns) is an executable action, then a set of child nodes representing executing ns in each t_i in the current deliberation cycle is created and added to the set of new nodes N. An action ns is executable if there is no action in t_i that is currently executing, and the precondition of A_i holds in the current environment (i.e., in node n). Before executing ns, the function EXPAND also checks if there are any actions cs in intention t_j ($j \neq i$) that are executable and have the same postcondition as ns. If there are such actions, the function EXPAND will merge action ns and all these actions cs to save execution steps[3]

[3]The details of merging actions is discussed in Section 7.2.

(function *samePost(act1,act2)* returns true if actions $act1$ and $act2$ have the same postcondition, and false otherwise). In the new nodes, t_i is advanced one step further by executing action *ns* (see line 16). For each $t_j (j \neq i)$, if its current step (*cs*) is an executable action that has the same postcondition as *ns*, then *cs* is merged with action *ns* and t_j is advanced one step by the operation `progressMerged()` (see line 12). Finally, the state of the environment is updated by applying the postconditions of the actions (*Act*) which finish their execution in the next deliberation cycle. The actions that will finish their execution in the next deliberation cycle (*Act*) are returned by the operation `getCompletedAction()`[4]. As there may be fallible action in *Act*, the function *apply* returns a set of possible environment states in the next deliberation cycle. For each of the possible environment state, the algorithm generates a new node $n^{'}$ and adds it to the set of new nodes *N*.

If the current step of t_i is a (sub)goal, for each of its relevant plans that is executable in *n*, a set of child nodes is created and added to the set of new nodes *N*. A plan is executable if its precondition holds in the current environment, and there is no action in t_i that is currently executing. In all the new nodes n', the current step in goal-plan tree t_i is set to the first step in the selected plan (by calling the operation `progress()`) and the environment is updated by applying the postconditions of the actions which finish their execution in the next deliberation cycle. Similarly, the actions that finish their execution in the next deliberation cycle (*Act*) are returned by the operation `getCompletedAction()`, and for each of the possible environment states, the algorithm generates a new node $n^{'}$ and adds it to the set of new nodes *N*.

[4]Note, SAM does not consider errors in the estimation of execution time in its expansion phase

Finally, when all possible next steps are checked, the set of new nodes N are set to be the child nodes of n.

Lines 7-11 in Algorithm 3 correspond to the simulation phase in (SP-)MCTS. In line 7, one of the newly created child nodes, n_s, is selected at random for simulation. The algorithm then runs β random and unbiased simulations from node n_s until a terminal node n_t is reached (line 9-11). A terminal node doesn't have any child nodes and cannot be expanded any further. The terminal node n is then returned. In line 11, the utility value of the terminal node is calculated by calling the utility function f_μ, and then stored in the set V.

Finally, the best simulation result (the utility value of the terminal node) V_{best} in V is *back-propagated* from n_s to all nodes on the path to the root node n_0 (line 13). The function BACKUP takes a node n and the utility value V_{best} as input, and recursively updates the statistics of n and all its ancestors. The statistics of a node include the number of times the node has been visited, the total simulation results and the best simulation results of node.

After α iterations, the *SAM* scheduler terminates and returns the best next step to perform at the current deliberation cycle (line 14). The function BEST-NEXT-STEP compares the child nodes of the root node n, and returns the edge (by calling the operation `getPreviousStep()`) between the root node and its best child node representing the best next step to progress in the current deliberation cycle. The best next step is then progressed (i.e., by calling the operation `progress()`), and the current intention of the agent is updated accordinately.

Algorithm 5 Return the set of expanded child nodes

```
1: function EXPAND(n)
2:     N ← ∅
3:     I' ← n.getGPTs()
4:     for each t_i ∈ I' do
5:         ns ← t_i.getCurrentStep()
6:         if ns instanceof ACTION and t_i.executable(n.getEnv()) then
7:             Act ← ∅
8:             for each t_j ∈ I' and j ≠ i do
9:                 if t_j.executable(n.getEnv()) then
10:                    cs ← t_j.getCurrentStep()
11:                    if cs instanceof ACTION and samePost(ns, cs) then
12:                        t_j.progressMerge()
13:                else
14:                    Act ← Act ∪ t_j.getCompletedAction()
15:                    t_j.progress()
16:            t_i.progress(ns)
17:            envs ← ∅
18:            envs ← apply(n.getEnv(), Act)
19:            for each env' ∈ envs do
20:                n' ← node(I', env')
21:                N ← N ∪ n'
22:        if ns instanceof GOAL then
23:            for each p_i ∈ n.getApplicablePlans() do
24:                Act ← ∅
25:                for each t_j ∈ I' do
26:                    if t_j ≠ t_i then
27:                        Act ← Act ∪ t_j.getCompletedAction()
28:                        t_j.progress()
29:                    else
30:                        t_j.progress(p_i)
31:                envs ← ∅
32:                envs ← apply(n.getEnv(), Act)
33:                for each env' ∈ envs do
34:                    n' ← node(I', env'(n))
35:                    N ← N ∪ n'
36:    n.setChildren(N)
37:    for each n_i ∈ N do
38:        n_i.setParent(n)
```

Algorithm 6 Run simulation from a given node and return a leaf node representing the terminal state

```
1: function SIMULATE(n)
2:     N ← EXPAND(n)
3:     if N ≠ ∅ then
4:         n ← random(N)
5:         SIMULATE(n)
6:     return n
```

Algorithm 7 Back-propagating the simulation values

```
1: procedure BACKUP(n, V_best)
2:     if n ≠ null then
3:         n.visitedInc(1)
4:         n.valueInc(V)
5:         if V > n.getBestValue() then
6:             n.setBestValue(V)
7:         BACKUP(n.getParent(), V)
```

Depending on the configuration of the scheduler, *SAM* is able to solve a wide range of intention progression problems. Moreover, it schedules the progression of intentions at the level of individual actions, rather than the plan level.

Algorithm 8 Return the step which leads to the best child node

```
1: function BEST-NEXT-STEP(n, f_b)
2:     n_best ← head(n.getChildren())
3:     for each n' ∈ tail(n.getChildren()) do
4:         if f_b(n', n_best) then
5:             n_best ← n'
6:     return n_best.getPreviousStep()
```

4.4.1 *SAM* and Agent's Deliberation Cycle

SAM implies some modification to the standard BDI deliberation cycle described in Chapter 2. It effectively merges the plan selection and intention selection of the standard deliberation cycle into a single process, which is

responsible for determining both which plan to adopt for a particular (sub)goal, and which step of which intention to execute at this cycle. Only the choice of which top-level goals to adopt falls outside the scope of *SAM*: once a top-level goal is adopted, *SAM* determines which plans to execute and how these plans should be interleaved so as to generate the best way of progressing intentions. When a subgoal is posted, *SAM* explores through sampled pseudo-random simulation, the implications of intending all relevant applicable plans for the subgoal (and the possible subplans of those plans) and their possible interleavings which represents all possible ways of progressing the agent's intentions.

As such it assumes responsibility for decisions that some agent platforms traditionally leave to the developer, e.g., the S_O and S_I selection functions in Jason [9] allow a developer to customise the plan and intention selection for a particular application domain. While such customisation may be necessary or desirable for some problems, it requires specialist expertise on the part of the developer to program at the level of the interpreter rather than writing BDI agent code. We believe the *SAM* scheduler can be beneficial for many problems, both in reducing developer effort (the developer no longer has to anticipate and control possible interactions between intentions using atomic constructs, for example) and in improving the performance of agent programs. (We provide empirical evidence that *SAM* outperforms current approaches to intention selection in different problem domains in Chapters 5, 6 and 7.)

SAM does entail additional developer effort in defining the pre- and post-conditions of actions. However alternative approaches to avoiding conflicts between intentions, e.g., SI, require similar information in the the form of pre-

and postconditions of plans and goals. Some BDI programming languages already require the developer to specify the pre- and postconditions of actions, e.g., basic actions in GOAL [34], belief-update actions in 2APL [23]. For languages such as Jason, our experimental results indicate the kinds of scenarios where the additional effort of specifying the pre- and postconditions of actions would be worthwhile in terms of increased agent performance.

Chapter 5

Avoiding Conflicts Between Intentions

In this chapter, we show how the *SAM* scheduler can be used to avoid conflicts between an agent's intentions. We evaluate the performance of our approach in both synthetic domains and a more realistic scenario based on the Miconic-10 elevator domain [40]. The results suggest the *SAM* scheduler outperforms existing approaches to intention progression both in terms of goals achieved and the variance in goal achievement time.

5.1 Avoiding Conflicts

As discussed in Chapter 1, when an agent pursues multiple goals in parallel, the interleaving of the execution steps in different intentions may result in conflicts. The conflicts can arise from the limited availability of resources, i.e., different plans are competing for the same resource, or be caused by breaking the dependency links between plans. Thangarajah et al. [60, 56] defined the

link between two plans (in the same intention) where the postcondition [1] of the first plan makes the precondition of the second true as a *dependency link*. The postcondition of the first plan is called a *preparatory-effect (p-effect)*. The plan that is dependent on a preparatory-effect (i.e., the second plan) is termed the *dependent-plan*. However, the definition of p-effect in [60, 56] only works for plans. We extend their definition of p-effect to include the establishment of the precondition of an action by a preceding action (in the same intention). The action that is dependent on a preparatory-effect, is thus termed the *dependent-action*. For example, in Figure 3.2, the precondition of action A_0 (i.e., At(*lift, f0*)) is established by executing action A_2. Therefore, the postcondition of action A_2 (i.e., At(*lift, f0*)) is a p-effect and action A_0 is a dependent-action. Executing any actions that make At(*lift, f0*) false before A_0 is executed will results in conflicts. We define the *size of a p-effect* as the number of possible interleaving points between the action that establishes a precondition and the later step which requires the p-effect as its precondition. For example, the size of the p-effect: At(*lift, f0*) in the previous example is 1, as there is only one point at which a step from another intention can be interleaved.

As discussed in Chapter 2, one possible way to avoid conflicts between intentions in practical agent programming languages like Jason and 2APL, is to use atomic constructs to force the agent not to interleave execution steps in different intentions. If an instance of a plan is atomic and is selected for execution, then this intention will be selected for execution continuously until it is finished. However, it is difficult for programmers to ensure that all potential interactions

[1] In their definition of goal-plan trees, plans have postcondition.

between plan steps are encapsulated within atomic constructs, and disallowing the interleaved execution of intentions may reduce the responsiveness of the agent (and the fairness of the schedule) to an unacceptable degree, i.e., the agent will focus on one intention until it finishes, which delays the execution of all other intentions. In [60, 56], the conflicts which are caused by the limited availability of resources and the conflicts which are caused by breaking the dependency links are considered as different problems and are solved using different kinds of summary information. Moreover, as discussed in Chapter 2, the summary-information-based approach is limited to interleaving intentions at the plan level. In some cases, conflicts can only be resolved at the action level. Interleaving intentions at the plan level also reduces the fairness of the schedule (i.e., the responsiveness of the agent).

The *SAM* scheduler is used to solve both kinds of conflicts (i.e., resource conflicts and the conflicts which are caused by breaking the dependency links between plans and actions) at the action level. The production and consumption of resources are considered as the pre- and postconditions of actions. For example, if a laptop costs 500 pounds, then the precondition of buying a laptop is having more than 500 pounds, and its postcondition is to pay 500 pounds.

5.2 Problem Specification and *SAM* Configuration

In this chapter, we are focusing on resolving conflicts between intentions. For simplicity, we assume the deadline and preferred achievement time for all goals are infinity, and the execution of actions are instantaneous, i.e., all actions require one deliberation cycle to finish their execution. We will discuss intention progression with deadlines and preferred achievement time in the

next chapter. In some problem domains, achieving the largest number of highest priority goals may be required as in [63]. Incorporating priorities in *SAM* is straightforward. However, in this thesis, we assume that all goals have the same priority for simplicity.

To make the problem of avoiding conflicts precise, we consider the number of top-level goals achieved as a key metric for BDI agents which indirectly measures the number of conflicts in the agent's interleaving of its intentions. We assume that an agent fails to achieve its goals primarily because of the conflicts between its intentions. Thus, the more top-level goals an agent has achieved, the fewer conflicts were in the agent's interleaving of its intentions. As discussed, the responsiveness of the agent is also important. We therefore define the fairness of an interleaving as a subsidiary objective in the problem. We consider an interleaving to be more fair if it minimises the variance in the elapsed time of each successful intention (subject to achieving as many goals as possible).[2] The elapsed time r of an intention is defined as $r = t - s$, where s is the cycle at which the top-level goal of the intention is adopted by the agent, and t is the cycle at which the goal is achieved. The *variance in elapsed time*, v, of a set of n intentions with elapsed times $r_1, \ldots, r_n$ is given by $v = \sum_{i=1}^{n} (r_i - \overline{r})^2 / n$, where $\overline{r}$ is the average elapsed time. Fairness is then defined as in the Equation 5.1, where v is the variance in elapsed time, and n is the number of finished intentions.

$$Fairness = \begin{cases} 0, & \text{if } n == 0 \text{ .} \\ 1 \text{ / v}, & \text{otherwise.} \end{cases} \tag{5.1}$$

[2] An alternative definition of fairness is used as an evaluation criterion in [67].

The problem of avoiding conflicts is therefore defined as: given a set of goal-plan trees with their corresponding current step pointers, representing the current intentions of the agent $\{t_1, \ldots, t_n\}$, a set of condition variables *Env* representing the current state of the environment and the agent, return a next step of a goal-plan tree t_i to be executed at each deliberation cycle such that the number of goals achieved by the agent is maximised, and the fairness of the interleaving is also maximised, subject to achieving as many goals as possible.

To produce a schedule which has fewer conflicts and is more fair, the function f_a in *SAM* is configured to return the UCT^{SP} values for a given node (see Equation 4.3), and the function f_μ is defined as follows:

Algorithm 9 Utility function f_μ

```
1: function f_μ(n)
2:     return n.getCompletedGoals() + FAIRNESS(n)
```

where n.`getCompletedGoals()` is the number of goals achieved in node n, and FAIRNESS(n) returns the fairness of achieving these goals as defined in Equation 5.1. The utility of a node is the sum of the number of goals achieved in this node and the fairness of achieving these goals. As the value of fairness is within the range $[0, 1]$ and the number of goals achieved is an integer, the utility function f_μ ensures that the number of goals achieved is always more important than the fairness of achieving the goals (i.e., achieving more goals is the first priority).

When the environment is static, the comparator function f_b is defined as follows:

Where n_i and n_j are two child nodes of the root node, n_i.`getBestValue()` and n_j.`getBestValue()` are the best simulation results in node n_i and n_j

Algorithm 10 function f_b

```
1: function f_b(n_i, n_j)
2:     if n_i.getBestValue() > n_j.getBestValue() then
3:         return True
4:     else
5:         return False
```

respectively. The function f_b compares two nodes on their best simulation results (i.e., f_b returns true if the first node has a larger best simulation result than the second, and false otherwise).

When the environment is dynamic, we assume the *SAM* scheduler has no knowledge of the Poisson means and the possible changes of the environment, and assumes a static environment during the simulation phase. The comparator function f_b is defined as follows:

Algorithm 11 function f_b

```
1: function f_b(n_i, n_j)
2:     t_i ← n_i.getPreviousGPT()
3:     t_j ← n_j.getPreviousGPT()
4:     if Floor(n_i.getBestValue()) > Floor(n_j.getBestValue())
   then
5:         return True
6:     if Floor(n_i.getBestValue()) == Floor(n_j.getBestValue())
   and Coverage(t_i) < Coverage(t_j) then
7:         return True
8:     if Floor(n_i.getBestValue()) == Floor(n_j.getBestValue())
   and Coverage(t_i) == Coverage(t_j)
9: and n_i.getBestValue() − Floor(n_i.getBestValue()) >
10: n_j.getBestValue() − Floor(n_j.getBestValue()) then
11:         return True
12:     return False
```

Where n_0 is a root node, n_i and n_j are two child nodes of the root node. t_i is the goal-plan tree progressed in the previous cycle which lead the parent node of n_i to n_i (i.e., progressing one step in t_i leads the root node n_0 to n_i. Similarly, t_j is the goal-plan tree progressed in the previous cycle which lead the root node

n_0 to n_j. `Coverage`(t_i) returns the coverage of the goal-plan tree t_i. `Floor()` is the same as `Math.floor()` in Java which returns the largest integer that is less than or equal to the input argument. Therefore, `Floor(`n_i`.getBestValue())` represents the number of goals achieved in the best simulation of node n_i, and n_i`.getBestValue()` − `Floor(`n_i`.getBestValue())` represents the fairness in the best simulation of node n_j. The comparator f_b compares two nodes, first on the number of goals achieved in their best simulation, secondly on the coverage (intentions with lower coverage are preferred) and thirdly the fairness value in their best simulations.

SAM therefore adopts a similar approach to [59, 66, 67] in dynamic environments, in prioritising intentions which are more likely to become unexecutable due to changes in the environment. However, unlike previous coverage-based approaches, it selects the intention to be progressed at the action, rather than the plan level, interleaving actions in a manner that is most likely to achieve the largest number of goals, and adaptively prioritises fairness in low dynamism environments. In the following experiments, we first consider the effect of increasing dynamism and reduced coverage.

5.3 Evaluation

In this section, the performance of *SAM* scheduler is compared with existing approaches to intention progression in practical agent programming languages, i.e., First-In-First-Out and Round-Robin scheduling (see Chapter 2), as well

as in the theoretical literature, i.e., summary-information-based approach and coverage-based approach

First-In-First-Out (FIFO) FIFO is one of the default intention selection strategy in JACK [68], which executes each of the agent's intentions to completion (the goal is achieved or the next step in the intention cannot be executed) before starting to execute the next intention (this is termed a *non-interleaved execution strategy* in [4]). Intentions may be executed in some fixed order (e.g., in order of priority), or the order of execution may be arbitrary. In the experiments below, we assume for simplicity that the agent's intentions are held in a queue and are executed in first-in-first-out order. FIFO scheduling has the advantage that it minimises conflicts between intentions. If the preconditions of plans for each top-level goal are disjoint, and the preconditions for all subplans (and actions) are established by preceding steps in the intention, then, if a set of intentions can be executed at all in a given environment, they can be executed in FIFO order. However, it has the disadvantage that the achievement of some goals may be significantly delayed compared to other goals. In many situations, it is desirable that the agent progresses all its intentions at approximately the same rate (i.e., achieves its top-level goals in parallel).

Round-Robin (RR) While FIFO scheduling minimises interactions between intentions, RR attempts to ensure 'fairness' between intentions by executing a fixed number of steps of each intention in turn. For example, the agent may iterate over its intentions, executing one step of the next intention in sequence at each deliberation cycle. Round-Robin is the default scheduling strategy for popular agent programming languages such as Jason [9]; some platforms, like JACK [68] also allow the programmer to choose between RR

and FIFO. The disadvantage of Round-Robin scheduling is that it increases the number of possible conflicts between intentions: even if in each intention the preconditions for all subplans (and actions) are established by preceding steps in the intention, the interleaving of steps in different intentions may destroy a precondition established by a step in another intention before the action that requires the precondition is executed.[3] Such conflicts between intentions can sometimes be avoided by using *atomic* constructs that prevent the interleaving of actions in one plan with actions from other plans. However, it is difficult for the programmer to ensure that all potential interactions between plan steps are encapsulated within atomic constructs, and disallowing the interleaved execution of intentions may reduce the responsiveness (and the fairness) of the agent to an unacceptable degree. In the extreme case when all plans are atomic, RR performs exactly the same as FIFO.

Summary-Information-Based Approach (SI) As discussed in Chapter 2, summary-information-based approach is an approach to avoid conflicts and to exploit synergies between intentions proposed by Thangarajah et al. [60, 57, 58, 56]. In SI, summary information about pre-, in- and postconditions is propagated up each goal-plan tree to allow reasoning about interactions between trees. For example, if there are two possible plans for a goal g which both have p as a postcondition, then it is possible to infer that p will definitely occur as a result of achieving g, and that any other plans that brings about $\neg p$ may cause a conflict. SI uses this information to protect active p-effects and in-conditions. If adopting a new goal will result in a conflict, i.e., a postcondition

[3]For simplicity, we assume interleaved execution of intention steps; however the same problems occur if the agent executes a step of each intention in parallel at each cycle as in, e.g., 2APL.

of the new goal destroys an active p-effect or in-condition, then progression of the intention containing the new goal must be delayed until the p-effect or in-condition is no longer active.

Coverage-Based Approach (C) The Coverage-based approach schedules the progression of an agent's intention based on the notions of *coverage* and *overlap* [59, 66, 67]. It prioritises the progression of intentions with lowest coverage, as they are more likely to fail due to changes in the environment. Waters et al. [66, 67] implemented and evaluated the approach proposed in [59] (which they call c0), and a preemptive variant of c0 which they call c1. c0 executes the progressable intention with lowest coverage and maintains focus as long as possible. Only when the current intention is finished or blocked will c0 recalculate coverage for all intentions and change focus to the progressable intention with lowest coverage. In c1, whenever a sub-goal is posted, the coverage of each intention is recalculated and its progressibility determined. As a result, c1 will change focus to a lower coverage progressable intention, even if the current intention is progressable. Like SI, C interleaves intentions at the plan level.

The performance of each approach was evaluated on two criteria: the number of goals achieved, and the variance in elapsed time (i.e., the fairness of the schedule). We only report variance results where the number goals achieved is > 8; as the number of intentions which complete successfully decreases, the variance value becomes less informative, and conflates the ability to achieve goals with the fairness of intention selection. For example, intentions that fail increase the variance of RR and decrease the variance of FIFO.

For all the experiments reported below, *SAM* is configured to perform 100 iterations ($\alpha = 100$) and 10 random simulations at each iteration (β = 10) and all the scheduling approaches do not make use of failure handling. With failure handling, the performance of agents is improved, as they can retry the failed intentions. However, retrying the failed intentions requires additional execution time. In cases where the agent fails to achieve its goals very often, the additional execution time to recover from execution failure is singificant. In addition, there are several approaches to recovering from intentions that can be used with each scheduling approach, and the additional execution time required for a recovery technique differs for different scheduling approaches. To focus on "pure" scheduling performance and avoid conflating the success of and time required by a recovery technique with that of the scheduling approaches, the scheduling approaches do not make use of failure handling. We consider failure recover in Section 7.1 .

5.3.1 Synthetic Domain

In the interests of generality, our first evaluation is based on sets of randomly-generated, synthetic goal-plan trees representing the current intentions of an agent in a simple environment. By controlling the characteristics of the trees, and the dynamism of the environment, we can evaluate the performance of each scheduling algorithm under different conditions.

Environment

The environment is defined by a set of propositions that may appear as pre-condition of plans and pre- or postconditions of actions in a goal-plan tree.

Each environment variable is modelled as a Poisson process with specified mean, allowing the frequency with which the environment changes the value of the variable to be controlled. For a static environment, the means of all Poisson processes are set to 0. Each trial consists of a randomly generated environment and a randomly generated set of goal plan trees (each generated with the same parameter values). The variables appearing as conditions in each tree are randomly selected from the environment variables. Execution of plans and actions in different trees therefore may interact through shared environment variables, and a change in the value of a variable may affect the selection of plans or the executability of actions in more than one tree.

Synthetic Goal-Plan Trees

The set of synthetic goal-plan trees are generated by a package called GenGPT (see Appendix A). The arguments required to generate the set of goal-plan trees are: the depth of the goal-plan tree $\#\delta$, the number of subgoals in each non-leaf plan (leaf plans contain only action nodes) $\#\gamma$, the number of plans to achieve a goal $\#\pi$, the number of actions in each plan $\#\alpha$, the maximum p-effect size S, the number of environment variables that may appear in the tree as pre- and postconditions $\#v$, the number of goal-plan trees $\#T$ and the output file path *Path* to which the forest of goal-plan trees is saved. The flag for each argument, their default values and the constraints on each argument are shown in Appendix A.

The depth of the goal-plan tree $\#\delta$, the number of subgoals $\#\gamma$ and actions $\#\alpha$ in each plan and the number of plans to achieve each goal $\#\pi$ determine

the shape of the goal-plan trees in the forest.[4] The maximum p-effect size of each goal-plan tree $\#S$ affects the difficulty in interleaving the goal-plan trees, i.e., with a larger p-effect size, interleaving goal-plan trees is more likely to cause conflicts. Finally, the number of environment variables $\#v$ represents the likelihood of actions and plans in different goal-plan trees having the same pre- and postconditions. By varying the number of environment variables we could vary the probability of both conflicts and synergies between goal-plan trees.

The set of goal-plan trees is saved in XML format (see Appendix A). We define pre- and postconditions as a set of literals. Each literal is denoted by an $(index, boolean)$ pair, where *index* is an identifier of the propositions and *boolean* is either *True* or *False*.

The GenGPT application iteratively generates goals, plans, and actions from the top-level goal to the leaf nodes in the tree. It first generates the top-level goal, and then generates the plans to achieve the top-level goal, and then generates a sequence of actions and subgoals for each of these plans and so on. For each goal, $\#\pi$ plans are generated. The precondition of each plan are established by the postcondition of a previous step in the goal-plan tree (i.e., p-effects). If there is no such p-effect (i.e., the plans to achieve the top-level goal), a proposition i is randomly selected from the set of environment variables, where i is an index. The odd numbered plans to achieve the goal have the literal $(i, True)$ as their precondition, and the even numbered plans have $(i, False)$ as their precondition. This ensures that if $\#\pi > 1$, there is at least one plan applicable to achieve each (sub)goal.

[4] It would be more flexible to allow specification of the maximum and minimum number of actions and subgoals in each plan; we assume a fixed number of actions and subgoals for simplicity.

Each plan consists of an ordered sequence of actions and subgoals. The actions in the plan are generated before the subgoals. The first action in the plan has the plan's precondition as its own precondition. The remaining actions either have the plan precondition as their own preconditions or their preconditions are established by previous actions in the plan. The postcondition of an action is selected randomly from the set of environment variables. When all actions in a plan have been generated, and the plan is not a leaf plan (i.e., the plan is not in depth $\#\delta$), we also generate $\#\gamma$ subgoals in this plan by using the same approach mentioned above.

We keep creating actions, plans and (sub)goals until we reach the depth of the tree. The generation process will stop after we generate the last leaf action, and the whole goal-plan tree is translated and saved in an XML file. The synthetic goal-plan trees generated by GenGPT have the following properties:

- Plans are well formed, i.e., if the precondition of a plan holds in the environment, then this plan can be successfully executed.

- Taken individually, each goal-plan tree is executable, i.e., there is at least one way to achieve the top-level goal in all (static) environment, if all the goals in the goal-plan trees have at least two plans to achieve it.

An example goal-plan tree generated by GenGPT with parameters: $\#\delta = 2$, $\#\gamma = 1$, $\#\pi = 2$, $\#\alpha = 3$, § = 5, $\#\nu = 100$, $\#T = 1$, $Path$ = example-gpt.xml is shown in Figure A.1). In Figure A.1, goals are represented using red ovals, plans are blue rectangles and actions are green triangles. For legibility, we didn't show the name , pre- and postcondition of actions and plans. In the experiments reported below, we vary the values of the parameters introduced above to create different goal-plan trees for different scenarios.

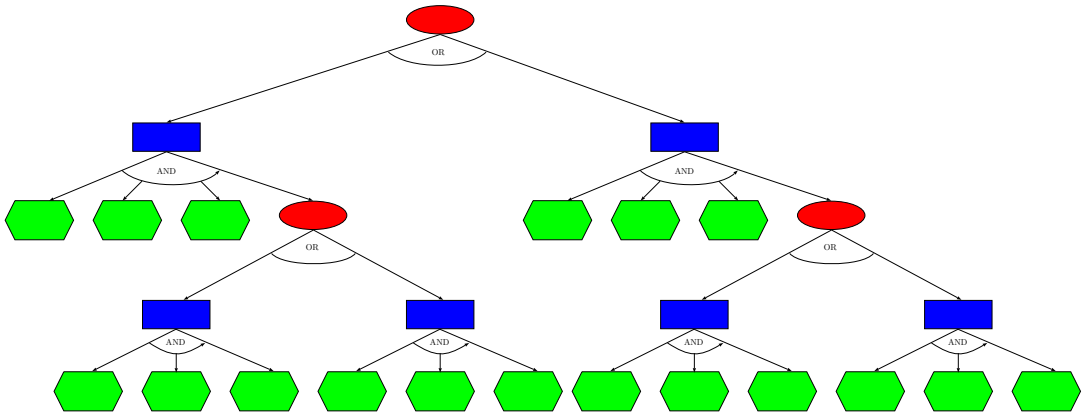


Fig. 5.1 An example of the synthetic goal-plan tree.

For each experiment reported below, we generated 50 sets of 10 goal-plan trees. The depth of each goal-plan tree is set to 5. Each (sub)goal has one or two plans to achieve it. Each plan consists a sequence of three or five actions, followed by a subgoal. The maximum p-effect size is either 5 or ∞. The number of environment variables is set to 20. Note, in some experiments reported below, the GenGPT trees were post-processed such that the trees can be controlle more precise.

Static Environment

The first set of experiments evaluate the performance of FIFO, RR, SI and *SAM* in a static environment (C is designed primarily for dynamic environments, so we only evaluate it in dynamic environment). The means of all Poisson processes are set to 0.

Experiment 1 In the first experiment, each goal has two plans to achieve it, each plan contains 3 actions, and the maximum p-effect size is 5. For trees of depth 5, this gives an average p-effect size of 3 at level 5 (i.e., on average, the precondition of a plan at level 5 is established by an action executed three steps earlier, in the parent of the plan). We vary the percentage of plan preconditions

established by p-effects (% p-effects) in each goal-plan tree from 33% to 100%. As discussed, the preconditions of the plans generated by GenGPT are always established by p-effects (i.e., % p-effects equals to 100%). In order to generate trees with % p-effects less than 100%, the plans in GenGPT trees were post-processed. We selected a fixed number of goals in each goal-plan trees, and replaced the preconditions of the plans[5] to achieve them by pairs of literals $(p, True)$ and $(p, False)$, where p is a variable randomly selected from the environment. For example, suppose π_0 and π_1 are two plans generated by GenGPT to achieve a goal g_0. The precondition of π_0 is $(r, True)$ and the precondition of π_1 is $(s, True)$. Both $(r, True)$ and $(s, True)$ are p-effects. If π_0 and π_1 are selected to post-process, then we replace $(r, True)$ by $(p, True)$, and replace $(s, True)$. by $(p, False)$, where p is randomly selected from the environment. By controlling the number of plans which needs to be post-processed in each goal-plan tree, we could control the % p-effects in each goal-plan tree. A goal-plan tree with a lower % p-effects has higher coverage (since the plans for g have no overlap), and therefore less potential for conflicts (since whatever the value of p, the agent has an applicable plan).

% p-effects	FIFO		RR		SI		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
33%	10	1856	3.04	-	10	291	10	17
66%	10	1856	3.02	-	9.88	320	10	20
100%	10	1856	2.82	-	9.72	373	10	19

Table 5.1 Goals & variance with increasing % p-effects

Table 5.1 shows the average number of goals achieved (#Goals) and variance (Var) in goal achievement time (smaller numbers are better) for each scheduling algorithm for each percentage of p-effects considered. The variance in goal

[5] As discussed, each goal has two plans to achieve it.

achievement time is calculated based on the start time (cycle) and finish time (cycle) of each top-level goals (see Section 5.2, page 92). For example, in Experiment 1, when a set of 10 top-level goals are scheduled in first-in-first-out order (i.e., column 2 and 3 in Table 5.1), the start time of all the top-level goals are cycle 0 and the finish time of these top-level goals are cycle 15, cycle 30, ..., cycle 150 respectively. Therefore the mean of the elapsed time is given by $\frac{(15-0)+(30-0)+...+(150-0)}{10} = 82.5$, and the variance in goal achievement time is $\frac{(15-82.5)^2+(30-82.5)^2+...+(150-82.5)^2}{10} = 1856$.

As can be seen, RR is able to achieve only 3.04 goals on average with 33% p-effects. This is because the precondition of actions and plans in the synthetic goal-plan trees are established by previous steps in the goal-plan tree. Even though 67% of subgoals have 100% coverage, there are still many points at which an action from another intention can destroy the precondition of a plan or an action. Without failure handling, RR cannot recover from execution failure, and thus has a very poor performance. With 33% p-effects SI is able to achieve all 10 goals, however as the percentage of p-effects increases, the number of goals achieved decreases slightly. FIFO and *SAM* are able to achieve all 10 goals even with 100% p-effects. However the variance in completion time is much higher with FIFO (1856) than with *SAM* (20).

Experiment 2 In the second experiment, each plan also contains 3 actions, and the maximum p-effect size is still 5. However, unlike in the first experiment, each goal may have one or two plans to achieve it. % 1-plan is used to denote the probability that a goal will have only one plan, i.e., 25% 1-plan means that for every (sub)goal in a goal-plan tree, there is a 25% chance that it has only one plan to achieve it. Increasing the probability that a goal has only one plan

reduces coverage, and increases the likelihood of conflicts between intentions, i.e., it is more likely the agent will have no applicable plan. The percentage of plan preconditions established by p-effects is set to 100% (i.e., all plans to achieve subgoals have their preconditions established by preceding actions); other parameters are the same as in Experiment 1.

% 1-plan	FIFO		RR		SI		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0%	10	1856	2.82	-	9.72	373	10	19
25%	8.92	1603	2.76	-	9.24	455	10	22
50%	7.92	-	2.44	-	8.88	475	10	27
75%	7.36	-	2.12	-	8.52	489	10	28

Table 5.2 Goals & variance with increasing % 1-plan (100% p-effects)

The results are shown in Table 5.2. As can be seen, for FIFO, RR and SI, as coverage decreases, the average number of goals achieved also decreases. This effect is particularly marked in the case of FIFO (if a top-level goal has only one plan, the postcondition of an action in a previously executed intention which destroys the precondition of the plan will make the intention unexecutable). *SAM* is able to achieve 10 goals at all coverage levels. For *SAM* and SI variance increases as coverage decreases. However *SAM* has significantly lower variance than SI for all coverage levels, as SI approximates FIFO when protecting p-effects.

Next, the performance of each approach is investigated when intentions are more complex and hence the likelihood of negative interactions is greater. In these experiments, the p-effect size and the number of actions in each plan is varied.

Experiment 3 In Experiment 3, the maximum p-effect size is set to ∞, i.e., the precondition of each plan is chosen from propositional variables appearing in

postconditions of any preceding actions in the tree. For trees of depth 5, this gives an average p-effect size of 5 at level 5 (i.e., on average, the precondition of a plan is established by an action executed five steps earlier, in the parent of the parent of the plan). As in Experiment 2, the probability of a goal having only a single plan is varied from 0% to 75%; all other parameters are the same as in Experiment 2.

% 1-plan	FIFO		RR		SI		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0%	10	1856	2.68	-	9.32	374	10	19
25%	9.08	1617	2.64	-	9.04	460	10	22
50%	7.94	-	2.20	-	8.60	471	10	25
75%	6.94	-	2	-	8.34	490	10	27

Table 5.3 Goals & variance with increasing % 1-plan (100% p-effects, average p-effect size 5)

The results are shown in Table 5.3. With a larger p-effect size, FIFO, RR and SI perform slightly worse in terms of goals achieved than in Experiment 2, while *SAM* is still able to achieve 10 goals in all cases. Variance for all approaches remains essentially the same as in Experiment 2. Goal-plan trees with a larger p-effect size are more difficult to schedule, as the agent needs to ensure the p-effects still hold when the corresponding dependent-plans/actions starts execution.

Experiment 4 In the fourth experiment, the number of actions in each plan is increased to 5. As in Experiment 2, the probability of a goal having only a single plan is varied from 0% to 75%; all other parameters are the same as in Experiment 2.

The results are shown in Table 5.4. As can be seen, the performance of FIFO, RR and SI decrease when plans have 5 actions, with a more marked

% 1-plan	FIFO		RR		SI		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0%	10	5156	1.64	-	9.28	544	10	40
25%	8.92	4564	1.56	-	9.00	755	10	58
50%	7.36	-	1.36	-	8.52	843	10	63
75%	6.52	-	1.32	-	8.30	874	10	66

Table 5.4 Goals & variance with increasing % 1-plan (100% p-effects, each plan contains 5 actions)

decline in the case of RR. SI shows a significant increase in variance. However, *SAM* still achieves all 10 goals, and the variance is increased which is much smaller than that for the other approaches.

Dynamic Environment

The second set of experiments evaluate the performance of FIFO, RR, *SAM*, and two variants of C from [66, 67], c0 and c1, in a dynamic environment. In the dynamic case, all the Poisson processes controlling the environment variables have the same non-zero mean. (In reality, different environment variables will change value at different rates, however for random trees this is a reasonable approximation.) The larger the mean, the greater the probability an environment variable will change value (from p to $\neg p$ or vice versa). Three different values for the mean are used: 0.01, which results in an average of 0.2 events (changes in the value of an environment variable) per agent cycle, 0.05 (which gives an average of 0.97 events per cycle), and 0.1 (which gives 1.91 events per cycle). In the dynamic case, the environment is updated to reflect the execution of the next step of the selected intention. The Poisson processes controlling the environment variables are then evaluated, possibly resulting in

additional changes to the environment. The resulting environment state is then 'sensed' by the agent at the beginning of the next deliberation cycle.

Experiment 5 In the fifth experiment, the sets of goal-plan trees were generated using the same parameters as in the first experiment in Table 5.2, i.e., all plans have two goals (0% 1-plan), %p-effects was 100%, each plan has 3 actions and the p-effect size at level 5 was 3. The means of Poisson process is varied from 0.01 to 0.1.

Dyn	FIFO		RR		c0		c1		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0.01	8.92	1625	2.24	-	9.52	1849	8.88	96	9.82	95
0.05	5.40	-	1.06	-	8.62	1830	8.64	94	9.40	95
0.10	3.28	-	1.00	-	8.16	1962	8.56	97	9.08	95

Table 5.5 Goals & variance with increasing dynamism (100% p-effects, 0% 1-plan)

The results are shown in Table 5.5. As can be seen, for all approaches, as the means of Poisson processes increase, the average number of goals achieved decreases. This effect is particularly marked in the case of FIFO. c0 performs better than c1 when the Poisson mean is small, and c1 has a better performance than c0 when Poisson means are larger. This is because c0 performs more like FIFO, while c1 tends to change intentions when a plan is complete, which is more like RR. As explained above, RR-like behaviour increases the number of possible conflicts between intentions. When the dynamism of the environment is low, the schedule produced by c1 is more likely to result in conflicts which c1 can't resolve. However c1 reacts more quickly to changes in coverage than c0, and in more dynamic environments this appears to be more important than reducing conflicts. *SAM* outperforms the other approaches in terms of the

number of goals achieved. The variance in goal achievement time of *SAM* is similar to c1 and significantly lower than c0.

Experiment 6 In Experiment 6, the probability of a goal having only a single plan is set to 25% (i.e., the second row in Table 5.2); the means of the Poisson processes are varied from 0.01 to 0.1; all other parameters are the same as in Experiment 2.

Dyn	FIFO		RR		c0		c1		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0.01	7.90	-	1.82	-	9.12	1877	8.40	116	9.74	114
0.05	4.74	-	1.14	-	7.98	-	8.26	118	9.28	120
0.10	2.54	-	0.72	-	7.64	-	8.22	124	8.94	123

Table 5.6 Goals & variance with increasing dynamism (100% p-effects, 25% 1-plan)

The results are shown in Table 5.6. As can be seen, for all approaches and levels of dynamism, with reduced coverage (25% 1-plan) the average number of goals achieved decreases and the variance in completion time increases. However, *SAM* still achieves the highest number of goals of any of any approach (between 8%–16% better than c1), and its variance in goal achievement time is comparable to c1.

Experiment 7 In Experiment 7, the maximum p-effect size is set to ∞. All other parameters are the same as in the first case of Experiment 2, and again the means of the Poisson processes are varied from 0.01 to 0.1.

Dyn	FIFO		RR		c0		c1		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0.01	8.88	1641	2.00	-	9.42	1863	8.60	100	9.80	96
0.05	5.34	-	1.02	-	8.48	1876	8.54	108	9.32	97
0.10	3.12	-	0.94	-	8.02	1978	8.48	110	8.98	96

Table 5.7 Goals & variance with increasing dynamism (100% p-effects, p-effect size 5)

The results are shown in Table 5.7. Compared to Experiment 5, the performance of c0 and c1 decreases, as increasing the p-effect size increases the number of possible conflicts between intentions. However, *SAM* still outperforms other approaches on the number of goals achieved and its variance in completion time is slightly lower than that of c1.

Experiment 8 In Experiment 8, the number of actions in each plan is increased to 5, all other parameters are same as the first case in Table 5.2 and the means of Poisson process is varied from 0.01 to 0.1.

Dyn	FIFO		RR		c0		c1		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0.01	8.40	4664	1.08	-	9.22	4605	8.32	360	9.52	346
0.05	3.76	-	0.62	-	8.18	4765	8.26	387	9.08	367
0.10	1.84	-	0.42	-	7.88	-	8.18	382	8.72	386

Table 5.8 Goals & variance with increasing dynamism (100% p-effects, action number 5)

The results are shown in Table 5.8. As can be seen, compared to the results in Experiment 5, the performance of FIFO, RR, c0, c1 and *SAM* decreases as we increase the number of actions in each plan. However, *SAM* still outperforms other approaches on the number of goals achieved and has a similar variance value compared to c1.

Overall, *SAM* outperforms FIFO, RR and SI on the number of goals achieved in a static environment. In simpler problems, FIFO and SI are able to achieve all 10 goals. However, as the problems become more challenging, their performance declines (FIFO more markedly than SI). *SAM* also has a significantly lower variance in elapsed time (particularly compared to FIFO), as it allows interleaving in action level. In dynamic environments, *SAM* outperforms FIFO, RR, c0 and c1 in terms of the number of goals achieved.

The variance in elapsed time time of *SAM* is comparable to that of c1, and significantly lower than that of c0. While the relative performance advantage of *SAM* in a dynamic environment is smaller than in a static environment, it consistently outperforms c1 in terms of number of goals achieved by between 5% and 10%.

5.3.2 Miconic-10 Elevator Domain

In this section the performance of *SAM* is evaluated in a more realistic scenario based on the Miconic-10 elevator domain [40].[6] Unlike traditional elevators, the Miconic-10 elevator allows passengers to enter their destination floor when calling the elevator using a 10-digit keypad installed in each elevator lobby. In the scenario the Miconic-10 elevator is controlled by a BDI agent program.

The methods and operators in the Miconic-10 Elevator HTN planning domain given in [40] were translated to goals, plans and actions in a BDI agent program using standard procedures for translating HTN planning problems into BDI agent programs, e.g., [27]. The goal-plan tree for the elevator program is shown in Figure 3.2.

In the experiments reported below, there is a single Miconic-10 elevator installed in a 10 floor building. Passengers may move from floor to floor using the elevator. Requests to travel to a particular floor give rise to top-level goals (one for each request) which are achieved by the BDI agent program. For simplicity, we assume there is no limitation on the number of passengers in the elevator at any one time.

[6] An elevator controller written in GOAL is evaluated in [2].

Results

The performance of *SAM* was evaluated in two scenarios. In the first experiment, all elevator requests are given initially. In this situation, the interactions between intentions are maximised (corresponding to the static environment in the synthetic domains). We generated 50 sets of 10 goals to travel between two randomly chosen distinct floors. *SAM* was configured as in Section 5.3.1.

FIFO		RR		SI		*SAM*	
#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
10	123	1.10	-	10	33	10	28

Table 5.9 Goals & variance for 10 goals

The number of goals achieved and variance in elapsed time (in parentheses) for FIFO, RR, SI and *SAM* are shown in Table 5.9. As can be seen, with the exception of RR, all approaches achieve 10 goals. However FIFO has significantly higher variance than SI and *SAM*, as FIFO executes the intentions sequentially.

In the second scenario, we use a more realistic model in which elevator requests arrive at specified intervals. Again there are 10 top-level goals in total. Two goals are given initially (at time 0) and the remaining top-level goals are posted while current intentions are executing. There are #g new top-level goals generated after executing each action, where #g varies from 0.33 to 3 (if #$g = 0.33$, there is a new top-level goal every 3 action steps. If #$g = 3$, there are 3 new top-level goals after executing one action). *SAM* was again configured as in Section 5.3.1.

The results are shown in Table 5.10. As in the first scenario, with the exception of RR, all approaches achieve 10 goals. However in the dynamic setting, c0

#g	FIFO		RR		c0 & c1		*SAM*	
	#Goals	Var	#Goals	Var	#Goals	Var	#Goals	Var
0.33	10	14	2.16	-	10	14	10	1
0.50	10	36	1.56	-	10	36	10	1
1	10	71	1.20	-	10	71	10	9
2	10	95	1.10	-	10	95	10	18
3	10	102	1.10	-	10	102	10	21

Table 5.10 Goals & variance with decreasing time between goals

and c1 have the same variance as FIFO (as the *MoveTo* subgoal has 100% coverage, achieving a *MoveTo* subgoal does not change the coverage of the current intention and so both c0 and c1 behave like FIFO). In contrast *SAM* has significantly lower variance (i.e., the waiting times for passengers is 'fairer'), as it is able to interleave actions in different intentions without destroying preconditions.

5.4 Summary

In this chapter, we show how *SAM* can be used to schedule the progression of an agent's intentions to maximise the number of goals achieved and the fairness of achieving these goals, subject to achieving as many goals as possible. We evaluate the performance of *SAM* in static and dynamic environments, both in synthetic domains and a more realistic scenario based on the Miconic-10 elevator domain [40]. Our results indicate that our approach out-performs RR, FIFO, SI and C, and is capable of successfully scheduling the progression of an agent's intentions even when the number of potential conflicts between intentions is high. In the next chapter, we will show how *SAM* can be used to schedule the progression of an agent's intention with real time constraints (e.g., deadlines).

Chapter 6

Intention Progression With Deadlines

In this chapter, we use the *SAM* scheduler to schedule the progression of an agent's intentions, where goals may have both preferred achievement times (the time by which the goal should ideally be achieved) and deadlines (the time by which the goal must be achieved). We evaluate the performance of the *SAM* scheduler in both synthetic domains and a more realistic domain based on the classic Blocks World domain. The experimental results suggest that *SAM* outperforms a state-of-the-art approach to intention progression with deadlines, AgentSpeak(RT) [63, 62], both in terms of the number of goals achieved and the total tardiness in achieving these goals.

6.1 Real Time Agency

Deciding which intention to progress next becomes more challenging in settings where goals must be achieved before a deadline. An interleaving of

steps in an agent's intentions that avoids conflicts, may still result in failure to achieve a goal by its deadline. In addition, if it is not possible to achieve a goal by the specified deadline, a rational agent should drop the goal, rather than waste effort attempting to achieve it. In the previous chapter, we used *SAM* to schedule the progression of an agent's intentions to avoid conflicts between intentions. However, temporal constructs were not considered and actions were assumed to be instantaneous. In this chapter, we evaluate *SAM* in a real-time environment, where a rational agent should select the next step of its current intentions to maximise the number of goals achieved by their deadlines.

In Chapter 3, we extended the notion of goals found in languages such as Jason [9] and 2APL [23], by introducing two temporal constructs: a preferred achievement time (soft deadline) p, and a deadline (hard deadline) d, where $p \leq d \leq \infty$. Recall that the *preferred achievement time* specifies the time by which a goal should ideally be achieved, and the *deadline* specifies the time by which a goal must be achieved. Goals which are not achieved by their (hard) deadline should be dropped. For goals where $p < d$, if the goal is not achieved by p, it is said to be *late*. If a top-level goal with preferred achievement time p is achieved at time $t \leq d$, we define the *tardiness* in achieving the goal as $max(0, t - p)$.

A top-level goal may only have a preferred achievement time and no deadline ($p < d = \infty$), e.g., the agent may have a goal to visit the gym this week; however if it's not possible to go this week, going next week (or later) is acceptable. A top-level goal may have only a deadline ($p = d < \infty$), e.g., a paper must be submitted before the conference submission deadline, otherwise

the goal to submit a paper to this conference should be dropped. A top-level goal may also have both a preferred achievement time and a deadline ($p < d < \infty$), e.g., submitting coursework after the due date will result in penalties, and submission is not possible after the model solution is published.

In order to schedule the progression of an agent's intentions in a real-time environment, we also need to know how much time it will take to execute each of the actions. We use *estimated execution time*, $e(a, \Phi)$, to represent the expected execution time of an action a with an agent's beliefs Φ. The estimated execution time may be specified by the programmer, e.g., the expected execution time of a move action may be based on the distance to be moved, or learnt from experience, e.g., the agent may learn that traversing a hallway during a busy period takes longer than in the middle of the night. In general, the actual time it takes an action to execute, t, will differ from the expected execution time $e(a, \Phi)$. We define the estimation error in the execution time of an action a by $\frac{|e(a,\Phi)-t|}{e(a,\Phi)}$.

Since external actions will, in general, require more time than a single deliberation cycle to execute, this means that multiple actions will typically be executing concurrently. We assume that each external action executes in its own thread (internal actions, e.g., belief update, execute in the interpreter thread). However since actions in each intention are executed sequentially, the total number of threads is bounded by the number of intentions.

Deferring the selection of plans until the corresponding goal must be achieved allows BDI agents to respond flexibly to the current state of the environment, by adapting the means used to achieve a goal to the current circumstances. However, the very large space of possible executions of a

BDI program [69] again makes it difficult for an agent developer to determine whether a goal can be achieved by its deadline, and if it can, which of the applicable plans for the goal should be selected to achieve it by the deadline.

6.2 Problem Specification and *SAM* Configuration

In the problem of intention progression with deadlines, two criteria are used to measure the performance of the agent. The first is the number of top-level goals achieved by their deadline, and the second is the total tardiness of achieving these goals (i.e, the difference between the time goals are achieved and their preferred achievement time). The problem of intention progression with deadlines is therefore defined as: given a set of goal-plan trees with their corresponding current step pointers, representing the current intentions of the agent $\{t_1, \ldots, t_n\}$, a set of condition variables *Env* representing the current state of the environment and the agent, return a next step of a goal-plan tree t_i to be executed at each deliberation cycle such that the number of goals achieved before their deadline is maximised, and the tardiness of achieving these goals is minimised, subject to achieving as many goals as possible. Note that this definition of what is the 'best' schedule means that any amount of tardiness in the achievement of a goal (up to the goal's hard deadline) is acceptable in order to achieve one more goal before its deadline.

In order to schedule the progression of an agent's intentions with deadlines, the function f_a in *SAM* is configured to return the UCT^{SP} values for a given node (see Equation 4.3), the comparator function f_b compares two nodes on their best simulation results (see Algorithm 11 in Chapter 5), and the function f_μ is defined as follows:

Algorithm 12 Utility function f_μ

```
1: function f_μ(n)
2:     if n.getCompletedGoals() == 0 then
3:         return 0
4:     else
5:         return n.getCompletedGoals() + 1/(TARDINESS(n) + 1)
```

where n.`getCompletedGoals()` returns the number of goals achieved in node n, and TARDINESS(n) returns the total tardiness of achieving these goals.[1] The utility of a node is the sum of the number of goals achieved in this node and $1/(\text{TARDINESS}(n)+1)$, where $1/(\text{TARDINESS}(n)+1)$ is in the range $(0,1]$ (i.e., we want to minimise the total tardiness of achieving goals). The utility function f_μ therefore ensures that the number of goals achieved is always more important than the total tardiness of achieving the goals (i.e., the first priority is to maximise the number of goals achieved before their deadline, and if the number of goals achieved are the same, minimise the total tardiness in achieving these goals). Note that the utility function return 0 when the number of goals achieved equals to 0.

6.3 Evaluation

In this section, we evaluate the ability of our approach to schedule intentions with deadlines. We compare the performance of *SAM* with AgentSpeak(RT).

AgentSpeak(RT) AgentSpeak(RT) (AS(RT)) [63, 62] is a real-time agent programming language based on AgentSpeak(L), in which top-level goals may have deadlines and priorities, and in which durative actions may execute in

[1] To calculate the total tardiness, the operations `getCompletionTime()`, `getDeadline()` and `getPreferredTime()` are used (see Chapter 4).

parallel.[2] In AgentSpeak(RT), top-level goals may optionally specify either a hard or soft deadline,[3] and/or a priority. A *hard deadline* is similar to our notion of deadline, and indicates that there is no value in achieving a goal after the deadline has passed; an intention that cannot be successfully executed by a hard deadline is dropped. A *soft deadline* corresponds to our notion of preferred achievement time; an intention that exceeds a soft deadline continues to execute, provided that this does not interfere with higher priority goals. A *priority* specifies the relative importance of achieving the goal. Real-time tasks can be freely mixed with tasks for which no deadline and/or priority is specified. Tasks without deadlines are processed after any task with a specified deadline.

AS(RT) and *SAM* take different approaches to handling potential conflicts between intentions. In AgentSpeak(RT) plans may be declared to be *atomic* (e.g., if actions in one plan may destroy the preconditions of actions in another plan). A non-atomic plan π cannot have atomic sub-plans, i.e., none of the plans in the goal-plan (sub)tree below π may be atomic. As a result, intentions in AS(RT) are either atomic or non-atomic; if the root plan of an intention is atomic, the intention is atomic, otherwise the intention is non-atomic. (In [3], this restriction is shown to be necessary for the scheduling approach adopted by AS(RT) to be tractable.) Intentions which do not contain an atomic plan (non-atomic intentions) are executed concurrently; the execution of intentions containing one or more atomic plans (atomic intentions) are serialised. In contrast, *SAM* avoids conflicts between intentions by appropriate interleaving

[2]We focus on the version of AgentSpeak(RT) described in [62], as this supports soft deadlines and parallel execution of intentions. A similar approach to scheduling intentions is used in the language N-2APL [3].

[3]Note, top-level goals cannot specify both a hard and soft deadline in AgentSpeak(RT).

of potentially conflicting plan steps. It is therefore not necessary to mark plans whose actions may interfere with actions in other plans as atomic.

AS(RT) and *SAM* also differ in how they estimate the time required to execute an intention. AS(RT) assumes that each action and plan has an associated *execution time profile* which specifies the probability that the action or plan will terminate successfully as a function of execution time. Execution time profiles can be derived from an analysis of the agent's actions, plans and environment, or using automated techniques, e.g., stochastic simulation.

In AS(RT), a developer specifies a required level of confidence for the successful execution of intentions in terms of a probability C, and an AS(RT) agent schedules its intentions so as to ensure that the probability that intentions are completed by their deadlines is at least C. By varying the confidence level, the developer can control the degree of 'optimism' the agent adopts when determining the time required to complete a task in a given environment. In task environments requiring a higher level of confidence, the agent will typically allow more time to complete tasks (and so schedule fewer tasks). As tasks are scheduled in priority order, increasing the level of confidence also has the effect of causing the agent to focus more on high priority tasks at the expense of lower priority tasks which might be achievable given a more optimistic view of execution time. In contrast, *SAM* uses the expected execution time of individual actions to estimate whether a particular schedule will achieve the agent's goals by their deadlines.

An AgentSpeak(RT) agent commits to a set of intentions that is *priority-maximal*: no more intentions can be added to the schedule if the scheduled intentions are to remain feasible (executable by their hard deadline) at the

specified confidence level. If not all atomic intentions can be achieved by their deadlines, the agent prefers intentions with greater priority. The AS(RT) scheduler maintains two schedules: an atomic schedule containing atomic intentions, and a non-atomic schedule for non-atomic intentions. The set of candidate intentions is processed in descending order of priority. Non-atomic intentions with a hard deadline are added to the non-atomic schedule if the probability that they will complete execution by their deadline is at least C. All non-atomic intentions with a soft deadline are scheduled in the non-atomic schedule (i.e., executed in parallel). Atomic intentions with a (hard or soft) deadline are scheduled if they can be inserted into the atomic schedule in deadline order while meeting their own and all currently scheduled deadlines with confidence C. Scheduling in AS(RT) is pre-emptive in that the adoption of a new high-priority atomic intention τ may prevent previously scheduled atomic intentions with priority lower than τ (including currently executing intentions) being added to the new schedule. Intentions which exceed their hard deadline are dropped. If an atomic intention has a soft deadline that has been exceeded, its deadline is reset to ∞, i.e., it is scheduled after all other atomic intentions. Non-atomic intentions that exceed their soft deadlines continue to be executed in parallel with other non-atomic intentions.

In [63], a number of properties of AS(RT) are shown, including bounds on the cycle time and reactivity delay of an AS(RT) agent; that the schedule returned by the AS(RT) scheduling algorithm is priority-maximal; and the probability that an intention will execute successfully (before its hard deadline) in a static environment is at least C.

To facilitate comparison with *SAM*, we extended AgentSpeak(RT) so that top-level goals may have both a hard and soft deadline, and all goals are assumed to have the same priority. We compare *SAM* and AgentSpeak(RT) in both synthetic domains and in a more realistic domain based on the classical Blocks World scenario. The Blocks World scenario is very simple, but representative of a large class of real-world problems in which an agent must collect and use resources in order to achieve goals. We evaluate the performance of each approach on two criteria: the number of goals achieved, and the total tardiness of all achieved goals. For all the experiments reported below, both *SAM* and AgentSpeak(RT) do not make use of failure handling as in Chapter 5. The confidence level used by AS(RT) was 0.7. *SAM* is configured to perform 100 iterations ($\alpha = 100$) and 10 random simulations at each iteration (β = 10).

6.3.1 Synthetic Domain

In the interests of generality, our first scenario is based on sets of randomly-generated, synthetic goal-plan trees representing the current intentions of an agent in a simple static environment. The sets of synthetic goal-plan trees are generated using GenGPT, as discussed in Chapter 5. By varying the number of atomic goal-plan trees, the average deadline (and preferred achievement time), and the execution time estimation error, we can evaluate the performance of each approach under different conditions. In this section, the performance of *SAM* and AS(RT) are evaluated only in static environment, as the changes from the environment (which is unpredictable) will hugely affect the performance of both approaches (with the real-time constructs).

Environment

As in Chapter 5, the environment is defined by a set of propositions that may appear as the pre- or postconditions of actions and the context conditions of the plans in a goal-plan tree. If the precondition of an action selected for execution does not hold in the environment, execution of the intention fails, and the corresponding top-level goal is dropped. If the precondition holds, the action is executed, and the environment updated with the postconditions of the action. Each action has an estimated execution time $e(a,\Phi)$. The actual time required to execute an action is determined by the environment, and is controlled by a parameter E, which specifies the estimation error in $e(a,\Phi)$ values. The time required to execute an action a is given by $e(a,\Phi)+\varepsilon$, where epsilon is a value randomly chosen in the range $[-e*E, e*E]$. By increasing E, we can increase the difficulty of scheduling, by making the estimated execution times of intentions less reliable.

Goal-Plan Trees

As explained above, AS(RT) and *SAM* take a different approach to avoiding conflicts between intentions. AS(RT) resolves potential conflicts by serialising execution of intentions where the plan selected for the top-level goal is an atomic plan. In contrast, *SAM* analyses the interactions between the pre- and postconditions of actions in plans, and attempts to schedule the actions so as to avoid conflicts. To allow evaluation of both approaches on the same set of goal-plan trees, each set of goal-plan trees may contain both non-atomic and atomic trees. In a non-atomic goal-plan tree, the pre- and postconditions of actions

are drawn from a set of variables that are unique to the tree.[4] This means that interleaved execution of non-atomic trees cannot give rise to conflicts (since each goal-plan tree uses its own set of conditions). In an atomic tree, the pre- and postconditions of actions in the goal-plan tree are drawn from a set of 20 environment variables shared between all atomic goal-plan trees in a given experiment. Parallel execution of the atomic goal-plan trees may therefore give rise to conflicts. By increasing the number of atomic goal-plan trees, we can increase the difficultly of the intention selection problem. A larger number of atomic goal-plan trees means more intentions must be serialised by AS(RT), and increases the number of potential conflicts between intentions for *SAM*.

For each experiment reported below, we generated 50 sets of 10 goal-plan trees. The depth of each goal-plan tree is set to 5. Each (sub)goal has two plans to achieve it. Each plan consists a sequence of three actions, followed by a subgoal. The maximum p-effect size is ∞. The number of environment variable is set to 20.

Execution times

Both *SAM* and AS(RT) require information about the expected execution time of plans. With each basic action a appearing in a plan we associate a randomly-generated expected execution time, $e(a,\Phi)$, in the range $100 \leq e(a,\Phi) \leq 500$. For simplicity, we assume that e is expressed as multiple of the time required for a single deliberation cycle of the agent.[5] As discussed, AS(RT) assumes that execution time profiles are available for both the actions and plans in a

[4]The goal-plan trees are generated using GenGPT. For non-atomic goal-plan trees, we replace the conditions in the tree by using a set of variables that are unique to other trees.

[5]The expected execution time for each action is added after the goal-plan trees are generated.

goal-plan tree. For actions, we used the expected execution time for the action. In the case of plans, we used stochastic simulation to generate an execution time profile. For each plan in a goal-plan tree, we performed 100 simulated executions. Each simulation consisted of a randomly selected path from the first action in the plan to a leaf node. For each simulation, we recorded the total execution time of all actions on the path. We then counted the number of simulations where the time required for the plan to complete execution, T, is less than or equal to $min + k * (max - min)/10$, $(1 \leq k \leq 10)$, where *min* is the minimum observed execution time for the plan and *max* is the maximum observed execution time. The number of simulations for each time interval are then used to produce an execution time profile for the plan.

Results

In the first three experiments, we evaluate the performance of *SAM* when each top-level goal has both a deadline and a preferred achievement time, and report both the number of goals achieved and the total tardiness of achieving the goals. For Experiments 4-6, the performance of *SAM* is evaluate when each top-level goal has only a preferred achievement time (i.e., the deadline for each top-level goal was set to ∞), and only the total tardiness of achieving the goals were reported (as both approaches are able to achieve all 10 goals in all cases).

Experiment 1 In the first experiment, the deadline for each goal-plan tree was randomly generated in the range 11500 – 15500, giving an average deadline of 13500. The preferred achievement time for each goal-plan tree was 5000 time units less than its deadline. To achieve a top-level goal requires on average 4500 time units (15 actions need to be executed and the average

execution time for each action is 300 time units). A deadline of 13500 means that, on average, the agent must execute approximately 3.3 goal-plan trees in parallel to achieve all 10 goals by their deadlines. The number of atomic goal-plan trees was varied from 0 to 10 and the execution time estimation error was 0. A larger number of atomic goal-plan trees means more intentions must be serialised by AgentSpeak(RT), and increases the number of potential conflicts between intentions for *SAM*.

Table 6.1 Goals & tardiness with increasing numbers of atomic goal-plan trees

	AgentSpeak(RT)		*SAM*	
#Atomic	#Goals	Tardiness	#Goals	Tardiness
$0\backslash 1$ [6]	10	222	10	222
2	9.72	2811	10	316
4	8.08	3326	10	862
6	6.42	3813	9.92	2153
8	4.64	3057	9.86	2807
10	3.02	3461	9.80	3712

Table 6.1 shows the average number of goals achieved and total tardiness of achieving the goals for both approaches as the number of atomic goal-plan trees increases. As can be seen, with only a small number of atomic goal-plan trees AgentSpeak(RT) is able to achieve nearly all goals. As the number of atomic goal-plan tree increases, the performance of AgentSpeak(RT) declines. In the most difficult case where all goal-plan trees are atomic, AgentSpeak(RT) is able to achieve only 3.02 goals on average. This is because, to avoid conflicts, the execution of all intentions are serialised by $AgentSpeak(RT)$. The performance of *SAM* also reduces as the number of atomic goal-plan trees

increases. However, even in the most difficult case, *SAM* is able to achieve 9.8 goals on average, and in simpler cases is also able to reduce tardiness.

Experiment 2 In the second experiment, we varied the average deadline of top-level goals from 9000 to 18000. When goal-plan trees have an average deadline of 9000 time units, the agent must execute approximately 5 goal-plan trees in parallel in order to achieve all the top-level goals by their deadline. With an average deadline of 18000 time units, approximately 2.5 goal-plan trees must be executed in parallel to achieve all the top-level goals by their deadline. As before, the preferred achievement time was 5000 time units less than the deadline. The number of atomic goal-plan trees was 4, and the execution time estimation error was 0.

Table 6.2 Goals & tardiness with decreasing average deadline

		AgentSpeak(RT)		*SAM*	
d	*p*	#Goals	Tardiness	#Goals	Tardiness
18000	13000	9.06	3094	10	0
13500	8500	8.08	3326	10	862
9000	4000	7.12	4017	10	1738

The results are shown in Table 6.2. As expected, the performance of both approaches decline as the average deadline reduces. AgentSpeak(RT) is able to achieve fewer goals and tardiness increases. In contrast, *SAM* is able to achieve all 10 goals in all cases, and tardiness is less than for AgentSpeak(RT).

Experiment 3 In the third experiment, the effect of including an execution time estimation error of 30% is investigated, i.e., the actual time required to execute an action may be up to 30% more or less than the estimate. As in the first experiment, the average deadline for each goal-plan tree was 13500, and the preferred achievement time was 5000 time units less than the corresponding

deadline. The number of atomic goal-plan trees was 4 (row 3 of Tables 6.1 and 6.2).

Table 6.3 Goals & tardiness with and without estimation error

	AgentSpeak(RT)		*SAM*	
E	#Goals	Tardiness	#Goals	Tardiness
0%	8.08	3326	10	862
30%	7.9	2850	9.98	937

The results are shown in Table 6.3. As can be seen, when there are estimation errors the performance of both *SAM* and AgentSpeak(RT) declines, i.e., the number of goals achieved decreases; however the performance of *SAM* is still significantly better than AgentSpeak(RT).

Experiment 4 In the fourth experiment, the same experimental values as in the first experiment were used except that the deadline for each top-level goal was ∞, and the average preferred achievement time was 13500 time units.

Table 6.4 Tardiness with increasing numbers of atomic goal-plan trees

	AgentSpeak(RT)	*SAM*
#Atomic	Tardiness	Tardiness
0\1	0	0
2	26	0
4	7611	52
6	30667	127
8	68504	203
10	118517	439

The results are shown in Table 6.4. As both approaches are able to achieve all 10 goals, we report only the tardiness in achieving the goals. As can be seen, as the number of atomic goal-plan trees increases, tardiness increases for both *SAM* and AgentSpeak(RT). However *SAM* has significantly lower tardiness than AgentSpeak(RT).

Experiment 5 In the fifth experiment, the average preferred achievement time of top-level goals were varied. The same setting as in the second experiment was used except that the deadlines for each top-level goal was ∞, and the average preferred achievement time was varied from 9000 to 18000.

Table 6.5 Tardiness with decreasing average preferred achievement time

	AgentSpeak(RT)	*SAM*
p	Tardiness	Tardiness
18000	1755	0
13500	7611	0
9000	15777	62

The results are shown in Table 6.5. As can be seen, the tardiness of AgentSpeak(RT) increases as the preferred achievement time decreases, however the tardiness of *SAM* is almost unaffected (tardiness of 62 is less than the time required to perform a single action).

Experiment 6 As in the third experiment, an execution time estimation error of 30% was included in Experiment 6. The deadline for each top-level goal was ∞ and the average preferred achievement time was 13500. Other values were as in Experiment 3.

Table 6.6 Tardiness with and without estimation error

	AgentSpeak(RT)	*SAM*
E	Tardiness	Tardiness
0%	7611	0
30%	8194	13

The results are shown in Table 6.6. As can be seen, increasing the estimation error increases tardiness significantly for AgentSpeak(RT) (as all goals are achieved), while *SAM* is almost unaffected.

6.3.2 Blocks World Domain

In this section, *SAM* was evaluated in a scenario based on the classic Blocks World. In the scenario, an agent must pick blocks and stack them to build towers. Each top-level goal involves picking up two blocks from different locations and building a tower at a specified location. Each top-level goal has a preferred achievement time and deadline, specifying the times by which the task should and must be completed. Picking up and stacking blocks, and moving from one location to another requires time. To test the ability of AgentSpeak(RT) and *SAM* to respond flexibly to the asynchronous arrival of goals, a subset of the goals are given initially, and the remainder are posted at intervals. As in the synthetic domain, AS(RT) uses atomic constructs to avoid conflicts. The execution time of actions and plans are computed using stochastic simulation as in Section 6.3.1.

This scenario is very simple, but representative of a large class of real-world problems where tasks require effort (time) to obtain resources which are then used to complete the task. (Consider an airport shuttle, which receives requests to collect groups of passengers and deliver them to different airport terminals in time for their flight.) In the scenario, the agent must travel to collect resources, but in general the acquisition of a resource may be any (atomic) activity that requires time, e.g., the agent could manufacture the resource itself. Similarly, the second phase of the task may be any (atomic) activity that requires the acquired resources. Different tasks require different resources (blocks) which may be located near resources required for other tasks, or near the location of a tower. There are therefore opportunities for the agent to interleave tasks so as to reduce the overall execution time for all tasks. At the same time, tasks

cannot be arbitrarily interleaved; for example, after travelling to a location to pick up a block or build a tower, the agent should complete the sub-task before moving to another location.

The goal-plan tree used in the experiments is shown in Figure 6.1. For legibility, the deadlines and preferred achievement time for each goal, and the estimated execution time for each action are not shown. The top-level goal, G_0 specifies that a tower of blocks b_1 and b_2 should be built at position p_3. There is a single plan to achieve G_0, which in turn has two subgoals: to collect the blocks G_1, and to stack the blocks in a tower, G_6. Goal G_1 can be achieved by either of two plans, P_1 (pickup b_1 and then b_2) or P_4 (pickup b_2 and then b_1) allowing blocks to be collected in any order. Finally, each leaf plan consists of a sequence of two actions. For example, plan P_2 to pick up block b_1, involves moving to the location of b_1 and picking it up. To ensure that the interleaving of the agent's intentions happens at the appropriate points, the P_2, P_3, P_5, P_6 and P_7 plans in the goal-plan tree shown in Figure 6.1 are marked as atomic for AgentSpeak(RT). This in turn requires that P_1, P_4 and P_0 are atomic (since in AgentSpeak(RT) an intention must be either atomic or non-atomic).

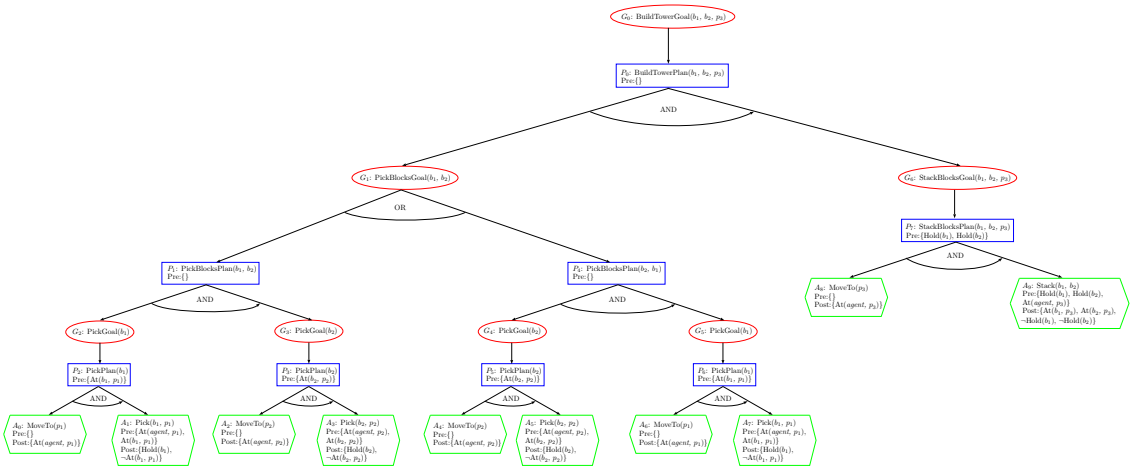


Fig. 6.1 Goal-plan tree for the Blocks World domain

In the experiments reported below, we used an environment with 30 cells. Initially, 20 randomly selected cells contain a block (numbered from 1 to 20), and the remaining 10 cells (the locations of the towers) are empty. The actions of picking up a block and stacking a block require 100 time units. The execution time required by a move action is determined by the distance the agent has to move (moving one cell of distance requires 100 time units). For simplicity, we assume the blocks and tower positions are located on a straight line, and the number of blocks the agent can carry is unlimited.

The performance of *SAM* and AS(RT) was evaluated in two experiments. Each experiment consisted of 50 sets of 10 goals. Each goal involves collecting two specified blocks and stacking them in a specified empty cell (the block ids and tower positions are different for each goal). Each goal has a deadline and preferred achievement time as specified below. Three goals are given initially, and the remaining top-level goals are posted every 1650 time units. As the average time for achieving a top-level goals is 3300 time units, posting a new goal every 1650 time units means that, typically, the agent must achieve more than one goal in parallel. *SAM* is configured to use the same settings as for the experiments in the synthetic domain. For AgentSpeak(RT), we generated an expected execution time profile for each goal using stochastic simulation of the ground instance of the goal-plan tree shown in Figure 6.1 for that goal, by varying the initial starting position of the agent. Note that this setting is favourable to AgentSpeak(RT), as it assumes that the locations of the blocks tower are known when generating the execution time profiles, and the only source of variability in expected execution time comes from the position of the agent. Computing the expected execution time profile for the generic goal-plan

tree, i.e., for randomly sampled block and tower positions, would give a much more 'conservative' execution time profile (since the positions of the blocks and tower vary, the time required to build a tower with C confidence is greater). As a result, the AgentSpeak(RT) scheduler will schedule fewer intentions and so achieve fewer goals in the generic case.

Results

Experiment 7 In the first experiment, both the number of goals achieved by the agent and the total tardiness in achieving these goals were reported. The deadline for each top-level goal was varied from 7500 to 30000, and the preferred achievement time was 2500 time units less than the deadline.

Table 6.7 Goals and tardiness with decreasing deadline

		AgentSpeak(RT)		*SAM*	
d	p	#Goals	Tardiness	#Goals	Tardiness
30000	27500	10	0	10	0
15000	12500	6.96	2386	10	0
10000	7500	5.38	3195	10	2340
7500	5000	5.04	3220	9.34	3874

The results are shown in Table 6.7. As can be seen, as the deadline decreases, the performance of AgentSpeak(RT) declines rapidly. In contrast, *SAM* was able to achieve at least 9.34 goals on average, though with tighter deadlines, tardiness increases significantly.

Experiment 8 In the second experiment, the deadline for each top-level goal was ∞ and the preferred achievement time was varied from 7500 to 30000 time units. As both approaches can achieve all top-level goals, only the total tardiness in achieving the goals was reported.

Table 6.8 tardiness with decreasing preferred achievement time

	AgentSpeak(RT)	*SAM*
p	Tardiness	Tardiness
30000	0	0
15000	22144	0
10000	52382	0
7500	72136	1918

The results are shown in Table 6.8. As expected, the performance of both approaches decline as the preferred achievement time decreases. However the tardiness of *SAM* is much lower than that for AgentSpeak(RT).

6.4 Summary

In this chapter, we show how *SAM* can be used to schedule with deadlines and preferred achievement times. In this setting, the agent is aiming to maximise the number of goals achieved before the deadline, and to minimise the total tardiness, subject to achieving as may goals as possible.

We evaluated the performance of *SAM* in both synthetic domains and a more realistic scenario based on the Blocks World domain. Our results indicate that our approach out-performs AgentSpeak(RT) in terms of the number of goals achieved and the total tardiness of all achieved goals. In addition, *SAM* requires only the expected execution time of actions rather than an execution time profile for each action and plan in an agent program, and can reduce the development's effort as the developer no longer has to anticipate and control possible interactions between intentions using atomic constructs. In the next chapter, we will show how *SAM* can be used to exploit synergies between intentions.

Chapter 7

Exploiting Synergies

In chapter 7, we use the *SAM* scheduler to exploit the synergies between an agent's intentions. In the first part of this chapter, we focus on scheduling fallible actions (actions which have more than one possible postconditions), where synergies between intentions can be used to recover from execution failure. In the second part of this chapter, we focus on merging actions which have same postconditions to save execution steps. As in previous chapters, we evaluate the performance of *SAM* in both synthetic domains and more realistic domains based on the Miconic-10 elevator domain and the classic Blocks World domain. The experimental results indicate that the potential benefits of using *SAM* to exploit synergies between intentions is quite significant.

7.1 Scheduling With Fallible Actions

Deferring the selection of plans until the corresponding goal must be achieved allows BDI agents to respond flexibly to the current state of the environment, by adapting the means used to achieve a goal to the current circumstances.

However, as with any software system, BDI agent programs may fail. Several different types of execution failure can be distinguished.[1]

Coverage failures occur when there is no applicable plan for the current (sub)goal in the current environment. For example, suppose you have the goal of going to the beach, and you have two plans to achieve this goal: going on foot or by taxi. The precondition of the first plan (i.e., going on foot) is that the weather is sunny and you are not tired. The precondition of the second plan (i.e., going by taxi) is you have enough money to pay the taxi fare. If the weather is rainy and you don't have money at the moment, then there is no applicable plan to achieve the goal, and thus there is a coverage failure.

Precondition failures occur when an action cannot be executed because a precondition of the action does not hold in the current environment (e.g., in order to pick up an object, a robot agent must be within gripping distance of the object).

Postcondition failures occur when the intended postcondition of an action does not hold after executing the action (e.g., if the gripper is faulty, a *pickup* action may not result in the agent holding the object).

In many cases, a postcondition failure leads to a subsequent coverage or precondition failure if the intention were to continue execution. We will use the general term *execution failure* to mean any of these types of failure.

In this section, we assume intentions may contain fallible actions. Fallible actions can fail, as they have a set of possible postconditions rather than a

[1] We ignore failures arising from programmer error, and assume that programs are correctly written.

single postcondition. Fail to achieve the intended postcondition of a fallible action causes a postcondition failure, and as discussed, a postcondition failure leads to a subsequent coverage or precondition failure if the intention were to continue execution.

7.1.1 Recovering From Execution Failure

The 'standard' approach to execution failures in BDI agent programming is to abort the plan in which the failure occurred (perhaps after performing some 'cleanup' actions), and try another applicable plan to achieve the current (sub)goal. For example, in agent programming languages such as JACK [68] and Jadex [43], when an execution failure occurs another applicable plan is tried to achieve the current (sub)goal. If no alternative means is available, then the goal is deemed failed, and the failure is propagated to higher-level motivating goals. This approach to failure handling gives rise to a very large behaviour space for BDI agent programs [69], and can result in robust execution. However, this robustness can come at a considerable cost in terms of backtracking (i.e., partial execution of a plan to the point of failure, followed by selection of an alternative plan for the current subgoal).

However in some situations, it may be possible to opportunistically recover from an execution failure by appropriate scheduling the agent's remaining progressable intentions to execute an already intended action which reestablishes a missing context or precondition. Actions which are executed in the same environment often have similar or overlapping preconditions; many physical actions rely on the location of the agent, for example. Alternatively, it may be possible to recover by selecting a plan for a subgoal in a progressable intention,

which, in addition to achieving its triggering goal, reestablishes the missing condition. Suppose an agent tries to open the door of the mailroom and finds out the door is broken. To recover from execution failure, the agent may try an alternative plan to ask someone in the school administration office to fix the door. However, moving from the current location to the school administration office and then moving back might be time-consuming. Instead, the agent could try to execute other intentions first, and then comes back and hopes the door of the mailroom is already fixed. Although such opportunistic recovery may delay the achievement of the top-level goal of the failed intention (until the relevant condition can be reestablished by another intention), exploiting synergies between intentions can reduce the total number of actions an agent has to perform (e.g., moving from the current location to the school administration office in the previous example), by reducing backtracking .

7.1.2 Problem Specification and *SAM* Configuration

For simplicity, we assume the deadline and preferred achievement time for all goals are infinity, and the execution of all actions are instantaneous as in Chapter 5. Two criteria are used to measure the performance of the agent. The first is the number of goals achieved, and the second is the number of 'backtracks' (selection of an alternative plan for a subgoal) required to achieve these goals. The number of 'backtracks' can be seen as the amount of effort 'wasted' by the agent in achieving its goals (i.e., plans that are partially executed before being dropped when they become unexecutable in favour of an alternative plan for a subgoal). The problem of intention progression with fallible actions is therefore defined as: given a set of goal-plan trees with their

corresponding current step pointers, representing the current intentions of the agent $\{t_1, \ldots, t_n\}$, a set of condition variables *Env* representing the current state of the environment and the agent, return a next step of a goal-plan tree t_i to be executed at each deliberation cycle such that the number of goals achieved is maximised, and the total number of backtracks requires to achieve these goals is minimised, subject to achieving as many goals as possible.

The problem of scheduling the progression of an agent's intentions which contain fallible actions is considered as a two-player game, where the different possible postconditions of a fallible action represent an opponent's moves. In the expansion phase, *SAM* creates multiple child nodes for a fallible action representing all its possible postconditions. In the simulation phase, after executing a fallible action, one of the fallible action's postcondition is randomly selected and applied to the environment.

SAM is used to exploit synergies between intentions to recover execution failures before backtracking is initiated. That is, when an execution failure occurs, e.g., the precondition of an action does not hold in the current environment, *SAM* does not backtrack to try an alternative plan for the current subgoal immediately as in JACK or Jadex. Instead, *SAM* attempts to find an execution order of the steps in progressable intentions which re-establishes the precondition of the failed action. Only when all the agent's remaining intentions become non-progressable, does *SAM* drop the currently intended plan for the current subgoal in each intention, and consider an alternative applicable plan for each subgoal. If all applicable plans for a subgoal have been tried and failed, then the current subgoal is dropped and failure is propagated to higher-level goals as in JACK and Jadex.

The function f_a in *SAM* is configured to return the UCB1 values for a given node (see Equation 4.2), the function f_μ calculates the number of goals achieved (see Algorithm 14), and the comparator f_b is defined as follows:

Algorithm 13 Comparator f_b

```
1: function f_b(n_i, n_j)
2:     if n_i.getVisited() > n_j.getVisited() then
3:         return True
4:     else
5:         return False
```

n_i.`getVisite()` represents the number of times node n_i has been visited. The comparator f_b is used to compare two nodes on the number of times they have been visited. That is, a node n_i is better than a node n_j, if n_i has been visited more times than n_j. When the algorithm terminates, *SAM* selects the child node of the root node which has been visited most times (i.e., the Robust Child defined in [51]).

Algorithm 14 Utility function f_μ

```
1: function f_μ(n)
2:     return n.getCompletedGoals()
```

7.1.3 Evaluation

In this section, the performance of *SAM* was evaluated and compared with the Round Robin scheduling with failure handling (*RR*+)[2] in scenarios of increasing difficulty. In *RR*+, intentions are executed in RR fashion. When an execution failure occurs, the current intended plan is dropped and an alternative applicable plan is tried. If there are no applicable plans, the current subgoal is dropped and failure is propagated to higher-level goals. *RR*+ was chosen

[2] *RR*+ does include failure handling, as handling action failures is the focus of this section.

as representative of the state of the art in practical implementations of agent programming languages such as JACK and Jadex.

The performance of each approach was evaluated on two criteria: the number of goals achieved, and the number of 'backtracks'. For all the experiments reported below, *SAM* is configured to perform 100 iterations ($\alpha = 100$) and 10 random simulations at each iteration ($\beta = 10$).

Synthetic domain

As in Chapter 5 and 6, our first sets of experiments are based on sets of randomly-generated, synthetic goal-plan trees representing the current intentions of an agent in a simple static environment. By controlling the characteristics of the trees, the probability of action failure, and the number of fallible actions, we can evaluate the performance of each approach under different conditions.

Environment As in Chapter 5 and 6, the environment is again defined by a set of propositions that may appear as pre- or postconditions of plans and actions in a goal-plan tree. If the precondition of an action selected for execution does not hold in the environment, execution of the intention fails, and the corresponding top-level goal is dropped. If the precondition holds, the action is executed, and the environment updated with the postconditions of the action.

Goal-Plan Trees The sets of synthetic goal-plan trees are generated using GenGPT (see Chapter 5). For each experiment reported below, we generated 50 sets of 10 goal-plan trees. The depth of each goal-plan tree is set to 5. Each (sub)goal has two plans to achieve it. Each plan consists a sequence of three actions, followed by a subgoal. The maximum p-effect size is ∞. The number of environment variable is varied from 20 to 50 in each experiment. After each

set of goal-plan trees is generated, a subset of the actions in the set of goal-plan trees is selected and modified to be fallible actions. Fallible actions have a set of possible postconditions rather than a single postcondition.

We distinguish two kinds of fallible action failures: *non-disruptive* and *disruptive*. The failure of a fallible action is *non-disruptive* if it has two possible postconditions and the unintended postcondition is the negation of the intended postcondition. For example, a *pickup* action fails non-disruptively if it has *holding_block* and ¬*holding_block* as its possible postconditions. An action which fails non-disruptively does not change the state of the environment. The agent was not holding the block before attempting the action and is not holding the block after executing the action.[3] An action failure is *disruptive* if at least one of the unintended postconditions is not the negation of the intended postcondition. For example, a *pickup* action fails disruptively if it has two possible postconditions: *holding_block* and *gripper_broken*. An action that fails disruptively changes the state of the environment in some unintended way. In addition to failing to establish the precondition of a subsequent action in the same intention, a disruptive failure may destroy the precondition of the action itself (e.g., ¬*gripper_broken*), with the result that the action can't simply be retried.

When the set of goal-plan trees is generated, each action in the goal-plan trees has *ND* probability to have another possible postcondition. An infallible action was transformed to a fallible action by adding an extra postcondition to it. The original postcondition of the action is then considered as the intended postcondition, and the new postcondition is considered as the unintended post-

[3] We assume that ¬*holding_block* is a precondition of the action, i.e., the agent would not attempt to pickup the block if it is already holding it.

condition. If the fallible action fails non-disruptively, then the unintended postcondition is the negation of its intended postcondition. For example, an action has p and $\neg p$ as its possible postconditions may fail non-disruptively, where p is its intended postcondition and $\neg p$ is its unintended postcondition. If the fallible action fails disruptively, then the unintended postcondition is randomly selected from the environment. Note that the unintended postcondition cannot be the same as the intended postcondition or its negation. For example, an action has p and q as its possible postconditions may fail disruptively, where p is its intended postcondition, q is its unintended postcondition, and $p \neq q$ or $\neg q$. In the experiments reported below, the performance of *SAM* was evaluated in the scenarios where fallible actions fail non-disruptively, as well as in the scenarios where fallible actions fail disruptively.

Action Fails Non-disruptively In the first scenario, all fallible actions fail non-disruptively. The results of the first set of experiments are shown in Tables 7.1 and 7.2. In the tables, $\#v$ is the number of environment variables and ND is the probability that an action is fallible when generating the goal-plan trees (0%, 5%, 10%, 20% and 30%). *#Goals* is the average number of top-level goals achieved by each approach, and *#Backs* is the average number of backtracks. For each percentage of fallible actions considered, the first row gives results for an action failure probability of 10% (i.e., ρ = 10%), and the second and third rows give results for action failure probabilities of 25% and 50%.

Table 7.1 shows the experimental results of *RR*+, and the experimental results of *SAM* is shown in Table 7.2. As can be seen, in cases where there is little interaction between the goal plan trees ($\#v = 50$), and the percentage of fallible actions in plans is small, the average number of goals achieved

Table 7.1 *RR+*; all fallible actions fail non-disruptively, 10 goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	9.32	13.82	9.34	15.82	9.32	21.50	8.26	30.26
5%	9.32	14.16	9.32	16.36	9.22	20.92	8.22	30.96
	9.30	14.48	9.26	17.88	9.00	23.06	8.20	30.26
	9.18	16.58	8.92	18.74	8.88	23.50	8.00	32.46
10%	9.26	14.88	9.14	17.34	9.12	22.92	8.20	29.32
	9.16	16.06	9.14	18.82	8.72	25.06	8.08	29.68
	8.88	19.90	8.70	23.22	8.46	25.70	7.90	33.94
20%	9.22	16	9.10	19.02	9.06	21.88	8.08	30.82
	8.98	19.26	8.94	22.26	8.48	25.60	7.72	32.32
	8.38	25.78	8.30	26.58	7.90	29.64	7.18	39.2
30%	9.04	18.18	9.08	19.18	8.94	24.90	7.94	33.46
	8.78	21.74	8.34	25.02	8.12	28.56	7.64	37.14
	7.62	30.96	7.24	34.60	7.42	37.18	6.56	44.78

Table 7.2 *SAM*; all fallible actions fail non-disruptively, 10 goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	10	0	10	0	10	0	10	0
5%	10	0	10	0	10	0.04	10	0.02
	10	0.02	10	0.02	10	0.04	10	0.04
	10	0	10	0.16	10	0.04	10	0.04
10%	10	0.02	10	0.04	10	0	10	0
	10	0.04	10	0.14	10	0.06	10	0
	10	0.24	10	0.12	10	0.50	10	0.22
20%	10	0.16	10	0.08	10	0.20	10	0.20
	10	0.38	10	0.36	10	0.24	10	0.22
	10	1.78	10	1.02	10	0.82	10	0.64
30%	10	0.52	10	0.48	10	0.48	10	0.28
	10	1.48	10	0.98	10	1.10	10	0.78
	10	6.16	10	3.58	10	2.26	10	1.76

by *RR+* and *SAM* is similar (9.32[4] for *RR+* vs 10 for *SAM*). However even in this setting, *RR+* requires significantly more backtracking to achieve the goals (13.82 vs 0). Although the number of interactions between intentions

[4]When a plan P fails to achieve a goal G, *RR+* will try alternative plans to achieve G but does not "clean up" the postconditions that are established by executing P. Therefore, in some cases, *RR+* cannot achieve all its top-level goals, if, for example, the postcondition of executing P affects the execution of other plans.

is small, actions in one intention may destroy the context conditions of plans or the preconditions of actions in other intentions. *SAM* in contrast is able to schedule the execution of intentions so as to avoid negative interactions (as discussed in Chapter 5), achieving all 10 goals, and in general no backtracking is required.

As the number of fallible actions, and the probability of each fallible action failing increases, the average number of goals achieved by *RR*+ decreases, and the number of backtracks required to achieve these goals increases. When ND = 30% and ρ = 50%, *RR*+ can achieve only 7.62 goals on average, and performs on average 30.96 backtracks to do so. The decrease in performance of *RR*+ relative to the case in which all actions are deterministic is due to the difficulty of recovering from action failures using only backtracking. *SAM* achieves all 10 goals and requires only 6.16 backtracks on average, demonstrating the utility of exploiting positive interactions.

As the number of environment variables decreases and the degree of interaction between the agent's intentions increases, the performance of *RR*+ declines, even when all actions in plans are deterministic. When the number of environment variables equals 20 (i.e., $\#v = 20$), *RR*+ achieves 8.26 goals on average and requires 30.26 backtracks, which is only a marginal improvement on the low interaction/high failure case. With high interaction and high failure of actions, the performance of *RR*+ declines significantly to 6.56 goals and 44.79 backtracks. In contrast, even in extremely challenging settings, *SAM* achieves all 10 goals, and requires only 1.76 backtracks on average. The number of backtracks is lower than in the low interaction ($\#v = 50$) case, as

SAM is able to exploit the larger number of interactions between intentions to recover from failures without backtracking in this case.

It might be argued that the degree of interactions between intentions in our experiments is higher than typically occurs in agent programs. For example, if the agent has only a single intention and it fails, *SAM* cannot use actions in other intentions to recover from the failure. To determine the extent to which the performance of *SAM* relies on the agent executing a 'large' number of intentions in parallel, the experiments above were repeated with five goal-plan trees in each trial (i.e., the agent progresses five intentions in parallel). Table 7.3 shows the experimental result of *RR*+, and the experimental result of *SAM* is shown in Table 7.4.

Table 7.3 *RR*+; all fallible actions fail non-disruptively, 5 top-level goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	4.96	3.18	4.92	3.82	4.88	5.36	4.72	7.46
5%	4.92	3.38	4.90	4.08	4.88	5.54	4.70	7.96
	4.92	3.92	4.90	4.58	4.86	6.18	4.70	8.02
	4.88	5.1	4.80	5.46	4.72	6.88	4.60	8.74
10%	4.92	3.90	4.92	4.14	4.84	5.72	4.68	8.52
	4.86	4.48	4.86	5.18	4.74	6.56	4.52	7.88
	4.72	6.22	4.66	7.28	4.62	8.30	4.42	9.72
20%	4.90	3.98	4.86	4.44	4.80	5.62	4.60	8.54
	4.78	5.48	4.72	6	4.72	7.34	4.42	9.22
	4.26	9.52	4.44	10.24	4.38	11.96	4.42	13.56
30%	4.86	4.72	4.84	5.12	4.80	6.72	4.54	9.18
	4.76	6.52	4.66	7.04	4.58	8.20	4.40	11.28
	4.20	14.12	4.06	12.82	3.96	14.30	3.76	15.86

As can be seen, in cases where are no fallible actions and the number of interactions between intentions is small ($\#v = 50$), the average number of goals achieved by *RR*+ and *SAM* is similar (4.96 for *RR*+ vs 5 for *SAM*). However, *RR*+ requires more backtracks than *SAM* (3.18 vs 0). As the number

Table 7.4 *SAM*; all fallible actions fail non-disruptively, 5 top-level goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	5	0	5	0	5	0	5	0
5%	5	0.02	5	0.02	5	0	5	0
	5	0.20	5	0.16	5	0.08	5	0.04
	5	0.34	5	0.30	5	0.18	5	0.14
10%	5	0.04	5	0.04	5	0.08	5	0.04
	5	0.32	5	0.20	5	0.32	5	0.06
	5	1.08	5	0.66	5	0.64	5	0.46
20%	5	0.16	5	0.12	5	0.12	5	0.08
	5	0.60	5	0.66	5	0.34	5	0.28
	5	1.80	5	1.70	5	1.62	5	0.96
30%	5	0.22	5	0.32	5	0.20	5	0.34
	5	1.14	5	0.92	5	0.66	5	0.60
	5	4.74	5	4.48	5	3.44	5	2.38

of fallible actions, and the probability of action failure increase, the average number of goals achieved by *RR*+ decrease, and the number of backtracks required by both *RR*+ and *SAM* increase. However, the performance of *RR*+ declines more markedly than *SAM*. As in the previous experiment (where the agent is asked to achieve 10 goals), the performance of *RR*+ declines as the number of environment variables decreases. In contrast, *SAM* has better performance when the number of environment variable is small. For example, when $\#v = 50$, ND = 30% and ρ = 50%, *SAM* is able to achieve all 5 goals and requires 4.48 backtracks on average. When $\#v = 20$, ND = 30% and ρ = 50%, *SAM* is able to achieve all 5 goals and only requires 2.38 backtracks.

As can be seen, the difference between *SAM* and *RR*+ becomes smaller as the number of parallel goals decreases. This is because *RR*+ is able to achieve more goals when the number of negative interaction (conflicts) is small. In addition, the number of possible synergies between intentions also decrease as the number of top-level goals decrease, and thus it is more difficult for *SAM* to

recover from execution failure by exploiting synergies. However, even in this setting, *SAM* out-performs *RR*+ on both the number of goals achieved and the number of backtracks required.

Actions Fail Disruptively In the second scenario, all fallible actions fail disruptively. The presentation of results is as in the first experiment.

Table 7.5 *RR*+; all fallible actions fail disruptively, 10 goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	9.32	13.82	9.34	15.82	9.32	21.50	8.26	30.26
5%	9.32	13.38	9.32	17.24	9.04	21.22	8.22	28.28
	9.16	14.36	9.06	16.58	9.02	22.16	8.20	28.66
	9.08	15.02	9.00	17.58	9.04	22.04	8.20	28.08
10%	9.30	13.68	9.26	16.54	8.92	21.54	8.12	27.78
	9.18	14.74	8.98	18.22	8.80	21.50	8.04	28.62
	9.04	15.76	8.94	19.34	8.76	24.08	7.78	30.36
20%	9.22	14.24	9.20	17.56	8.80	22.04	8.10	29.30
	9.10	16.20	9.02	19.88	8.70	24.10	7.88	30.04
	9.00	19.76	8.64	21.56	8.52	26.24	7.72	35.84
30%	9.20	14.44	9.16	17.42	8.68	20.72	7.88	30.96
	9.10	17.16	9.00	20.86	8.44	23.48	7.82	30.84
	8.68	20.62	8.52	34.60	8.14	31.44	7.52	36.92

As can be seen, in general, disruptive actions facilitate failure recovery, as there is a higher probability that the unintended outcome of a fallible action will reestablish a condition necessary to progress a failed intention. *RR*+ achieves more goals and requires fewer backtracks in the low interaction / high failure case, 8.68 goals and 20.62 backtracks compared to 7.62 goals and 30.96 backtracks when actions fail non-disruptively. This increase in performance also extends to the high interaction / high failure case, where *RR*+ achieves 7.52 goals and requires 36.92 backtracks compared to 6.56 goals and 44.78 backtracks in the first experiment. As in the first experiment, *SAM* achieves 10 goals in all cases, but the number of backtracks required is significantly

reduced. *SAM* requires 1.08 backtracks on average to achieve 10 goals when 30% of actions in plans are fallible, and actions fail with 50% probability in the low interaction ($\#v = 50$) case, and only 0.46 backtracks in the high interaction ($\#v = 20$) case.

Table 7.6 *SAM*; all fallible actions fail disruptively, 10 goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	10	0	10	0	10	0	10	0
5%	10	0	10	0	10	0.02	10	0
	10	0	10	0.02	10	0.02	10	0.02
	10	0.04	10	0	10	0	10	0
10%	10	0.02	10	0	10	0	10	0.02
	10	0.02	10	0	10	0.02	10	0.04
	10	0.06	10	0.02	10	0.04	10	0
20%	10	0.02	10	0	10	0.06	10	0.02
	10	0.14	10	0.10	10	0.24	10	0.02
	10	0.32	10	0.12	10	0.12	10	0.20
30%	10	0.18	10	0.10	10	0.22	10	0.18
	10	0.22	10	0.20	10	0.26	10	0.20
	10	1.08	10	0.76	10	1.14	10	0.46

Again, the experiments above was repeated with five goal-plan trees in each trial (i.e., the agent progresses five intentions in parallel). The results of *RR*+ and *SAM* are shown in Tables 7.7 and 7.8.

As can be seen, in general, the performance of *RR*+ improves, as there are fewer potential conflicts when the number of top-level goals decreases. The difference between *RR*+ and *SAM* also decreases, as there are fewer synergies when only 5 top-level goals are given. However, with a smaller number of top-level goals, *SAM* still outperforms *RR*+ on both the number of goals achieved and the number of backtracks required.

Table 7.7 *RR+*; all fallible actions fail disruptively, 5 goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	4.96	3.18	4.92	3.82	4.88	5.36	4.70	7.46
5%	4.94	3.34	4.92	4	4.86	5.50	4.70	7.60
	4.94	3.42	4.92	3.94	4.84	5.88	4.66	7.48
	4.86	4.12	4.82	4.88	4.84	6.06	4.68	8.24
10%	4.92	3.32	4.90	4.18	4.90	5.60	4.66	7.66
	4.88	3.78	4.86	4.24	4.84	5.66	4.62	8.10
	4.86	4.78	4.80	5.50	4.80	6.40	4.52	8.92
20%	4.90	3.66	4.90	4.22	4.84	5.84	4.70	8
	4.86	3.96	4.84	4.88	4.76	6.64	4.74	8.36
	4.64	5.54	4.68	6.16	4.72	7.94	4.42	9.32
30%	4.88	4.02	4.88	4.70	4.86	6.06	4.62	8.30
	4.82	4.66	4.76	5.36	4.70	7.62	4.52	8.82
	4.58	8.14	4.62	8.46	4.48	10.10	4.34	12.12

Table 7.8 *SAM*; all fallible actions fail disruptively, 5 goals

	$\#v = 50$		$\#v = 40$		$\#v = 30$		$\#v = 20$	
ND	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
0%	5	0	5	0	5	0	5	0
5%	5	0	5	0	5	0	5	0
	5	0.06	5	0.02	5	0.02	5	0.02
	5	0.18	5	0.10	5	0.08	5	0.06
10%	5	0.04	5	0.04	5	0.02	5	0.02
	5	0.18	5	0.08	5	0.08	5	0.06
	5	0.30	5	0.24	5	0.22	5	0.18
20%	5	0.08	5	0.06	5	0.06	5	0.04
	5	0.24	5	0.20	5	0.14	5	0.12
	5	0.62	5	0.54	5	0.56	5	0.48
30%	5	0.26	5	0.10	5	0.08	5	0.08
	5	0.38	5	0.42	5	0.32	5	0.24
	5	1.88	5	1.38	5	0.92	5	0.74

Nondeterministic Elevator Domain

In this section, the performance of *SAM* was evaluated in a more realistic scenario based on the Miconic-10 elevator domain [40]. As in Section 5.3.2, a single Miconic-10 elevator, which allows passengers to enter their destination floor when calling the elevator using a 10-digit keypad installed in each elevator

lobby, is installed in a building. Passengers could move from floor to floor using the elevator. Requests to travel to a particular floor give rise to top-level goals (one for each request) which are achieved by the BDI agent program. To incorporate nondeterminism, we assume the elevator system is faulty, such that executing the *MoveTo* action will possibly lead the elevator to a wrong floor. We assume that a *MoveTo* action has two possible outcomes: (1) the elevator reaches its intended floor (2) the elevator reaches one floor above the intended floor if it is moving upwards, and reaches one floor below the intended floor if it is moving downwards. (Note, moving to the ground floor and moving to the top floor never fails). For example, suppose an elevator is currently at ground floor. After performing an action to move to the second floor, the elevator may be at floor 2, if the moving action is executed successfully. Or the elevator may be at floor 3, if there is an error while the elevator is moving. The representation of a fallible action MoveTo(l_1) is shown in Figure 7.1, where $At(lift, f_2)$ is the unintended postcondition of this action. For legibility, the estimated execution time for the action is not shown.

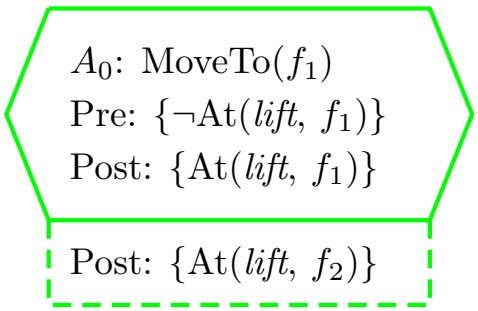


Fig. 7.1 MoveTo Action with unintended postcondition

An example of goal-plan tree used in the experiments reported below is shown in Figure 7.2, which is the same as the one in Section 5.3.2, except that the MoveTo action has two possible postconditions. For legibility, the

deadlines and preferred achievement time for each goal, and the estimated execution time for each action are not shown.

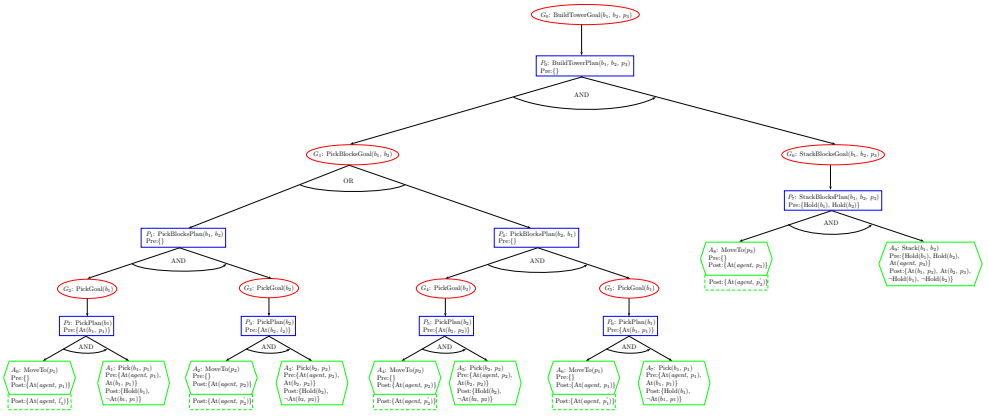


Fig. 7.2 Goal-plan tree for the nondeterministic Miconic-10 elevator domain

For all the experiments reported below, we assume the building has n floors, and there are m passengers distributed at random floors. We generated 50 sets of m goals (each passenger has one goal) to travel between two randomly chosen distinct floors.

In the first experiment, there are 10 passengers (i.e., 10 top-level goals). The number of floors is varied (#Floors) from 4 to 10, and the probability of action failure (ρ) is varied from 10% to 50%. The experimental results of *RR*+ and *SAM* are shown in Tables 7.9 and 7.10.

Table 7.9 *RR*+; non-deterministic elevator domain, 10 goals

	#Floors = 4		#Floors = 6		#Floors = 8		#Floors = 10	
ρ	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
10%	5.32	45.42	3.14	52.06	2.66	52.60	2.18	53.76
25%	4.54	47.90	2.92	52.48	2.26	54.28	1.98	53.62
50%	3.26	53.42	1.86	56.26	1.20	56.90	1.26	56.18

As can be seen, when there are 10 floors and the probability of action failure is 10%, *RR*+ can only achieve 2.18 goals and requires 53.76 backtracks. As the probability of action failure increases, the performance of *RR*+ declines.

Table 7.10 *SAM*; non-deterministic elevator domain, 10 goals

	#Floor = 10		#Floor = 8		#Floor = 6		#Floor = 4	
ρ	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs	#Goals	#Backs
10%	10	0.04	10	0.28	10	0.38	10	0.62
25%	10	0.14	10	0.44	10	0.48	10	0.78
50%	10	0.16	10	0.70	10	1.22	10	1.42

In the elevator domain, *RR*+ fails mainly because the elevator fails to board the passengers. For example, suppose the elevator has two delivery goals. To achieve its first goal, the elevator needs to move to the second floor to board the passenger, and then move to the forth floor to debark the passenger. The plan to achieve the second goal involves moving to the ground floor, boarding the passenger, moving to the third floor and debarking the passenger. Round-Robin scheduling first moves the elevator to the second floor, and then moves the elevator to the ground floor, and then tries to board the people on the second floor and cause an execution failure. Therefore, with fewer floors in the building, the performance of *RR*+ is better (e.g., passengers are on the same floor, or their intended floor are the same). In Table 7.9, the number of goals achieved by *RR*+ increases, as the number of floors in the building decreases. However, even in the best case where there are 4 floors and the probability of action failure is 10%, *RR*+ can only achieve 5.32 goals and requires 45.42 backtracks. In contrast, *SAM* is able to achieve all 10 goals in all cases. When there are 10 floors in the building and the probability of action failure is 10%, *SAM* requires 0.62 backtracks on average. As the probability of action failure and the number of floors increase, the number of backtracks required by *SAM* increase. When there are 4 floors in the building and the probability of execution failure is 10%, *SAM* only requires 0.04 backtracks on average.

In the second experiment, the performance of both approaches were evaluated in a more challenging scenario. In this experiment, the number of the floor in the building was set to 10, and the number of passengers (goals) was varied from 8 to 14. We also vary the probability of action failure from 10% to 50% as in our first experiment. The experimental results of *RR*+ and *SAM* are shown in Tables 7.11 and 7.12

Table 7.11 *RR*+; non-deterministic elevator domain, 10 floors

	#People = 8		#People = 10		#People = 12		#People = 14	
ρ	%Goals	#Backs	%Goals	#Backs	%Goals	#Backs	%Goals	#Backs
10%	26.25%	41.28	21.8%	53.76	24.5%	57.72	21.6%	70.20
25%	17%	43.54	19.8%	53.62	16.8%	58.84	20.7%	75.12
50%	6.25%	42.96	12.6%	56.18	14.3%	65.10	9.14%	79.72

Table 7.12 *SAM*; non-deterministic elevator domain, 10 floor

	#People = 8		#People = 10		#People = 12		#People = 14	
ρ	%Goals	#Backs	%Goals	#Backs	%Goals	#Backs	%Goals	#Backs
10%	100%	0.40	100%	0.62	100%	0.56	100%	0.92
25%	100%	0.48	100%	0.78	100%	0.94	100%	1.22
50%	100%	0.80	100%	1.42	100%	1.70	100%	1.72

As the number of passengers was varied from 8 to 14 and the number of goals achieved is no longer easily comparable, the percentage of goals achieved (%Goals) is used to compare the performance of both approaches. As can be seen, in the best case, when there are 8 passengers, and the probability of action failure is 10%, *RR*+ can achieve 26.25% top-level goals. In contrast, *SAM* is able to achieve all the top-level goals in all cases. As the number of passengers increase, the number of backtracks required increases slightly. In the most difficult case, where there are 14 passengers and the probability of execution failure is 50%, *SAM* only requires 1.72 backtracks.

Overall, *SAM* is able to exploit synergies between intentions to recover from execution failures/ The results of a preliminary empirical evaluation of

SAM suggests it out-performs the failure handling mechanisms used by the state-of-the-art practical BDI agent programming languages. *SAM*'s performance advantage is greatest in those scenarios that are most challenging for conventional approaches (high interaction / high failure), as it is able to exploit positive interactions to reduce backtracking while at the same time avoiding negative interactions between intentions.

7.2 Saving Execution Steps

In addition to recovering from execution failures, synergies between intentions can also be used to save execution steps. Thangarajah et al. [58] argued that the plans which have the same postcondition in different intentions can be merged to improve the performance of an agent (i.e., to save times required to execute a plan). They used summary information to identify the definite and potential postconditions of plans and goals, and merge the plans which have the same postcondition. However, they didn't consider the conflicts between an agent's intentions while exploiting synergies between them (i.e., they assume there are no conflicts between intentions). Their approach is also limited to merging plans rather than actions. In the cases where the plans cannot be merged, it may be possible to merge some of the actions in the plans. In this section, we consider both conflicts and synergies between intentions at the same time, and try to merge actions which have the same or similar postcondition.

7.2.1 Merging Actions

When an agent is progressing multiple intentions which contain the same actions, it may be possible for the agent to merge some of these actions by

appropriate interleaving the progression of its intentions. For example, consider a Mars Rover which has a goal to perform a rock experiment at location A, a goal to perform a rock experiment at location B and another goal to perform a soil experiment at location A. A Mars Rover which doesn't have any reasoning ability might achieve its goals as follows: move to location A, perform the rock experiment, move to location B, perform the rock experiment, move to location A, perform the soil experiment. If the Mars Rover is clever enough, it might achieve its goals in a more sensible way, i.e., move to location A, perform the rock experiment, perform the soil experiment, move to location B, perform the rock experiment. Moreover, as we assumed that programmers will provide all the pre- and postcondition of an action that are relevant to the problem, it is also possible for the agent to merge actions which have exactly the same postcondition.[5] Any effects of an action not included in the action's postconditions is assumed not to affect the execution of other actions.

We formally define the conditions to merge two actions a_1 and a_2 as the following: If two actions a_1 and a_2 have the same postcondition e, where e is their only postcondition, then the agent is allowed to merge these two actions by executing either of them.

7.2.2 Problem Specification and *SAM* Configuration

For simplicity, we assume the deadline and preferred achievement time for all goals are infinite, and the execution of actions are instantaneous as in Chapter 5. In the problem of saving execution steps, the first priority of the agent is to achieve as many goals as possible, as we consider both conflict and synergies

[5]Note that the preconditions of the actions are not necessarily the same.

between intentions at the same time. If the same number of goals are achieved, the agent is required to maximise the number of merged actions (step savings). As we assume the execution of actions are instantaneous (i.e., the execution of all actions require one deliberation cycle), the number of merged actions can be seen as an approximation of the amount of times the agent saves to progress its intentions. Therefore, the problem of saving execution steps is defined as: given a set of goal-plan trees with their corresponding current step pointers, representing the current intentions of the agent $\{t_1, \ldots, t_n\}$, a set of condition variables *Env* representing the current state of the environment and the agent, return a next step of a goal-plan tree t_i to be executed at each deliberation cycle such that the number of goals achieved by the agent is maximised, and the total number of merged actions is also maximised, subject to achieving as many goals as possible.

To save execution steps, the function f_a in *SAM* is configured to return the UCT^{SP} values for a given node (see Equation 4.3), the comparator function f_b compares two nodes on their best simulation results (see Algorithm 11 in Chapter 5), and the function f_μ is defined as follows:

Algorithm 15 Utility function f_μ

```
1: function f_μ(n)
2:     if n.getCompletedGoals() == 0 then
3:         return 0
4:     else
5:         return n.getCompletedGoals() - 1/(n.getMerged() + 1)
```

where n.`getCompletedGoals()` returns the number of goals achieved in node n, and n.`getCompletedGoals()` returns the number of merged actions in n. The utility of a node is then defined as the number of goals achieved minus $1/(n.\texttt{getMerged()}+1)$, where $1/(n.\texttt{getMerged()}+1)$ is in the range

(0, 1] The utility function f_μ therefore ensures the number of goals achieved is always more important than the number of merged actions (i.e., the first priority is to maximise the number of goals achieved, and if the number of goals achieved are the same, prefer the schedule which merges more actions). Note that the utility function returns 0 when there is no top-level goal has been achieved.

7.2.3 Evaluation

In this section, we evaluate the ability of *SAM* to merge actions in different intentions. we compare the performance of *SAM* scheduler with Round-Robin scheduling (RR) in both synthetic domains and in a more realistic scenario based on the classic Blocks World domain.

The performance of *SAM* and RR is evaluated using two metrics: the number of goals achieved, and the number of merged actions. We assume RR could merge actions. For example, if two actions A_1 and A_2 are scheduled to be executed sequentially by RR, and their postconditions are the same, then the agent merges these two actions by only executing the first action A_1. For all the experiments reported below, both *SAM* and RR do not have failure handlers, *SAM* is configured to perform 100 iterations ($\alpha = 100$) and 10 random simulations at each iteration (β = 10).

Synthetic domain

As in Chapter 5 and 6, our first sets of experiments are based on sets of randomly-generated, synthetic goal-plan trees representing the current intentions of an agent in a simple static environment. The sets of synthetic goal-plan

trees are generated using GenGPT, as discussed in Chapter 5. By controlling the number of variables $\#v$ that may appear in each goal-plan tree (as the pre- and postconditions of actions and plans), and the number of actions in each plan $\#\alpha$, we can evaluate the performance of each scheduling algorithm under different conditions.

Environment As in previous chapters, the environment is defined as a set of propositions that may appear as pre- or postconditions of plans and actions in a goal-plan tree. If the precondition of an action selected for execution does not hold in the environment, execution of the intention fails, and the corresponding top-level goal is dropped. If the precondition holds, the action is executed, and the environment updated with the postconditions of the action.

Goal-Plan Trees For each experiment reported below, we generated 50 sets of 10 goal-plan trees. The depth of each goal-plan tree is set to 5. Each (sub)goal has two plans to achieve it. Each plan consists a sequence of actions, followed by a subgoal. The maximum p-effect size is ∞. The number of environment variable is set to 20. Other parameters of the trees were varied in each experiment as detailed below.

Results

Experiment 1 In the first experiment, the number of actions in each plan was set to 3 (i.e., $\#\alpha = 3$), and the number of environment variables ($\#v$) was varied from 20 to 50. The performance of RR and *SAM* are shown in Table 7.13. #Goals represents the average number of goals achieved by RR or *SAM*. #Merge represents the average number of actions merged by RR or *SAM*.

As can be seen, when there are 50 environment variables, RR is able to achieve 4.74 goals and merge 0.52 actions on average. The number of goals achieved by RR decreases as the number of environment variables decreases. This is because with fewer environment variables, it is more likely there are conflicts between intentions. In the worse case where there are 20 environment variables, RR can only achieve 2.68 goals. Conversely, the number of actions merged by RR increases slightly as the number of environment variables decreases. As with fewer environment variables, actions are more likely to have the same postcondition, and RR can merge 1 or 2 actions in each trial by chance. However, even in the best case where there are 20 environment variables, RR can only merge 0.96 actions.

$\#v$	RR		*SAM*	
	#Goals	#Merge	#Goals	#Merge
50	4.74	0.52	10	15.26
40	4.16	0.60	10	17.54
30	3.59	0.76	10	21.36
20	2.68	0.96	10	28.08

Table 7.13 Goals & the number of merged actions with decreasing number of environment variables, each plan contains 3 actions

Unlike RR, *SAM* is able to achieve all 10 goals in all cases. When there are 50 environment variables, *SAM* is able to merge 15.26 actions on average. As the number of environment variables decreases, the number of actions merged by *SAM* increases. Compared to RR, *SAM* is able to merge more actions. In the best situation (when there are 20 variables), *SAM* is able to save about 20%

of the execution steps (28.08 out of 150), while RR can only merge about one action (by chance).

Experiment 2 In the second experiment, the number of environment variables $\#v$ was set to 50, and the number of actions in each plan was varied from 3 to 7. The experimental results are shown in Table 7.14.

$\#\alpha$	RR		*SAM*	
	#Goals	#Merge	#Goals	#Merge
3	4.74	0.52	10	15.26
5	2.91	0.52	10	23.24
7	2.00	0.43	10	32.50

Table 7.14 Goals & the number of merged actions with increasing number of actions in each plan

As can be seen, the number of goals achieved and the number of actions merged by RR decrease as the number of actions in each plan increases. This is because there are more potential conflicts between intentions if each plan has more actions. In the worse case, when there are 7 actions in each plan, RR can only achieve 2 goals on average. In contrast, *SAM* is able to achieve all goals in all cases. As the number of actions in each plan increases (and thus the number of actions that have the same postcondition increases), the number of actions merged by *SAM* increases. In the best case, when there are 7 actions in each plan, *SAM* is able to merge 32.5 actions on average.

Experiment 3 The performance of RR and *SAM* were then evaluated with increasing number of top-level goals. In Experiment 3, the number of environment variables was set to 50, the number of actions in each plan was set to 3, and the number of top-level goals was varied from 8 to 14. The experimental results for both RR and *SAM* are shown in Table 7.15.

#*T*	RR		*SAM*	
	%Goals	#Merge	%Goals	#Merge
8	53.5%	0.56	100%	13.86
10	47.4%	0.52	100%	15.26
12	42.3%	0.60	100%	23
14	38%	0.69	100%	25.58

Table 7.15 Goals & the number of merged actions with increasing number of top-level goals

As the number of top-level goals was varied from 8 to 14 and the number of goals achieved is no longer easily comparable, the percentage of goals achieved (%Goals) is used to compare the performance of both approaches. As can be seen, the percentage of goals achieved by RR decrease, as the number of top-level goals increases. In the best situation, where there are 8 top-level goals, RR is able to achieve 53.5% of the goals on average. In contrast, *SAM* is able to achieve all top-level goals in all cases. The number of actions merged by both RR and *SAM* increases as the number of top-level goals increase. However, *SAM* is able to merge significantly more actions than RR (e.g., 25.58 vs 0.69 when there are 14 top-level goals).

Blocks World Domain

In this section, the performance of *SAM* was evaluated in a more realistic scenario based on the classic Blocks World domain. As discussed in Chapter 6, an agent must pick blocks and stack them to build towers. Each top-level goal involves picking up two blocks from two different locations and building a tower at a specified location. A goal-plan tree used in the experiment is shown in Figure 6.1 in Chapter 6. Unlike the blocks world domain introduced in Chapter 6, there is no deadline and preferred achievement time for the agent to achieve its goals.

In the experiments reported below, we assume there are n cells. Each cell represents one location in the Blocks World domain. The first n/2 cells contain blocks with different colours. Blocks which are stacked at the same cell have the same colour. Each top-level goal involves picking up two blocks with specified colours in the first n/2 cells and building a tower at a specified empty cell. For example, a top-level goal g_1 may require an agent to pick up a red block at cell 1 and a blue block at cell 2, and use these two blocks to build a tower at cell 4. Similarly, the agent may also have another goal to pick up a red block at cell 1 and a green block at cell 2, and stack these two blocks to build a tower at cell 5. In this situation, the agent may go to cell 1 once and pick up two red blocks by merging the MoveTo action. For each experiment reported below, the colour of the blocks and the location of the tower required in each top-level goal are generated randomly. As in Chapter 6, we assume the number of blocks the agent can carry is unlimited.

Experiment 4 In the first experiment, the number of cells is set to 20. Each of the first 10 cells is stacked with #*goals* blocks with different colour, where #*goals* is the number of top-level goals. All other cells are empty. The number of top-level goals (#*goals*) was varied from 10 to 20, and the experimental results are shown in Table 7.16. As the number of goals achieved is no longer easily comparable, we use %Goals to represent the percentage of goals achieved by the agent.

As can be seen, *SAM* is able to achieve all top-level goals in all cases, while RR can only achieve less than 10% goals on average. For both approaches, the number of merged actions increases, as the number of top-level goals increases (as there are fewer chances to merge actions). However, *SAM* is able to merge

#goals	RR		*SAM*	
	%Goals	#Merge	%Goals	#Merge
10	10%	1.12	100%	15.08
15	6.8%	1.60	100%	27.72
20	5.1%	1.84	100%	41.26

Table 7.16 Goals & the number of merged actions with increasing number of top-level goals, the number of cells is set to 20

a significantly larger number of actions than RR (e.g., 41.26 vs 1.12 in the best case where there are 20 top-level goals).

Experiment 5 In our second experiment, the number of top-level goals is set to 10 and the number of cells was varued from 10 to 30. The number of cells corresponds to the number of environment variables in the synthetic domain. With fewer number of cells, the top-level goals are more likely to require the blocks of the same colour, and thus cause synergies. The experiment results for both RR and *SAM* are shown in Table 7.17.

#cells	RR		*SAM*	
	#Goals	#Merge	#Goals	#Merge
10	1.08	2.54	10	20.50
20	1	1.12	10	15.08
30	1	0.72	10	11.32

Table 7.17 Goals & the number of merged actions with increasing number of top-level goals, the number of cells is set to 10

As can be seen, *SAM* is able to achieve all 10 top-level goals in all cases, while RR can only achieve about one goal on average. For both approaches, the number of merged actions decreases, as the number of cells increases (as the blocks required in different top-level goals are more likely to have different colours). However, *SAM* is able to merge a significantly larger number of actions than RR in all cases.

7.3 Summary

In this chapter, we show how *SAM* can be used to exploit synergies between intentions. The synergies are used for both recovering from execution failures, and for saving execution steps. In Section 7.1, *SAM* is used to schedule the progression of intentions which contain fallible actions, where synergies between intentions are used to recover from execution failure. In Section 7.2, *SAM* is used to merge actions that have the same postconditions to save execution steps. We evaluate the performance of *SAM* in both synthetic domains and a more realistic domain based on the Miconic-10 elevator domain (in Section 7.1) and the classic Blocks World domain (in Section 7.2). The experiment results indicate that the potential benefits of using *SAM* to exploit synergies between intentions is quite significant. In the next chapter, we will discuss the computational overhead of *SAM*.

Chapter 8

Computational Overhead

In this chapter, we present an preliminary evaluation of the performance of *SAM*. We evaluate the performance of *SAM* with different computational budgets in the synthetic domains in Chapter 8. The results suggest the *SAM* can achieve all goals even when the computational budget is very small (i.e., 50 iterations and 5 simulations in each iteration), and only requires 9ms time to return a full schedule. If additional time is given, *SAM* can improve its performance by running additional simulations.

8.1 Anytime *SAM*

One advantage of BDI agents is their ability to respond quickly to the changes in the environment. At each deliberation cycle, the agent needs to decide which intention to progress and how to progress it. If the approach to make such decisions takes a long time, then there will a delay in choosing the next action. Our proposed scheduler *SAM* is based on an anytime algorithm MCTS (and its variant SP-MCTS), which can return a result at any moment in time.

The computational overhead of *SAM* depends on the value of α (i.e., how many iterations the algorithm performs) and β (how many simulations are run in each iteration), which can be specified by programmers for different applications.[1] For a set of 10 goal-plan trees, each with 15 actions from the top-level goal to the last action, *SAM* requires 35 milliseconds to compute a complete schedule. As discussed, *SAM* builds a search tree at each deliberation cycle. As we show in the evaluation chapters, in many cases *SAM* is able to find an interleaving that achieves all the agent's goals. As the best interleaving (best simulation) is stored in each node, if the environment does not change, all actions are infallible, and there is no estimation error on the execution time of each action, then the next step in this interleaving can simply be returned at subsequent deliberation cycles without further computation. It is only necessary to build a new search tree when the environment changes or the agent adopts a new top-level goal. In low dynamism environments, this means the already small overhead of *SAM* can be amortised over multiple execution steps. In the previous example, if the environment is static and all actions are infallible (and require their expected execution time has no estimation error if goals have deadlines), this schedule only needs to be recomputed when the agent adopts a new top-level goal, so in the best case (no new top-level goals during the execution of the current intentions) the overhead of *SAM* is effectively amortised over the execution of all 10 intentions, and the amortised CPU time required to choose which intention to execute at each cycle is approximately 0.2 milliseconds. Otherwise, the interleaving must be

[1] Alternatively, programmers can customise the amount of time used by the scheduler rather than α and β. However, it is not in the scope of this thesis.

recomputed at every deliberation cycle, so the cost of intention selection will be correspondingly higher.

8.2 Experiments

To investigate the relationship between the performance of the *SAM* scheduler, and its computational budget (i.e., the values of α and β), *SAM* was evaluated using sets of randomly-generated, synthetic goal-plan trees. For the experiments reported below, we generated 50 sets of 10 goal-plan trees. The depth of each goal-plan tree is set to 5. Each (sub)goal has two plans to achieve it. Each plan consists a sequence of three actions, followed by a subgoal. The maximum p-effect size is ∞. The number of environment variable is set to 20.

The scenarios of saving execution steps in Section 7.2.3 was chosen and used to evaluate the performance of *SAM*, as the effects on the number of merged action is more intuitive. *SAM* is configured the same as in Section 7.2.3, except the computational budget. The computational budget of *SAM* was varied from (50, 5) (i.e., α = 50 and β = 5) to (500, 50). In all cases, *SAM* is able to achieve all 10 goals, and the number of merged actions in each setting is shown in Figure 8.1.

As can be seen, the performance of *SAM* increases, as it runs more iterations and simulations. The improvement of the results decreases as the number of iterations and simulations increase. However, even with the setting $\alpha = 50$, $\beta = 5$, *SAM* is able to achieve all 10 goals and merge 25.8 actions on average.

The computational overhead of *SAM* with each pair of α and β is shown in Figure 8.2. As expected, the computational overhead increases, as the number of iterations and simulations increases.

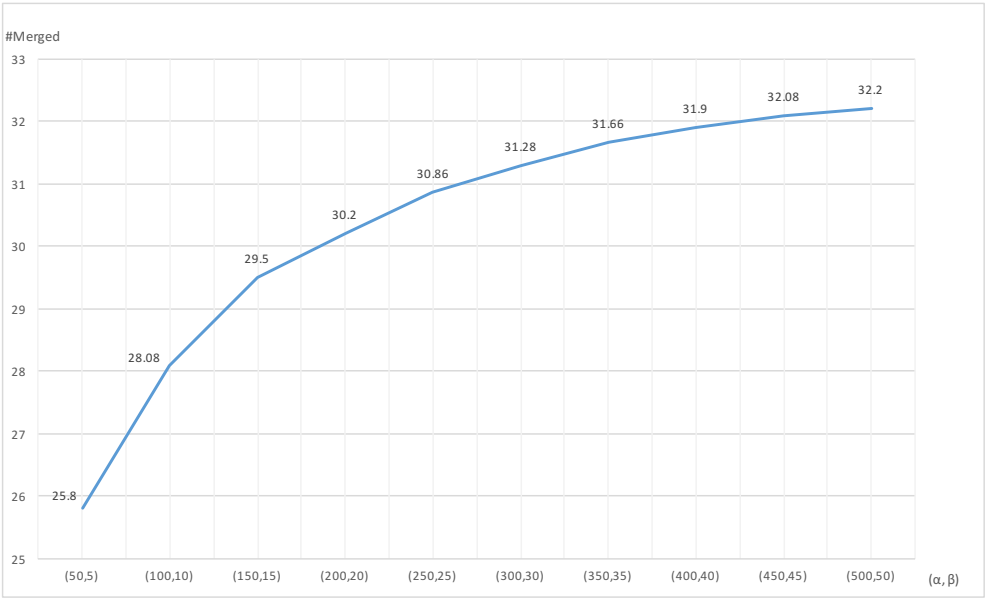


Fig. 8.1 Number of merged actions with increasing α and β

Overall, *SAM* requires very low computational overhead to schedule the synthetic domains introduced in previous chapters (about 35 ms). If additional time is given, *SAM* can improve its performance by running additional simulations. However, the improvements decrease as the amount of time spent on the simulations increases.

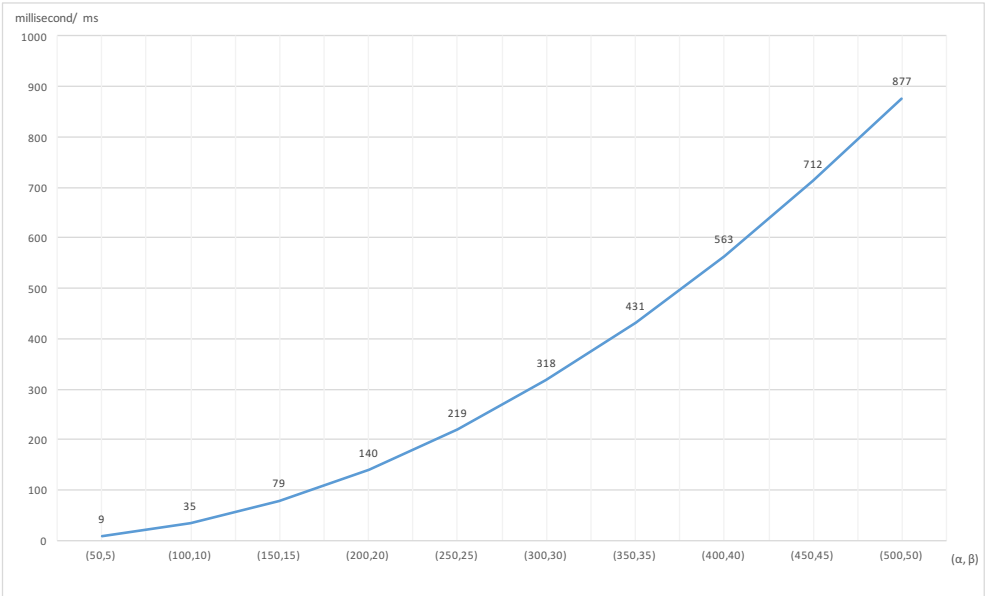


Fig. 8.2 Computational overhead with increasing α and β

Chapter 9

Conclusion and Future Work

Agent-based systems are becoming increasingly popular as a paradigm for building software that exhibits a high degree of autonomy and operates in dynamic, complex domains. In such software or applications, an agent usually pursues multiple goals in parallel. The interleaving of the execution steps in different intentions to achieve these goals may cause either conflicts where one or more goals fail, or synergies where the achievement of one goal assists the achievement of other goals. A rational agent should have the ability to reason about such interactions and achieve its goals in a way that best satisfies programmers' requirements. However, there are no general approaches implemented in the state of the art BDI agent programming languages that can avoid conflicts and exploit synergies at the action level. The approaches proposed in the literature to reason about the interactions between intentions also have certain limitations.

Summary of contributions In this thesis, we consider the problem of intention progression for BDI agents which concerns both the plan selection

problem (i.e., which plan should be applied to achieve a goal) and the intention selection problem (i.e., which intention should be progressed next). To make it easier to integrate our scheduler into different agent programming languages, the intentions of an agent are represented using a set of goal-plan trees with their current step pointer. We extended the existing goal-plan trees structure in [60, 57, 58, 56] to include an explicit representation of actions which allows the interleaving of intentions at the level of individual actions. We proposed a domain-independent scheduler *SAM* based on the stochastic scheduling approach Monte-Carlo Tree Search and its variant Single-Player Monte-Carlo Tree Search, which can be integrated into agent-based systems that follow the BDI model of agents.. The key idea of the *SAM* scheduler is to use random sampling to estimate the utility of progressing each intention, and to progress the intention which is most "promising" with respect to a user-defined utility function μ, a UCB value calculation function f_a and a best child selection function f_b. With appropriate choice of function μ, f_a and f_b, both conflicts and synergies between intentions can be addressed in the simulation, e.g., interleavings which cause conflicts will get low utility values, and saving execution steps or recovering from execution failure increase the utility values of an interleaving.

We showed how *SAM* can be used to avoid conflicts between intentions in Chapter 5, and to schedule with real time constraints in Chapter 6. In addition, we also showed how *SAM* can be used to exploit synergies between intentions in Chapter 7. The performance of *SAM* was evaluated in both synthetic and real-world domains. To evaluate *SAM*, we developed GenGPT, a package to generate synthetic goal-plan trees. By controlling the input

parameters of GenGPT, we could generate goal-plan trees of different shapes and properties. There is a lack of widely accepted benchmark agent programs in the BDI literature. In this thesis we use the Miconic-10 elevator domain and the classic Blocks World domain to evaluate the performance of *SAM*. The Miconic-10 elevator domain was originally formulated as an HTN planning problem. HTN planning problems are a fruitful source of 'realistic' example BDI programs. The planning community has produced an extensive set of benchmark problems, and there are well established procedures for translating planning problems into BDI agent programs. For example, de Silva and Padgham [27] describe how to convert JSHOP (a Java version of the SHOP HTN planner) plans into JACK programs, and evaluate the performance of JSHOP and JACK based on HTN planning domains. Meneguzzi and Luck [42] present an algorithm to convert STRIPS plans to AgentSpeak(L) programs. There is also work on how to translate BDI agent programs to HTN planning domains [49]. In each of the experiments, we compared the performance of *SAM* with representative approaches to intention progression in practical BDI agent languages (i.e., RR, FIFO), or in the theoretical literature (i.e., SI, C, AgentSpeak(RT)). The experiment results suggested that the potential benefits of using *SAM* are quite significant.

Compared to the existing approaches to intention progression, *SAM* has several advantages. First, it allows the developer to customise their requirements for different problem domains. Existing approaches in the literature such as SI, or C are designed for solving one type of problem e.g., resolving the conflicts between intentions. In situations where the developer has a particular objective (or requirement), modifications may be necessary for

these approaches to work. Second, *SAM* is able to schedule the execution of an agent's intention at the action level, i.e., *SAM* interleaves the execution of actions rather than plans. Intention scheduling approaches like SI and C are limited to schedule the progression of an agent's intention at the level of individual plans. Conflicts which cannot be solved at the plan level may have solutions at the action level. Moreover, *SAM* takes both infallible and fallible actions into consideration. Most of the existing approaches assume plans and actions always achieve their desired effects (postcondition). However, *SAM* is able to schedule fallible actions and recover from execution failure by exploiting synergies between intentions. Finally, *SAM* is an anytime algorithm which means it can return a results at any moment in time. The developer can specify the computational budget of *SAM* for different applications.

Limitations While *SAM* has several advantages compared to the existing approaches, it also has certain limitations. For example, the design of the *SAM* scheduler is based on the Belief-Desire-Intention agent architecture and relies on the agent program being translated and represented using the goal-plan trees structure. Therefore, for agents developed using other agent architectures, it may be not easy to translate their intentions to goal-plan trees. In order for *SAM* to work, we also require additional developer effort in defining the pre- and postconditions of actions. Some BDI programming languages already require the developer to specify the pre and postconditions of actions, e.g., actions in GOAL [34], belief-update actions in 2APL [23]. For languages such as Jason, our experimental results indicate the kinds of scenarios where the additional effort of specifying the pre- and postconditions of actions would be worthwhile in terms of increased agent performance. In this thesis, we

assumed plans consist of a sequence of actions and subgoals (i.e., actions and subgoals in a plan are executed or achieved sequentially). Some BDI agent programming languages such as JACK allow the actions and subgoals in a plan to be executed in parallel. Moreover, we also didn't consider other forms of subgoals, such as conditional subgoals (i.e., conditional subgoals need to be achieved only if some condition is true).

Future Work In this thesis, we focussed on achievement goals which represent states of the world that the agent wants to achieve. Once achieved, the goal is dropped with success. Maintenance goals that represent the states of the world that the agent must maintain for a given period of time are not considered. If there are maintenance goals, in addition to interleaving the progression of an agent's intentions such that all intentions are successfully executed, the *SAM* scheduler also needs to maintain the state of the world as specified by the maintenance goals (e.g., any actions that conflict with the maintenance goal should be delayed until the maintenance goal is dropped).

SAM was proposed specifically for a single agent with multiple concurrent goals. One possible line of future work would be to extend *SAM* to multi-agent systems. In multi-agent systems, the interactions between intentions and the changes in the environment are more complex, as there are multiple agents and each of them could have multiple concurrent goals. If the dynamism of the environment is primarily due to the actions of other agents (e.g., StarCraft), then it depends on whether the interactions of the agents are primarily cooperative or competitive. If the interactions between the agents are competitive, one possible way to use *SAM* is to model the other agents like the environment, and consider what action to choose based on what actions the other agents may

choose. The model of how other agents choose actions can be very simple, or, if information about the agents' goals and plans and sufficient computation are available, the *SAM* scheduler could estimate other agents' choices and use them in its simulation. If the interactions between agents are cooperative, then in addition to the distributed scheduling approach described above, we can also use centralised scheduling for all agents. However, this is often not desirable (e.g., failing of an agent would result in the failure of the system) and may significantly increase the computational overhead, as each agent in a multi-agent system may have multiple concurrent goals which increases the number of possible combinations of next steps.

In Section 5.3.1, *SAM* was evaluated in dynamic environments where the changes in the environment is not primarily due to the actions of other agents, and *SAM* didn't have any knowledge of the environment changes, i.e., *SAM* runs simulations as if there is no change in the environment. As the dynamism of the environment increases, the performance of *SAM* declines. The performance of *SAM* can be improved if a good model of how the environment changes is available. For example, if the environment model can tell what changes of the environment are likely to happen at what time, then the *SAM* scheduler could consider these changes when running simulation and thus improve its performance. The environment model doesn't have to tell the exactly changes in each deliberation cycle, even some rough information may help *SAM* to work in dynamic environments. The more information an environment model can give, the better performance *SAM* can produce in a dynamic environment. However, if the environment becomes extremely

dynamic, e.g., nearly all the variables in the environment change their values in each deliberation cycle, then scheduling may be of limited value.

Our current implementation of *SAM* simply runs unbiased random simulations in the simulation phase. This has the advantage that it is simple, requires no domain-specific knowledge and will most likely cover different areas of the search space. However, given domain-specific knowledge, it is possible for developers to enhancements the simulation phase of *SAM* to make it more "realistic" and thus to improve the performance of *SAM*. For example, an elevator is not desired to move to other floors before debarking the customers who reach their destination floor, even though the move action is legal. One possible way to enhancements the simulation phase is to hand-code rule-based policies for a given domain. Therefore, another line of future work is to allow developers to customise simulation strategies for different applications.

Finally, another line of future work is to incorporate *SAM* into a practical agent programming platform (e.g., Jason). To incorporate *SAM* into a practical agent platform, we need to restructure the agent's deliberation cycle. As discussed in Section 4.4.1, *SAM* effectively merges the plan selection and intention selection of the standard deliberation cycle into a single process, and *SAM* is used for both plan selection and intention selection. Adding SAM code to agent deliberation cycle should be straightforward[1]. The only other things that are needed for the *SAM* to work in a practical agent programming platform are to translate the agent program into a set of goal-plan trees, to extend the agent program with the descriptions of pre- and postcondition of actions[2] and

[1]Preliminary discussions with Rafael Bordini regarding the integration of *SAM* into the agent programming platform Jason suggest that the changes required to the Jason code will be fairly minor.

[2]This would not be necessary for all languages, e.g., 2APL already has pre- and postconditions for its belief update actions.

the specification of the functions that define the quality of a schedule (see page 81), and, if required, to extend the notion of goals in the agent programming language to support deadlines. For languages that support parallel constructs, *SAM* would also have to be extended to include the representation of parallel constructs[3]. For example, the parallel composition of actions and subgoals was introduced in [75], where there is no assumptions about the execution of a parallel composition, i.e., the actions and subgoals in a parallel composition may overlap in any of the ways defined in [5], or their execution may be arbitrarily interleaved. Once the key steps (i.e., translating agent programs to GPTs and adding pre- and post conditions) in integrating SAM into an agent programming have been done, it will also be possible to evaluate SAM using large real agent programs[4]. The challenge is to define the functions (f_μ, f_a and f_b) which capture the desired behaviour of the target agent programs, because this is typically not explicit in the program itself.

[3] *SAM* does consider parallel execution of actions and goals from different intentions, but does not have the representation of parallel goals and actions from the same intention.

[4] *SAM* was already evaluated using simple real world examples, e.g., the elevator domain.

References

[1] Abramson, B. (1990). Expected-Outcome: A General Model of Static Evaluation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(2):182–193.

[2] Alechina, N., Behrens, T., Dastani, M., Hindriks, K., Hubner, J., Logan, B., Nguyen, H., and van Zee, M. (2013). Multi-cycle query caching in agent programming. In desJardins, M. and Littman, M., editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013)*, volume 1, pages 32–38, Bellevue, Washington. AAAI, AAAI Press.

[3] Alechina, N., Dastani, M., and Logan, B. (2012). Programming norm-aware agents. In Conitzer, V., Winikoff, M., Padgham, L., and van der Hoek, W., editors, *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, volume 2, pages 1057–1064, Valencia, Spain. IFAAMAS.

[4] Alechina, N., Dastani, M., Logan, B., and Meyer, J.-J. C. (2008). Reasoning about agent deliberation. In Brewka, G. and Lang, J., editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 16–26, Sydney, Australia. AAAI.

[5] Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

[6] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256.

[7] Biernatzki, R., Bitzer, B., Zivanovic, R., EIS, A., and EIS, S. (2004). Agent Technology used in Power Systems. In *Proceedings of AFRICON 2004 (IEEE Conference)*, volume 2, pages 705–710, Gaborone, Botswana. IEEE.

[8] Bordini, R., Bazzan, A. L. C., Jannone, R. d. O., 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 First International Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 1294–1302.

[9] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley.

[10] Bratman, M. (1984). Two Faces of Intention. *The Philosophical Review*, 93:375–405.

[11] Bratman, M. E. (1990). What Is Intention? In Cohen, P. R., Morgan, J., and Pollack, M. E., editors, *Intentions in Communication*, chapter What is intention?, pages 15–32. The MIT Press, Cambridge, MA.

[12] Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Liebana, D. P., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43.

[13] Calhau, F., Pereira, L., Costa, P., and Botelho, L. M. (2004). Agents Technology Extended with Mobile Devices. In *First International Workshop,MATA*, pages 118–126, Florianópolis, Brazil. Springer.

[14] Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008). Monte-Carlo tree search: A new framework for game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, Stanford, California, USA.

[15] Chaslot, G., Saito, J., Bouzy, B., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2006). Monte-Carlo strategies for computer Go. In *Proceedings of the 18th Belgian-Dutch Conference on Artificial Intelligence*, pages 83–90, Namur, Belgium.

[16] Chen, B. and Cheng, H. H. (2010). A Review of the Applications of Agent Technology in Traffic and Transportation Systems. *IEEE Transactions on Intelligent Transportation Systems*, 11(2):485–497.

[17] Clement, B. J. and Durfee, E. H. (1999). Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. In Hendler, J. and Subramanian, D., editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence*, pages 495–502, Orlando, Florida, USA. AAAI Press / The MIT Press.

[18] Clement, B. J. and Durfee, E. H. (2000a). Performance of Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. In Castelfranchi, C. and Lespérance, Y., editors, *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop*, pages 213–227, Boston, MA, USA. Springer.

[19] Clement, B. J. and Durfee, E. H. (2000b). Performance of Coordinating Concurrent Hierarchical Planning Agents Using Summary Information.

In *4th International Conference on Multi-Agent Systems*, pages 373–374, Boston, MA, USA. IEEE Computer Society.

[20] Clement, B. J., Durfee, E. H., and Barrett, A. C. (2007). Abstract Reasoning for Planning and Coordination. *J. Artif. Intell. Res. (JAIR)*, 28:453–515.

[21] Coquelin, P. and Munos, R. (2014). Bandit Algorithms for Tree Search. *CoRR*, abs/1408.2028.

[22] Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-Directed Requirements Acquisition. *Sci. Comput. Program.*, 20(1-2):3–50.

[23] Dastani, M. (2008). 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.

[24] Dastani, M., de Boer, F. S., Dignum, F., and Meyer, J. C. (2003a). Programming agent deliberation: an approach illustrated using the 3apl language. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS*, pages 97–104, Melbourne, Victoria, Australia. ACM.

[25] Dastani, M., van Riemsdijk, B., Dignum, F., and Meyer, J.-J. C. (2003b). A Programming Language for Cognitive Agents Goal Directed 3APL. In Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003*, volume 3067 of *Lecture Notes in Computer Science*, pages 111–130, Melbourne, Australia. Springer.

[26] Dastani, M., van Riemsdijk, M. B., and Meyer, J. C. (2005). Programming Multi-Agent Systems in 3APL. In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer.

[27] de Silva, L. and Padgham, L. (2004). A Comparison of BDI Based Real-Time Reasoning and HTN Based Planning. In Webb, G. I. and Yu, X., editors, *AI 2004: Advances in Artificial Intelligence, 17th Australian Joint Conference on Artificial Intelligence*, volume 3339, pages 1167–1173, Cairns, Australia. Springer.

[28] Decker, K. S. and Lesser, V. R. (1993). Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 2:215–234.

[29] d'Inverno, M., Luck, M., Georgeff, M. P., Kinny, D., and Wooldridge, M. (2004). The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53.

[30] Doan, T. T., Alechina, N., and Logan, B. (2011). The agent programming language meta-APL. In Dennis, L. A., Boissier, O., and Bordini, R. H., editors, *Proceedings of the Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011)*, pages 72–87, Taipei, Taiwan.

[31] Doan, T. T., Yao, Y., Alechina, N., and Logan, B. (2014). Verifying heterogeneous multi-agent programs. In Bazzan, A. L. C., Huhns, M. N., Lomuscio, A., and Scerri, P., editors, *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14*, pages 149–156, Paris, France. IFAAMAS/ACM.

[32] Georgeff, M. P. and Ingrand, F. F. (1990). Real-time reasoning: the monitoring and control of spacecraft systems. In *Proceedings of the sixth conference on Artificial intelligence applications*, pages 198–204, Santa Barbara, CA. IEEE.

[33] Georgeff, M. P. and Lansky, A. L. (1987). Reactive Reasoning and Planning. In Forbus, K. D. and Shrobe, H. E., editors, *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 677–682, Seattle, WA. Morgan Kaufmann.

[34] Hindriks, K. V. (2009). Programming rational agents in GOAL. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., and Bordini, R. H., editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US.

[35] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J. C. (1999). Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401.

[36] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J. C. (2000). Agent Programming with Declarative Goals. In Castelfranchi, C. and Lespérance, Y., editors, *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL*, pages 228–243, Boston, MA, USA. Springer.

[37] Katehakis, M. N. and Jr., A. F. V. (1987). The Multi-Armed Bandit Problem: Decomposition and Computation. *Math. Oper. Res.*, 12(2):262–268.

[38] Katsuno, H. and Mendelzon, A. O. (1991). On the Difference between Updating a Knowledge Base and Revising It. In Allen, J. F., Fikes, R., and Sandewall, E., editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991.*, pages 387–394, Cambridge, MA, USA. Morgan Kaufmann.

[39] Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In Fürnkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *17th European Conference on Machine Learning*, pages 282–293, Berlin, Germany. Springer.

[40] Koehler, J. and Schuster, K. (2000). Elevator Control as a Planning Problem. In Chien, S., Kambhampati, S., and Knoblock, C. A., editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, pages 331–338, Breckenridge, CO, USA. AAAI.

[41] Matsumoto, S., Hirosue, N., Itonaga, K., Ueno, N., and Ishii, H. (2010). Monte-Carlo tree search for a reentrant scheduling problem. In *40th International Conference on Computers and Industrial Engineering (CIE)*, pages 1–6, Awaji. IEEE.

[42] Meneguzzi, F. and Luck, M. (2013). Declarative planning in procedural agent architectures. *Expert Syst. Appl.*, 40(16):6508–6520.

[43] Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In Bordini, R., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer US.

[44] Rajan, K., Bernard, D. E., Dorais, G., Gamble, E. B., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P. P., Rouquette, N. F., Smith, B. D., Taylor, W., and Tung, Y.-W. (2000). Remote Agent: An Autonomous Control System for the New Millennium. In Horn, W., editor, *Proceedings of 14th European Conference on Artificial Intelligence*, pages 726–730, Berlin, Germany. IOS Press.

[45] Rao, A. S. (1996). AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In de Velde, W. V. and Perram, J. W., editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55, Eindhoven, The Netherlands. Springer.

[46] Rao, A. S. and Georgeff, M. P. (1991). Modeling Rational Agents within a BDI-Architecture. In Allen, J. F., Fikes, R., and Sandewall, E., editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, Cambridge, MA, USA. Morgan Kaufmann.

[47] Rao, A. S. and Georgeff, M. P. (1992). An Abstract Architecture for Rational Agents. In Nebel, B., Rich, C., and Swartout, W. R., editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 439–449, Cambridge, MA. Morgan Kaufmann.

[48] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach, 3rd ed.* Pearson Education.

[49] Sardiña, S., de Silva, L., and Padgham, L. (2006). Hierarchical planning in BDI agent programming languages: a formal approach. In Nakashima, H., Wellman, M. P., erhard Weiss, and Stone, P., editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1001–1008, Hakodate, Japan. ACM.

[50] Sardiña, S. and Padgham, L. (2011). A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70.

[51] Schadd, F. C. (2009). Monte-Carlo search techniques in the modern board game Thurn and Taxis. Master's thesis, Maastricht University.

[52] Schadd, M. P. D., Winands, M. H. M., Tak, M. J. W., and Uiterwijk, J. W. H. M. (2012). Single-player Monte-Carlo tree search for SameGame. *Knowl.-Based Syst.*, 34:3–11.

[53] Shaw, P. H. (2010). *Reasoning about Goal-Plan Trees in Autonomous Agents: Development of Petri net and Constraint-Based Approaches with Resulting Performance Comparisons*. PhD thesis, Durham University.

[54] Shaw, P. H. and Bordini, R. H. (2007). Towards Alternative Approaches to Reasoning About Goals. In Baldoni, M., Son, T. C., van Riemsdijk, M. B., and Winikoff, M., editors, *Declarative Agent Languages and Technologies V, 5th International Workshop*, volume 4897, pages 104–121, Honolulu, HI, USA. Springer.

[55] Shaw, P. H. and Bordini, R. H. (2010). An Alternative Approach for Reasoning about the Goal-Plan Tree Problem. In Coelho, H., Studer, R., and Wooldridge, M., editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence*, volume 215, pages 1035–1036, Lisbon, Portugal. IOS Press.

[56] Thangarajah, J. and Padgham, L. (2011). Computationally Effective Reasoning About Goal Interactions. *Journal of Automated Reasoning*, 47(1):17–56.

[57] Thangarajah, J., Padgham, L., and Winikoff, M. (2003a). Detecting & Avoiding Interference Between Goals in Intelligent Agents. In Gottlob, G. and Walsh, T., editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 721–726, Acapulco, Mexico. Morgan Kaufmann.

[58] Thangarajah, J., Padgham, L., and Winikoff, M. (2003b). Detecting & Exploiting Positive Goal Interaction in Intelligent Agents. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003*, pages 401–408, Melbourne, Victoria, Australia. ACM.

[59] Thangarajah, J., Sardina, S., and Padgham, L. (2012). Measuring plan coverage and overlap for agent reasoning. In van der Hoek, W., Padgham, L., Conitzer, V., and Winikoff, M., editors, *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, pages 1049–1056. IFAAMAS.

[60] Thangarajah, J., Winikoff, M., Padgham, L., and Fischer, K. (2002). Avoiding Resource Conflicts in Intelligent Agents. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence*, pages 18–22, Lyon, France. IOS Press.

[61] van Riemsdijk, B., van der Hoek, W., and Meyer, J. C. (2003). Agent programming in dribble: from beliefs to goals using plans. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS*, pages 393–400, Melbourne, Victoria, Australia. ACM.

[62] Vikhorev, K., Alechina, N., Bordini, R., and Logan, B. (2011a). An operational semantics for AgentSpeak(RT) (preliminary report). In Sakama, C., Sardina, S., Vasconcelos, W., and Winikoff, M., editors, *Proceedings of the Ninth International Workshop on Declarative Agent Languages and Technologies (DALT 2011)*, pages 79–94, Taipei, Taiwan.

[63] Vikhorev, K., Alechina, N., and Logan, B. (2011b). Agent programming with priorities and deadlines. In Turner, K., Yolum, P., Sonenberg, L., and Stone, P., editors, *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, pages 397–404, Taipei, Taiwan.

[64] Wagner, T., Garvey, A., and Lesser, V. (1998). Criteria-directed heuristic task scheduling. *International Journal of Approximate Reasoning*, 19:91–118.

[65] Wang, Y. and Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007*, pages 175–182, Honolulu, Hawaii, USA. IEEE.

[66] Waters, M., Padgham, L., and Sardina, S. (2014). Evaluating coverage based intention selection. In Lomuscio, A., Scerri, P., Bazzan, A., and Huhns, M., editors, *Proceedings of the 13th International Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2014)*, pages 957–964. IFAAMAS.

[67] Waters, M., Padgham, L., and Sardiña, S. (2015). Improving domain-independent intention selection in BDI systems. *Autonomous Agents and Multi-Agent Systems*, 29(4):683–717.

[68] Winikoff, M. (2005). JACK Intelligent Agents: An Industrial Strength Platform. In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer.

[69] Winikoff, M. and Cranefield, S. (2014). On the Testability of BDI Agent Systems. *Journal of Artificial Intelligence Research*, 51:71–131.

[70] Winikoff, M., Padgham, L., Harland, J., and Thangarajah, J. (2002). Declarative & Procedural Goals in Intelligent Agent Systems. In Fensel, D., Giunchiglia, F., McGuinness, D. L., and Williams, M., editors, *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 470–481, Toulouse, France. Morgan Kaufmann.

[71] Wobcke, W. (2007). Reasoning about BDI Agents from a Programming Languages Perspective. In *AAAI Spring Symposium: Intentions in Intelligent Systems*, pages 48–, Stanford,California, USA. AAAI.

[72] Wooldridge, M. and Jennings, N. R. (1994). Agent Theories, Architectures, and Languages: A Survey. In *Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Proceedings*, pages 1–39, Amsterdam, The Netherlands. Springer.

[73] Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10:115–152.

[74] Wooldridge, M. J. (2009). *An Introduction to MultiAgent Systems (2. ed.)*. Wiley.

[75] Yao, Y., de Silva, L., and Logan, B. (2016a). Reasoning about the executability of goal-plan trees. In *Proceedings of the 4th International Workshop on Engineering Multi-Agent Systems (EMAS 2016)*, pages 181–196, Singapore.

[76] Yao, Y. and Logan, B. (2016). Action-level intention selection for BDI agents. In Thangarajah, J., Tuyls, K., Jonker, C., and Marsella, S., editors, *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, pages 1227–1236, Singapore. IFAAMAS, IFAAMAS.

[77] Yao, Y., Logan, B., and Thangarajah, J. (2016b). Robust execution of BDI agent programs by exploiting synergies between intentions. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, pages 2558–2565, Phoenix, USA. AAAI, AAAI Press.

Appendix A

GenGPT

GenGPT is a package to generate synthetic goal-plan trees. It can be used in a standalone fashion (as a bundled Java application) to generate a set of goal-plan trees (in XML format) as input to another program. Alternatively, the source code provided can be integrated directly into another program.

The application is packaged as a jar file, and is invoked from the command-line as

```
java -jar GenGPT.jar <args>
```

The arguments required to generate the set of goal-plan trees are: the depth of the goal-plan tree $\#\delta$, the number of subgoals in each non-leaf plan (leaf plans contain only action nodes) $\#\gamma$, the number of plans to achieve a goal $\#\pi$, the number of actions in each plan $\#\alpha$, the maximum p-effect size S, the number of environment variables that may appear in the tree as pre- and postconditions $\#\nu$, the number of goal-plan trees $\#T$ and the output file path *Path* to which the forest of goal-plan trees is saved. The flag for each argument,

their default values and the constraints on each argument are shown in Table A.1.

Table A.1 Arguments required to generate the goal-plan trees

Symbol	Flag	Default	Constraints	Description
$\#\delta$	-d	3	$\#\delta \geq 1$	depth of the tree
$\#\gamma$	-g	1	$\#\gamma \geq 1$	number of subgoals in each plan
$\#\pi$	-p	2	$\#\pi \geq 1$	number of plans to achieve a goal
$\#\alpha$	-a	1	$\#\alpha \geq 1$	number of actions in each plan
$\#S$	-s	0	$S \geq 1$	maximum p-effect size
$\#\nu$	-v	50	$\#\nu \geq 1$	number of environment variables
$\#T$	-t	1	$\#T \geq 1$	number of goal-plan trees
Path	-f	gpt.xml		output file path

The arguments $\#\delta$, $\#\gamma$, $\#\pi$ and $\#\alpha$ together determine the shape of the goal-plan trees in the forest.[1] $\#S$ represents the maximum p-effect size in the goal-plan tree, and $\#\nu$ represents the number of variables that may appear in each goal-plan tree (as the pre- and post-conditions of actions and plans). By varying the value of $\#\nu$ we can vary the likelihood of actions and plans in different goal-plan trees having the same pre- and postconditions, and hence the probability of both conflicts and synergies between goal-plan trees.

The set of goal-plan trees is saved in XML format. The XML file contains a forest(set) of goal-plan trees. Each goal γ has a set of plans $\pi_1, \ldots, \pi_n$ to achieve it. Each plan π consists of a sequence of actions and subgoals, $\pi = \alpha_1; \ldots; \alpha_m$, where each α_i is either an action or a subgoal. We model the environment as a set of propositional variables, and define pre- and postconditions as a set of literals. Each literal is denoted by an $(index, boolean)$ pair in the XML file, where *index* is an identifier of the propositions and *boolean* is either true or false.

[1] It would be more flexible to allow specification of the maximum and minimum number of actions and subgoals in each plan; we assume a fixed number of actions and subgoals for simplicity.

The GenGPT application iteratively generate goals, plans, and actions from the top-level goal to the leaf nodes in the tree. It first generates the top-level goal, and then generates the plans to achieve the top-level goal, and then generates a sequence of actions and subgoals for each of these plans and so on. For each goal, $\#\pi$ plans are generated. The precondition of each plan are established by the postcondition of a previous step in the goal-plan tree (i.e., p-effects). If there is no such p-effect (i.e., the plans to achieve the top-level goal), a proposition i is randomly selected from the set of environment variables, where i is an index. The odd numbered plans to achieve the goal have the literal $(i, True)$ as their precondition, and the even numbered plans have $(i, False)$ as their precondition. This ensures that if $\#\pi > 1$, there is at least one plan applicable to achieve each (sub)goal.

Each plan consists of an ordered sequence of actions and subgoals. The actions in the plan are generated before the subgoals. The first action in the plan has the plan's precondition as its own precondition. The remaining actions either have the plan precondition as their own preconditions or their preconditions are established by previous actions in the plan. The postcondition of an action is selected randomly from the set of environment variables. When all actions in a plan have been generated, and the plan is not a leaf plan (i.e., the plan is not in depth $\#\delta$), we also generate $\#\gamma$ subgoals in this plan by using the same approach mentioned above.

We keep creating actions, plans and (sub)goals until we reach the depth of the tree. The generation process will stop after we generate the last leaf action, and the whole goal-plan tree is translated and saved in an XML file. The synthetic goal-plan trees generated by GenGPT have the following properties:

- Each plan is well formed, i.e., if the precondition of a plan holds in the environment, then this plan can be successfully executed (Assuming there is no environmental changes).

- Taken individually, each goal-plan tree is executable, i.e., there is at least one way to achieve the top-level goal in all (static) environment.

An example goal-plan tree generated by GenGPT with parameters $\#\delta = 2$, $\#\gamma = 1$, $\#\pi = 2$, $\#\alpha = 3$, § = 5, $\#\nu = 100$, $\#T = 1$, $Path$ = example-gpt.xml is shown in Figure A.1), and its corresponding XML representation is shown in Figure A.2. In Figure A.1, goals are represented using red ovals, plans are blue rectangles and actions are green triangles. For simplicity, we didn't provide the name , pre- and postcondition of actions and plans.

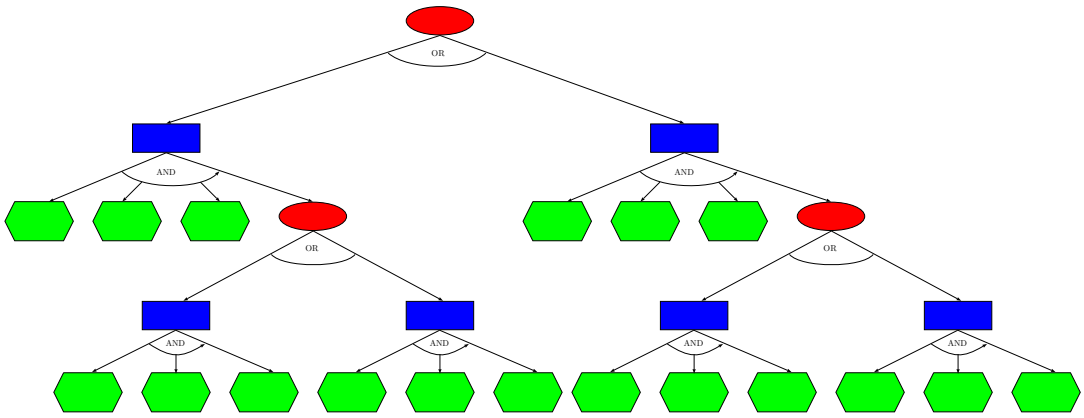


Fig. A.1 An example of synthetic goal-plan tree.

```
<Forest>
  <Goal name="T0-G0">
    <Plan name="T0-P0" precondition="(57,false);">
      <Action name="T0-A0" precondition="(57,false);" postcondition="(45,false);"/>
      <Action name="T0-A1" precondition="(57,false);" postcondition="(18,true);"/>
      <Action name="T0-A2" precondition="(18,true);" postcondition="(61,false);"/>
      <Goal name="T0-G1">
        <Plan name="T0-P1" precondition="(18,true);">
          <Action name="T0-A3" precondition="(18,true);" postcondition="(28,false);"/>
          <Action name="T0-A4" precondition="(18,true);" postcondition="(29,false);"/>
          <Action name="T0-A5" precondition="(29,false);" postcondition="(39,true);"/>
        </Plan>
        <Plan name="T0-P2" precondition="(61,false);">
          <Action name="T0-A6" precondition="(61,false);" postcondition="(82,false);"/>
          <Action name="T0-A7" precondition="(82,false);" postcondition="(83,false);"/>
          <Action name="T0-A8" precondition="(82,false);" postcondition="(24,true);"/>
        </Plan>
      </Goal>
    </Plan>
    <Plan name="T0-P3" precondition="(57,true);">
      <Action name="T0-A9" precondition="(57,true);" postcondition="(87,true);"/>
      <Action name="T0-A10" precondition="(87,true);" postcondition="(58,true);"/>
      <Action name="T0-A11" precondition="(57,true);" postcondition="(22,true);"/>
      <Goal name="T0-G2">
        <Plan name="T0-P4" precondition="(22,true);">
          <Action name="T0-A12" precondition="(22,true);" postcondition="(14,false);"/>
          <Action name="T0-A13" precondition="(22,true);" postcondition="(50,true);"/>
          <Action name="T0-A14" precondition="(50,true);" postcondition="(85,false);"/>
        </Plan>
        <Plan name="T0-P5" precondition="(58,true);">
          <Action name="T0-A15" precondition="(58,true);" postcondition="(31,true);"/>
          <Action name="T0-A16" precondition="(58,true);" postcondition="(29,true);"/>
          <Action name="T0-A17" precondition="(58,true);" postcondition="(59,false);"/>
        </Plan>
      </Goal>
    </Plan>
  </Goal>
</Forest>
```

Fig. A.2 XML representation of goal-plan trees