This electronic thesis or dissertation has been downloaded from Explore Bristol Research, http://research-information.bristol.ac.uk

Author:
Xu, Mengwei

Title:
Extending BDI Agents with Robust Program Execution, Adaptive Plan Library, and Efficient Intention Progression

General rights
Access to the thesis is subject to the Creative Commons Attribution - NonCommercial-No Derivatives 4.0 International Public License. A copy of this may be found at https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode This license sets out your rights and the restrictions that apply to your access to the thesis so it is important you read this before proceeding.

Take down policy
Some pages of this thesis may have been removed for copyright restrictions prior to having it been deposited in Explore Bristol Research. However, if you have discovered material within the thesis that you consider to be unlawful e.g. breaches of copyright (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please contact collections-metadata@bristol.ac.uk and include the following information in your message:

- Your contact details
- Bibliographic details for the item, including a URL
- An outline nature of the complaint

Your claim will be investigated and, where appropriate, the item in question will be removed from public view as soon as possible.

Extending BDI Agents with Robust Program Execution, Adaptive Plan Library, and Efficient Intention Progression

Mengwei Xu, B.Sc

Department of Computer Science

University of Bristol

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering.

Feb 2020

Word Count: 65268

ABSTRACT

The Belief-Desire-Intention (BDI) architecture, where agents are modelled based on their (B)eliefs, (D)esires, and (I)ntentions, provides a practical approach to developing intelligent agent systems. These agents operates by context sensitive expansion of plans, thus allowing fast reasoning cycle. However, the practical capability of BDI agents can still remain limited due to the lack of abilities to handling execution failure, adapting to the environment, and pursuing multiple intentions correctly and efficiently. In this thesis, we will address these issues in the following ways.

Firstly, we introduce a novel operational semantics for incorporating First-principles Planning (FPP) to recover execution failure by generating new plans when no alternative pre-defined plan exists or worked. Such a semantics provides a detailed specification of the appropriate operational behaviour when FPP is pursued, succeeded or failed, suspended, or resumed in BDI. Therefore, the robustness of a BDI agent can be substantially improved when facing unforeseen situations.

Secondly, we advance the state-of-the-art in BDI agent systems by proposing a plan library evolution architecture with mechanisms to incorporate new plans (plan expansion) and drop old/unsuitable plan (plan contraction) to adapt to changes in a realistic environment. Such a proposal follows a principle approach to define plan library expansion and contraction operators, motivated by postulates that clearly highlight the underlying assumptions, and quantified by decision-support measure information. Therefore, the adaptivity of BDI can be improved for a fast-changing environment.

Thirdly, we provide a theoretical framework where FPP is employed to manage the intention interleaving in an automated fashion. Such a framework employs FPP to plan ahead to not only avoid the potential negative intention interactions, but also capitalise on their positive interactions (i.e. overlapping sub-intentions). As a benefit, the achievability of intentions (i.e. a correct execution) is guaranteed, and the overall cost of intentions execution is reduced (i.e. an efficient execution).

DEDICATION AND ACKNOWLEDGEMENTS

First of all, I want to thank my principal supervisor, Prof. Weiru Liu, for taking me as her student, and appreciating and thinking highly of my mathematical background. During my PhD, I thank her inspirational role to me. Her leadership, excellent management skills, strong work ethic, and a kind heart set a perfect example for me to follow, which indeed pushed me to nurture similar characters, e.g. my infant early leadership.

I want to offer my sincere thanks to my second supervisor Dr. Kim Bauters. When I think of him, the best word to describe him to me is the word of "mentor". I still remember so vividly when he helped (or should say "urge") me to strike a balance between work and life in the very early stage of my PhD journey. Such advice is significant to me and completely changed my approach to my PhD program. One thing I am overwhelmingly proud of is not the research outputs I produced. Instead, the true pride lies in that I produced these research outputs while having all my weekend off to relax and always spending Easter/Summer/Christmas holiday with family. This would not happen if without Kim's advice among many others. To some extent, my PhD project and being me a researcher would not have been possible without the guidance and input of my mentor Kim.

I also want to thank Dr. Kevin McAreavey for his help as my third supervisor. Some of part of my work would not be this good if without the valuable comments and discussions from Kevin.

When I look beyond the work environment, I would not be who I am now without the help from many senior people whom I respect from the bottom of my heart. To name a few, I thank Jim Crookes, Dr. Ariel Blanco, and Lizzy Page for their wisdom and for lifting me up when I felt low.

There is no doubt that I would be nothing without my parents Guangcai Xu and Ping Zhang. I thank them for their unconditional and continuous love. I know that they are always there for me whenever I need them. Similar thanks go out to my parents-in-law Jack McClintock and Heather McClintock, who will not hesitate to help my wife and me if they can.

Most important of all is my wife, Laura McClintock-Xu. Just as we vowed at our wedding, for better, for worse, for richer, for poorer, in sickness and in health, we will stand by each other side until death do us part. Finally, I want to thank this particularly important time for Laura and me as a couple when we are expecting our first baby. I am looking forward to hugging my baby soon. And I know our baby will certainly be spoiled by its two sets of grandparents.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE: ....

Contents

Abstract **i**

Contents **vii**

List of Tables **ix**

List of Figures **xi**

List of Algorithms **xv**

List of Acronyms **xv**

List of Symbols **xvii**

List of Pulications **xxv**

1 Introduction **1**

1.1 Belief-Desire-Intention Agents 2
1.2 AgentSpeak and CAN 4
1.3 Planning 8
1.4 Thesis Outline 11
1.5 Overview 12

2 Preliminaries **13**

2.1 General Notation 13
2.2 Logic Programming 14
2.3 Conceptual Agent Notation (CAN) 16
2.4 Planning 30
2.5 Graph Theory: Fundamentals 36

3 Literature Review **39**

3.1 Planning to Generate New BDI Plans 39
3.2 Applying Lookahead Planning to Existing BDI Plans 45

3.3 Plan Selection . . . 48
3.4 Intention Selection . . . 53
3.5 Summary . . . 59

4 Recovering Agent Program Failure via Planning **63**
4.1 Introduction . . . 63
4.2 Execution Failure in BDI . . . 65
4.3 Declarative Intentions in BDI . . . 67
4.4 First-Principles Planning in BDI . . . 71
4.5 Formal Relationship between FPP and BDI . . . 74
4.6 Feasibility Study . . . 76
4.7 Conclusions . . . 78

5 Equipping BDI Agents with Adaptive Plan Library **81**
5.1 Introduction . . . 81
5.2 Plan Library Analysis . . . 84
5.3 Plan Library Expansion . . . 88
5.4 Plan Library Contraction . . . 89
5.5 Conclusion . . . 95

6 Efficient Intention Progression via Planning **99**
6.1 Introduction . . . 99
6.2 Intention Interleaving Planning Framework . . . 102
6.3 Intention Interleaving Planning Implementation . . . 113
6.4 Intention Interleaving Planning Evaluation . . . 115
6.5 Conclusion . . . 116

7 Conclusion **117**
7.1 Planning in BDI . . . 117
7.2 Managing Multiple Intentions . . . 119
7.3 Discussion and Future Work . . . 120

Bibliography **123**

List of Tables

Table **Page**

3.1 Summary of the works of incorporating planning in BDI discussed in this chapter . . 60
3.2 Summary of the works of plan selection in BDI discussed in this chapter 61
3.3 Summary of the works of intention selection in BDI discussed in this chapter 62

5.1 Criterion Values and Normalised Criterion Values 93

6.1 STRIPS Progression Links 111
6.2 Effectiveness Analysis of Approach 115

List of Figures

Figure **Page**

1.1 The Reason Cycle of An AgentSpeak Agent 5

2.1 Clean Robot in a Traffic-world 25
2.2 BDI Agent Belief Design in the Robot Cleaning Scenario 26
2.3 BDI Agent Plan Library Design in the Robot Cleaning Scenario 27
2.4 Semantic Evolution Flow of the Program `+location(waste,b)`. 28
2.5 Robotic Cleaner in a Two-grid World 32
2.6 Examples of Hierarchical Plan Library Structure 36

4.1 Layout of a Smart House with a Domestic Robot 64
4.2 Diagrammatic Evolutions of the Rule (1) $R_{\mathtt{cov}}$, (2) $R_{\mathtt{pre}}$, and (3) $M_{\mathtt{pr}}$ 69
4.3 Diagrammatic Evolution of the Rule (1) $R_{\mathtt{fail}}$, (2) $R_{\mathtt{emp}}$, and (3) $R_{\mathtt{mot}}$ 71
4.4 BDI Agent in Domestic Cleaning Scenario 76

5.1 A Set of Uncertain and Certain Clauses 94
5.2 Arguments and accrued structures 96

6.1 AND/OR Graphs for Goal-plan Trees. 104
6.2 A Goal-plan Tree with Two Relevant Plans. 105

List of Algorithms

Algorithm **Page**

1 Computation for Contraction Operator ∇^{abm} 92

2 Intention Interleaving Replanning 113

List of Acronyms

ADL Action Description Language

AI Artificial Intelligence

AOP Agent-oriented Programming

BDI Belief-Desire-Intention

CAN Conceptual Agent Notation

DTC Design-To-Criteria

FPP First-principles Planning

HTN Hierarchical Task Networks

IPC International Planning Competition

MCTS Monte-Carlo Tree Search

MDPs Markov Decision Processes

mgu most general unifier

OOP Object-oriented Programming

PDDL Planning Domain Definition Language

POMDPs Partially Observable Markov Decision Processes

PRS Procedural Reasoning System

RDDL Relational Dynamic Diagram Language

STRIPS Stanford Research Institute Problem Solver

3APL Artificial Autonomous Agents Programming Language

2APL A Practical Agent Programming Language

List of Symbols

General Notation

Symbol	Description
$\preceq$	a total order, page 13
2^U	the power set of U, page 13
$\cap$	set intersection, page 13
$\cup$	set union, page 13
$\mathbb{N}$	the set of natural numbers, page 14
$\mathbb{R}$	the set of real numbers, page 14
$\mathbb{R}_{\geq 0}$	the set of non-negative real numbers, page 14
$\mathbf{u}$	a vector such that $\mathbf{u} = (u_1, \ldots, u_n)$, page 14
$\lvert U \rvert$	the cardinality of the set U, page 13
$\prec$	a strict order, page 13
$\setminus$	set difference, page 13
$\subset$	strict set inclusion, page 13
$\subseteq$	set inclusion, page 13
U	a set symbol, page 13
u	an object of a set, page 13

Logic Programming

Symbol	Description
$\bot$	the falsity value *false*, page 15
ω	a Herbrand interpretation such that $\{p \in H \mid \omega(p) = \top\}$ where p is an atom implicitly grounded and H is the Herbrand base, page 15
θ	a substitution such that $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$, page 15

$\top$	the truth value *true*, page 15
c	a constant and a term, page 14
$h \leftarrow b_1 \wedge \ldots \wedge b_n$	a (normal) clause, page 14
p	a predicate symbol such that $p(t_1,\ldots,t_n)$ is an atom, page 14
$t_1 = t_2$	an equation, page 14
V	a variable and a term, page 14

BDI

Λ	the action library, page 16
$\mathcal{B}$	the belief base, page 16
$+?b(\mathbf{t})$	a test goal, page 17
$+b(\mathbf{t})$	a belief addition, page 17
$-b(\mathbf{t})$	a belief deletion, page 17
$?\phi$	a test for ϕ entailment, page 17
Γ	the intention base, page 24
$\mathcal{A}$	the sequence of actions executed so far by an agent, page 19
$goal(\varphi_s, P, \varphi_f)$	a (normal) declarative goal, page 18
$P(\mathbf{t_3})$	the plan-body, page 16
$P_1 \rhd P_2$	execute P_2 only on failure of P_1, page 18
$P_1;P_2$	P_1 followed by P_2, page 17
$P_1 \parallel P_2$	interleaved concurrency of P_1 and P_2, page 17
$\models$	the entailment operator, page 16
$\xrightarrow{*}$	the transitive closure of a transition, page 19
$\phi^{-}(\mathbf{x})$	a delete set of belief atoms of an action, page 18
Π	the plan library, page 16
$\psi(\mathbf{x})$	the precondition of an action, page 18

$\varphi(\mathbf{t_2})$ the context condition of a BDI plan, page 16

φ_f the failure condition in a declarative goal, page 18

φ_s the success condition in a declarative goal, page 18

$a(\mathbf{x})$ an action symbol, page 18

act a primitive action, page 17

b a belief predicate such that $b(\mathbf{t})$ is a belief atom, page 16

$C \to C'$ a configuration transition, page 19

$e(\mathbf{s})$ an event goal, page 17

$e(\mathbf{t_1})$ the triggering event of a BDI plan, page 16

$e(\mathbf{t_1}) : \varphi(\mathbf{t_2}) \leftarrow P(\mathbf{t_3})$ a BDI plan, page 16

nil an empty program, page 18

$\phi^+(\mathbf{x})$ an add set of belief atoms of an action, page 18

C a configuration of a BDI agent, page 19

Planning

φ_g a goal formula in FPP, page 31

$A(s)$ the set of actions in A that are applicable in each state $s \in S$, page 30

$add(o)$ the add-list of an operator, page 31

$b_0(s_0)$ the probability distribution of state s_0, page 34

$del(o)$ the delete-list of an operator, page 31

$f(a,s)$ the deterministic transition function, page 30

$f(s,o)$ the effects of applying an operator o to a state s, page 32

O a set of operators such that $o \in O$, page 31

Ot a finite set of observation token such that $ot \in Ot$, page 34

$P_a(ot \mid s)$ the probability of receiving observation token ot in state s when the last applied action was a, page 34

$P_a(s' \mid s)$ the transition probability for s' being the next state after doing the action $a \in A(s)$ in the state s, page 34

$post(o)$ the post-effects of an operator o such that $post(o) = add(o) \cup \{\neg l \mid l \in del(o)\}$, page 31

$pre(o)$ the precondition of an operator, page 31

$Res(s_0, \langle o_1; \ldots; o_n \rangle)$ the (res)ult of applying the sequence $\langle o_1; \ldots; o_n \rangle$ to s_0n, page 32

S a finite and discrete set of states, page 30

s_0 the initial state, page 30

S_G the non-empty set of goal states, page 30

$\langle s_0, \varphi_g, O \rangle$ an FPP problem, page 31

Graph Theory

$\bar{n}$ a root node in a directed graph, page 37

$child(n)$ the child node of n, page 36

E the set of directed edges such that $E \subseteq N \times N$, page 36

E' the set of multiedges such that $E' \subseteq N \times L \times N$, page 36

$E_\vee$ a set of OR-edges such that $E_\vee \subseteq N_\vee \times L_\vee \times N_\wedge$, page 37

$E_\wedge$ a set of AND-edges such that $E_\wedge \subseteq N_\wedge \times L_\wedge \times N_\vee$, page 37

L the set of labels, page 36

$L_\vee$ a set of OR-labels, page 37

$L_\wedge$ a set of AND-label, page 37

N a set of nodes, page 36

$N_\vee$ a set of OR-nodes, page 37

$N_\wedge$ a set of AND-nodes, page 37

Chapter 4

Γ_{de} the declarative intention set, page 67

$\mathcal{M}$ a motivational library, page 68

$goal(\varphi_s, \varphi_f)$ a pure declarative goal, page 68

$sol^{off}(\mathcal{B},\varphi_s,\Lambda)$ the offline solution of the FPP problem $\langle\mathcal{B},\varphi_s,\Lambda\rangle$, page 72

$\psi \rightsquigarrow P$ a rule in the motivational library, page 68

Γ_{pr} the procedural intention set, page 67

$sol^{on}(\mathcal{B},\varphi_s,\Lambda)$ the online solution of the FPP problem $\langle\mathcal{B},\varphi_s,\Lambda\rangle$, page 72

Chapter 5

$>_{\mathcal{C}}$ a strict order of the element of $\mathcal{C}$, page 90

$\delta(\Pi,t_1,t_n)$ an execution frequence of a plan library Π from time point t_1 to t_n, page 86

$\delta(P,t_1,t_n)$ an execution frequence of a plan P from time point t_1 to t_n, page 84

η the success rate tolerance threshold, page 91

λ a set of uncertain clauses, page 91

$\langle X,\mathcal{K},\mathcal{R}\rangle$ a problem of multi-criteria argumentation-based decision, page 90

$\mathcal{C}$ a set of non-cyclic (i.e. linear) criteria, page 90

$\mathcal{F}(\Pi)$ a degree of the functionality of a plan library Π, page 87

$\mathcal{K}$ an epistemic knowledge, page 90

$\mathcal{O}(\{P,P_1,\cdots,P_n\})$ the overlap of P and $\{P_1,\cdots,P_n\}$, page 85

$\mathcal{P}$ a set of BDI plans, page 84

$\mathcal{R}$ a set of decision rules in multi-criteria decision, page 91

$\mathcal{S}$ a status function of BDI plans, page 84

$\mathcal{T}$ a set of time points, page 84

ACC a user-specified aggregation function, page 91

$\nabla(\Pi)$ the contraction of Π by ∇, page 89

$\Phi(P,t_1,t_n)$ a success rate for a plan P from time point t_1 to t_n, page 84

$\succeq_{activeness}$ a binary relation such that $\Pi \succeq_{activeness} \Pi'$ iff $\delta(\Pi,t_1,t_n) \geq \delta(\Pi',t_1,t_n)$, page 87

$\succeq_{functionality}$ a binary relation such that $\Pi \succeq_{functionality} \Pi'$ iff $\mathcal{F}(\Pi) \geq \mathcal{F}(\Pi')$, page 87

$\succeq_{robustness}$ a binary relation such that $\Pi \succeq_{robustness} \Pi'$ iff $\nexists P \in \Pi\ s.t.\ P \in \Pi'$, $\Upsilon_{\Pi}(P) \leq \Upsilon_{\Pi'}(P)$, and $\varsigma_{\Pi}(P) \leq \varsigma_{\Pi'}(P)$, page 87

$\succeq_{success}$ a binary relation such that $\Pi \succeq_{success} \Pi'$ iff $\Phi(\Pi, t_1, t_n) \geq \Phi(\Pi', t_1, t_n)$, page 87

$\Upsilon_{\mathcal{P}}(P)$ a relevancy measure of a plan P in the set of plans $\mathcal{P}$, page 85

$\Phi(\Pi, t_1, t_n)$ a successful rate of a plan library from time point t_1 to t_n, page 87

$\Pi \circ P$ the expansion of Π by P, page 88

ϖ a set of certain clauses, page 91

$\varsigma_{\mathcal{P}}(P)$ a degree of replaceability for plan P in the set of plans $\mathcal{P}$, page 86

C_1 the overall execution frequency of a plan P from initial time point t_0 to current time point $t_{current}$, namely $C_1 = \delta(P, t_0, t_{current})$, page 91

C_2 the latest execution frequency of P from a chosen recent time point t' to $t_{current}$, namely $C_2 = \delta(P, t', t_{current})$, page 91

C_3 the overall success rate of a plan, page 91

C_4 the latest success rate, page 91

e^P a set of relevant plans $\{P_1, \cdots, P_n\}$ for achieving an event e, page 85

$P \rhd_{mr} S$ plan P can be minimally replaced by a set of plans S, page 86

$P \rhd_r S$ plan P can be replaced by a set of plans S, page 86

$post(P)$ the post-effects of a plan P, page 85

$sol(\langle X, \mathcal{K}, \mathcal{R} \rangle)$ the solution to problem of multi-criteria argumentation-based decision, page 91

X the set of all possible candidates, page 90

Chapter 6

$add(\alpha^o)$ the add-list of an overlap progression link, page 111

$(n \rightarrow n')$ a primitive progression link, page 108

$\mathcal{G}$ a set of (sub)goals, page 103

$add(\alpha^p)$ the add-list of a primitive progression link α^p, page 111

$del(\alpha^o)$ the delete-list of an overlap progression link α^o, page 111

$del(\alpha^p)$ the delete-list of a primitive progression link α^p, page 111

$pre(\alpha^o)$ the precondition of an overlap progression link α^o, page 111

$pre(\alpha^p)$	the precondition of a primitive progression link α^p, page 111
$size(\alpha^o)$	the size of an overlap progression link, page 110
$size(\alpha^p)$	the size of a primitive progression link, page 110
$\nu(G)$	the terminal node set of a goal node G, page 107
Ω	An FPP problem of intention interleaving, page 112
$\omega(G)$	the set of all execution traces of a goal G, i.e. $\tau(G) \in \omega(G)$, page 104
ρ	a sequence of progression link, page 111
Σ	a finite set of (propositional) atoms, page 110
σ	an execution trace of a set of intentions $\{T_1, \ldots, T_m\}$, page 105
$\sigma[i]$	the ith element of σ, page 105
σ^m	the merged execution trace of σ, page 107
$\tau(n)^\infty$	the last element of execution trace $\tau(n)$, page 107
$\tau(T)$	an execution trace of a goal-plan tree T such that $\tau(T) = \tau(T(\bar{n}))$, page 104
$n^{P,j,T}$	the jth member of the body of the plan P in T, page 107
n^T	a plan node $n \in \Pi$ in an intention T, page 107
N_{id}	a set of node indexes of a set of intentions I, page 110
O	a set of progression links, page 110
T	a goal-plan tree for an intention in a BDI agent to achieve a top-level goal $G \in \mathcal{G}$ such that $T = (N_\vee \cup N_\wedge, L_\vee \cup L_\wedge, E_\vee \cup E_\wedge, \bar{n})$, page 103
$T[\bar{n}]$	the top-level goal of T, page 103
$T[N]$	both OR-nodes and AND-nodes of a goal-plan tree T, page 103
z_0	an initial node set of a set of intentions $I = \{T_1, \ldots, T_m\}$ such that $z_0 = \{T_1(\bar{n}), \ldots, T_m(\bar{n})\}$, page 107
z_g	a terminal node set of a set of intentions $I = \{T_1, \ldots, T_m\}$ such that $z_g = \{tn_1, \ldots, tn_m\}$, page 107
$J(idx)$	the actual node of the index idx, page 107
O^o	a set of overlap progression links, page 110
O^p	a set of primitive progression links, page 110

List of Pulications

The publications published in the duration of my PhD are listed below:

2019 **Intention Interleaving Via Classical Replanning** (Mengwei Xu, Kevin McAreavey, Kim Bauters, Weiru Liu), *In Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 85-92, 2019.

2018 **A Framework for Plan Library Evolution in BDI Agent Systems** (Mengwei Xu, Kim Bauters, Kevin McAreavey, Weiru Liu), *In Proceedings of the 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 414-421, 2018.

2018 **A Formal Approach of Embedding First-principles planning in BDI Agent Systems** (Mengwei Xu, Kim Bauters, Kevin McAreavey, Weiru Liu), *In Proceedings of the 12th International Conference on Scalable Uncertainty Management (SUM)*, pp.333–347, 2018.

Chapter 1

Introduction

Ever since the invention of the computers, there has been a vast amount of approaches to programming developed over time spanning from early mechanical computers to modern tools for software development. In early approaches to programming, the programming languages usually directly represent the instructions machine codes, thus conventionally being identified as the so-called low-level languages. Later on, the high-level programming languages are introduced to enable using vocabularies related to actual programming problems because of the development of compiler theory, thus making program development simpler and more understandable. One of the popular representatives is the object-oriented programming languages based on the concept of objects. In these languages (e.g. Java), the instructions are encapsulated into objects (e.g. providing methods with variables) and the computer programs are designed by having objects as basic modules interacting with one another. Nowadays, given the growing presence of intelligent agents (e.g. robots), there has been a new programming paradigm, namely the Agent-oriented Programming (AOP) languages, which designs the computational system from a mentalistic view consisting of state of basic units such as beliefs, obligations, and capabilities.

Intuitively, the construction of the computer programs in AOP languages is to specify a (software) agent as if it has “mental states". In work of [Sho93], which first articulated the concept of AOP, the agent is considered to be an entity in which its state is composed of human-like mental components, e.g. beliefs and commitments. Naturally, these mental components originate from their common sense counterparts which are part of our everyday linguistic ability. To correctly correspond to the common sense use of the mental terms, however, there is often a precise theory regarding the particular mental category. Expectedly, the variance of agent systems depends on the types of mental components that an agent system is viewed as possessing. By ascribing mental qualities to machines with associated precise theories, AOP languages can thus offer the programmers not only a familiar, non-technical way to talk about complex systems,

but also a formal vehicle to build (software) agents which is more likely to exhibit the degree of "intelligence", e.g. flexible (reactive, proactive, social) autonomous behaviours. This is particularly appealing and important nowadays when robotics and autonomous systems have already been identified as one of the eight great technologies [Wil13] with the potential to revolutionise our economy and society.

Belief-Desire-Intention (BDI) agent paradigm, which will be used throughout this thesis, is one of the predominant approaches for AOP languages to designing intelligent agent systems via the mental component of Beliefs, Desires, and Intentions. In particular, it has been claimed that the employment of BDI agent technology in complex business settings can improve overall project productivity by an average $350-500\%$ according to an industry study [BHG06]. In this thesis, we develop a number of extensions to BDI agents to advancing its capabilities, of which we outline the underlying motivations and ideas in Section 1.1 and Section 1.2. The extensions of BDI agents that we present mostly rely on the advanced planning techniques (the details of which are discussed in Section 1.3). As a consequence, our planning-centric BDI agents can not only cope with potential failure during execution to remain robust, but also think ahead to ensure correct and efficient execution when pursuing multiple tasks. In particular, we also investigate how a BDI agent can adopt the new knowledge, e.g. from the external planning tools, and discard its obsolete and erroneous knowledge to adapt to a changing environment. Before we can explore these topics and present our contributions, we first describe the foundation of BDI agent framework and one classical line of BDI languages, namely AgentSpeak and Conceptual Agent Notation (CAN), in Section 1.1 and Section 1.2, respectively.

1.1 Belief-Desire-Intention Agents

The BDI agency model [Bra87, BIP88], as the most dominant and mature outputs of the AOP community, specifically targets the modelling of intelligent agent based on three mental categories, namely (B)eliefs, (D)esires, and (I)ntentions. Intuitively, the beliefs of the agent represent knowledge that the agent has about the environment in which it is situated. Since these beliefs are from the perspective of the agent, they can be incomplete or even incorrect (e.g. mistaken that it is rainy today). The desires of the agent, meanwhile, are all the possible state of affairs that the agent might want to bring about in an ideal world. In principle, the agent can have a set of inconsistent desires (e.g. the desires of humans are often inconsistent). However, the set of intentions must be consistent where intentions are those desires that an agent has committed to achieving. In fact, it would be irrational for an agent to entertain two options that are inconsistent with each other. To illustrate, the agent can desire as it wishes, e.g. to be in both London and Dublin tomorrow. Still, the physical law forbids it from actually being in both these two places tomorrow. In other words, it cannot actually intend to be both in London and Dublin tomorrow because these two intentions are inconsistent. Finally, even if all desires of an agent are consistent, the

agent generally will not be able to commit to all its desires in a realistic environment due to the bounded resources [Sim72]. Therefore, an agent typically has to fix upon some subset of its desires and commits its resources to achieve them only.

To build an intelligent agent, the BDI paradigm is particularly appealing because it provides a sound philosophical foundation, clear logic and semantics, and a collection of mature agent-oriented programming languages and platforms. We now succinctly explain these three key components of BDI model one by one. Firstly, the philosophical root of BDI model is in the philosophical tradition of understanding practical reasoning in humans, deciding moment by moment which action to do next. At its simplest, practical reasoning is the process of figuring out what to do (i.e. reasoning directed towards actions). Formally speaking, the practical reasoning is a matter of weighing conflicting considerations for and against completing options, where the relevant considerations are provided by what the agent desires, values, and cares about, and what the agent believes [Bra90]. In detail, practical reasoning consists of two distinct activities, namely deliberation and mean-end reasoning. Whereas the process of deliberation decides what state of affairs the agent wants to achieve, the means-end reasoning involves deciding how to achieve these states of affairs.

Secondly, the theory in the family of BDI logic has also been rigorously formalised. As such, the different mentalistic concepts (e.g. intentions) and their relationship can be studied in a formal setting. The theory of intention in practical reasoning is first formalised by Cohen and Levesque in [CL90] where a composition concept of intention is given, namely (i) chosen desires, (ii) persistent desires, and (iii) intentions. By construction, chosen desires are consistent. Meanwhile, persistent chosen desires amount to intentions. In contrast to treating intentions as being reducible to beliefs and desires, Rao and Georgeff [RG91] embraces a primitive notion of intention which has equal status with the notions of belief and desire to define different strategies of commitment. This primitive notion of intention allows flexibility to define different strategies of commitment with respect to the intentions of an agent by imposing certain conditions on the persistence of the beliefs, desires, and intentions. For instance, an agent assigned with the blind commitment will maintain an intention until it is believed by the agent that such an intention has been achieved.

Finally, a multitude of BDI languages and software platforms has also been developed along with philosophical and logic research advancement in BDI paradigm. Notably, one of the earliest implementations inspired by the BDI model is the so-called Procedural Reasoning System (PRS) [GL87]. In PRS implementations, the agent beliefs are directly represented in the form of Prolog-like facts [Bra01], while the desires and intention are realised through the use of a plan library (i.e. a collection of plans). In each plan, it consists of a body, which describes the steps to achieve its goal, and an invocation condition that specifies under which situations the agent should consider applying such a plan. From a conceptual standpoint, the PRS agent works simply by choosing plans to respond to active goals given the current beliefs. As a consequence,

unlike those logic formalisations discussed previously (e.g. [RG91]) which tend to be inefficiently computable, the PRS implementations are suitable for building practical BDI agent systems.

1.2 AgentSpeak and CAN

While the PRS framework is a useful system for implementing the BDI model, its practical perspective makes it difficult to investigate its theoretical properties such as soundness and, furthermore, to examine itself in contrast with other aforementioned theoretical works (e.g. [RG91]). To this end, the AgentSpeak language [Rao96] is proposed in a sufficiently simple, uniform language framework. In essence, the AgentSpeak language can be reviewed as an abstraction of PRS in a formal setting. Indeed, AgentSpeak retains the key features of PRS systems but allows the programs of the agent to be constructed in a fashion close to the logic programming (whose details is given in Section 2.2). Therefore, it would be possible to investigate it from a theoretical point of view, for example, by giving it a formal semantics.

To design an AgentSpeak agent, it needs specifying by beliefs, intentions, a set of events, and plan rules. The set of the base beliefs is the specification of the agent, which represents what it believes to be true. Formally, the set of base beliefs is a collection of grounded atoms (e.g. `male(Bob)` encoding that Bob is male), as in traditional logic programming (which is discussed in more details in Section 2.2). The set of plan rules is called to be the plan library of an agent. In AgentSpeak language, the agent does not produce the plan from scratch. Instead, it is equipped with a library of pre-defined plans. These plans are manually constructed, in advance, by the agent programmers. Each plan in AgentSpeak has three following components, namely (i) a triggering event, (ii) a context condition, and (iii) a plan-body. In detail, the triggering event defines what the plan is good for, i.e. the goals that it can achieve. The context condition of an AgentSpeak plan, however, articulates when such a plan is good for. In other words, the context condition of a plan defines what must be true of the environment in order for this plan to be applied. Normally, a plan will only be applied to achieve the event which matches its triggering event when the context condition of such a plan holds in its current set of base beliefs. Finally, the plan-body of a plan in AgentSpeak encodes procedural information on how to respond to its associated event. Such procedural information usually consists of entities such as actions, which are executed directly by the agent, and some internal events (as subgoals) which require further refinement (i.e. selecting plans for such internal events) to execute their corresponding actions.

In a nutshell, an AgentSpeak agent addresses events (i.e. the inputs to the agent systems) by (i) selecting plans from the plan library, (ii) placing it into the intention set, and (iii) selecting intention to progress to address the events. The Figure 1.1 depicts such an event-driven reasoning cycle of an AgentSpeak agent. In detail, given a pending event to deal with, the agent starts with searching through the plan library to find a suitable pre-defined plan at run time. The first step of this searching is to retrieve all plans whose triggering event matches the given pending event.

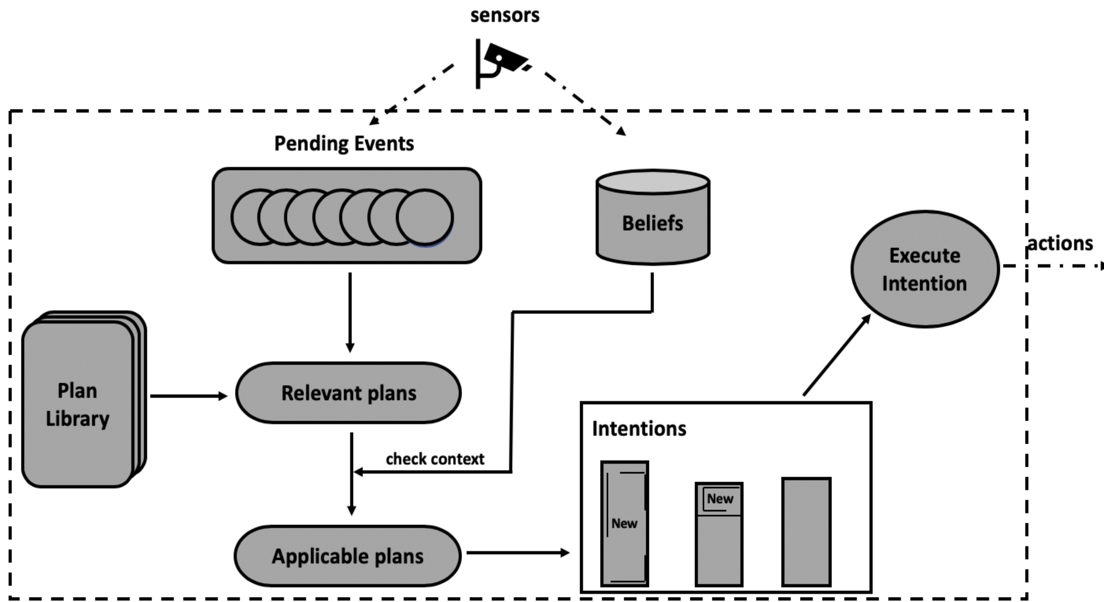

Figure 1.1: The Reason Cycle of An AgentSpeak Agent

These plans are often called relevant plans. Recall that the context condition of a plan tells us whether a plan can be applied at a particular moment in time, given the information the agent currently has. Therefore, the agent next needs to select an applicable plan from previously all retrieved relevant plans. In order to do so, the agent simply needs to check whether the context condition of each relevant plan is believed to be true according to the current base belief. Once an applicable plan exists and is selected, the plan-body of such a plan becomes an intention adopted by the agent. Intention adoption can amount to (i) adding the instantiated plan-body to the current set of intentions when the triggering event is external, (ii) replacing the associated existing intention with the newly refined intention when the event is internal. Finally, given a set of current intentions, the agent selects an intention each time and execute one step of such an intention. This reasoning cycle will carry on until there is no pending event to respond to and all intentions are completed.

While the AgentSpeak language is a useful formal language for BDI agents, it is limited to a procedural interpretation for goals. To illustrate, the goals in AgentSpeak are treated as events which trigger plans. The pursuit of a goal is equivalent to the execution of a set of procedures written in a related plan-body. Despite the practicality of the procedural aspect of goals in dynamic environments, the absence of declarative aspect of goals (i.e. a description of the state sought) renders the ability to reason about goals impossible. For instance, without declarative information of what a goal an agent is trying to achieve, the agent has no mechanism to check, e.g. whether it has been indeed achieved after the successful execution of procedures. To decouple the plan failure (resp. failure) from goal failure (resp. failure), CAN language [WPHT02] is proposed

to introduce a representation of goals which allows for both declarative and procedural aspects to be specified and used. To do so, CAN language includes a new declarative goal construct, namely $goal(\varphi_s,P,\varphi_f)$ where φ_s and φ_f are grounded atoms built from normal connectives, e.g. the logical conjunction $\wedge$, whereas P stands for the plan-body. Such a declarative goal construct succinctly states that the success condition φ_s should be achieved using (procedural) plan-body program P, failing if the failure condition φ_f becomes true. The operational semantics provided by CAN language, which is simplified in [SSP06] later on, nicely captures the sensible behaviours of dropping goals when they are either achieved (i.e. φ_s holds), or become unachievable (i.e. φ_f holds). Although persistence is dependent on the commitment strategy of a given agent (e.g. the blind commitment in [RG91]), the declarative goal construct in CAN does not omit the persistence in BDI theory of practical reasoning [Bra87]. As a matter of fact, to ensure the certain amount of persistence in CAN, if the plan-body program P within $goal(\varphi_s,P,\varphi_f)$ has been completely executed, but the success condition φ_s still does not hold true, then P will be re-executed.

In addition to the unification of declarative and procedural goals, CAN language also proposes another distinguishing feature, namely the automatic failure handling mechanism. To cope with failure, a built-in backtracking failure handling mechanism exists to try other available plans to address a goal if one plan currently selected to achieve the same goal has failed. To illustrate, let two plans be P_1 and P_2 to achieve a goal G and the plan P_1 is currently selected to pursue goal G. To anticipate the potential occurrence of the goal recovery, the CAN agent keeps plan P_2 as a back-up plan while executing plan P_1. Whenever plan P_1 has failed before its completion, the agent can initiate the execution of the back-up plan P_2 to continue accomplishing goal G. In the case of no alternative back-up plan available, the goal will then be deemed failed, and the failure is propagated to higher-level goals. Finally, CAN language also provides semantics for the concurrency of the agent programs to enable interleaving steps from different intentions. For instance, an intention to go out for dinner can be interleaved with an intention to buy milk, e.g. by buying bread on the way back from dinner rather than buy milk after reaching home from dinner. To sum up, the language of CAN is the superset of AgentSpeak language. From now on, we stick to the CAN language, whose syntax and formal semantics are provided in a fine detail in Section 2.3.

While CAN language is an excellent framework to model intelligent CAN agents, the resulting agent still can fail to accomplish intentions in a realistic environment pervaded by uncertainty. In particular, CAN agents often rely on a pre-defined plan library to reduce the planning problem to the much simpler problem of plan selection. Although such a pre-defined plan library allows for a fast agent reasoning cycle, it also causes an insurmountable problem to CAN agent programmers to obtain a plan library which can cope with every possible eventuality. Unfortunately, a plan library which covers every possible eventuality is not always available, particularly when dealing with uncertainty. Furthermore, the agent can still suffer from the failure of execution of agent programs, even if the plan library is adequate. For example, let it be a CAN agent which is

executing a plan containing an action to open a door. Despite fully executing the action of door-opening, the agent can still fail to open the door (e.g. the door is jammed). This kind of undesirable situation particularly holds true in an environment pervaded by uncertainty, e.g. actions have stochastic effects. To cope with the potential execution failure of the agent programs, however, planning (which is reviewed in Section 1.3) can automatically create a new plan from actions to achieve a goal for which either no pre-defined plan exists or worked.

Indeed, planning can augment the range of the behaviours (i.e. the plans) of the CAN agent by generating new plans to adapt to the changes in an environment (which we discuss in Chapter 4). However, it will be even more beneficial for a CAN agent if it can also remember the new plans generated by, e.g. the external planning tools. Intuitively, it is similar to how human beings learn. We will refer to the step of adopting new plans as plan library expansion. However, merely adding plans is not enough for an agent. As the agent ages, some plans may become unsuitable, hampering its reactive nature which is crucial to the success of CAN agents. For instance, an approach to an event (e.g. the need to enter another room) which worked in the past (e.g. turning a handle) may no longer work in the future (e.g. the handle has been removed, and a button needs to be pressed instead). Therefore, there is a need for plan contraction as well, which we refer as plan library contraction. However, plan library contraction is an altogether more significant – albeit challenging – problem than the plan library expansion. Unlike plan library expansion, plan library contraction relies on both qualitative and quantitative measures associated with each plan in the library to determine which plans are no longer deemed valuable and so can be removed. For example, a plan may be flagged for deletion because it became obsolete (e.g. a low number of calls) or because it became incorrect (e.g. a high failure rate). Due to the inherently structural nature of a plan library, however, care must also be taken when deleting plans to avoid undesirable side-effects. To illustrate, given a plan with a mild failure rate, the deletion of such a plan should not be recommended if, e.g. the agent has no alternative to replace it at the time. Therefore, plan library contraction process must be conducted with the consideration of qualitative and quantitative measures, and structural properties associated with each plan.

Finally, the intelligent agent is typically expected to respond to new events while always dealing with other events. Indeed, CAN agents support concurrent execution of goals in an interleaved manner. Therefore, it is naturally expected that the CAN agent should be sensible in the way it pursues multiple goals. In fact, there are often some complex interactions within the CAN agent which could cause itself to fail in achieving its intentions. For example, a previously achieved effect for an action in one intention may be undone by a step in another intention before such an action that relies on it begins executing, thus preventing this action from being able to execute (i.e. deadlock). Therefore, it is critical for the agent to avoid harmful interference between intentions as well. However, to avoid execution inefficiency, the agent also should capitalise on positive interactions between intentions. Opportunities for positive interactions between intentions enable the agent to reduce the effort (e.g. resources) it exerts to accomplish

its intentions. In particular, positive interactions exist when intentions overlap with each other. In this case, the agent with the overlapping intentions can merge its intentions (effectively allowing one to skip some of its plan steps in its plan) to reduce the overall execution cost. While, unlike negative interference, exploiting commonality of intention is necessary for the agent to perform its tasks correctly, it can be of vital importance in a resource-critical domain such as in the autonomous manufacturing section [SWH06]. Before we can address the aforementioned limitations of CAN agents, we first look at what planning is, and how it links with, yet is different from CAN agents.

1.3 Planning

In the previous sections, we introduce the BDI paradigm and a particular line of classical BDI languages, e.g. CAN, to representing intelligent agents. Recall that CAN agents are usually equipped by a library of plans for achieving different goals. Such a library of plans is usually pre-programmed in a suitable high-level language with procedural knowledge so that the agent can choose its own suitable way of achieving the given goal depending on the current situation. For this reason, the problem in CAN agents is effectively solved by the programmer in her (or his) head, and the solution is expressed afterwards, e.g. as a collection of rules. To some extent, it is indeed useful and important that the system possesses a wealth of pre-compiled procedural knowledge about how to function [GL86], e.g. ensuring that the goals can be achieved efficiently in a dynamic environment. However, such an approach unavoidably puts all the burden on the programmers who may not be able to anticipate all possible contingencies. Therefore, CAN agents could result in systems that tend to be brittle, in particular, in a hostile environment. In contrast to the CAN agent approach, planning is an approach in which a plan instructing which actions to execute is derived automatically from a model consisting of, e.g. the specifications of actions and goals. Instead of hand-crafting specific procedural knowledge in CAN agents, planning is interested in formalising a representation of problem (i.e. model) and finding a general way to solve this model (e.g. achieving the given goal) which can normally be scaled up to large and meaningful instances. In general, the solution of a planning model results in the selection of a sequence of actions which can start from the initial state and should end with the goal state. Therefore, planning is often defined as the branch of Artificial Intelligence (AI) concerned with the "synthesis of plans of actions to achieve goals".

To build a planning system, it usually requires three parts: (i) the model that express, e.g. the dynamics and goals of the system; (ii) the languages that express the model in a compact and computable form; and (iii) the algorithms that use the representation of the model for selecting the action to do next. Overall, these three parts interact with one another. On the one hand, the complex dynamics and goals in a planning system naturally require expressive models and languages. On the other hand, the expressive models and languages, in turn, require efficient

and tractable planning algorithms to select the actions. To aggregate the computational issue in planning, planning is typically required to satisfy the principles of scalability and generality. Intuitively, the scalability requires the planning tools to accept the same problem in any size, whereas the generality demands the planning tools to allow a description of any problem in terms of the same mathematical model. Fortunately, the past several decades have witnessed significant and promising advances in all three components of planning in the planning community (seen in the book written by Hector Geffner and Blai Bonet [GB13]). We now present a succinct review of the mechanisms of these components in a planning system for the purpose of utilising planning in BDI agents in Chapter 5 and Chapter 6.

The basic model of planning is the so-called First-principles Planning (FPP) (also called classical planning) model in which the environment is fully observable, actions are deterministic, and the initial state is fully known. The task of FPP is to drive a system from a given initial state into a goal state by applying actions whose effects are deterministic and known. To solve an FPP problem, the key is to solve a search problem. In fact, an FPP problem can be formulated as a path-finding problem over a directed graph (the fundamentals of the graph is discussed in Section 2.5) whose nodes represent the states of the environment, and whose edges capture the state transitions that the actions make possible. As such, the problem of an FPP problem is equivalent to the path-finding problem in a directed graph. In principle, there are two main categories of path-finding algorithms, namely forward search and backward search. As the name of forward search intuitively conveys, this style of search starts from the initial state and keeps enumerating all applicable actions forward until the goal is reached. The style of backward search, contrarily, starts at the goal and applies the actions backward until it finds a sequence of actions that lands in the initial state. Despite the intuitions in these two searches, none of them scales up well as they are all uninformed, thus blind when searching a large state space. To make the search informed about the direction to a goal, it is common to use heuristic functions. In general, a heuristic function is derived from the specification of the planning instance and used for guiding the search through the state space. For example, the classical FF planning system [HN01] uses information about the “helpful actions" to select a set of promising successors to a state to prune the search space. To some extent, a key accomplishment in the modern planning research community is to derive useful heuristics in an automatic fashion from the representation of the problem itself [BLG97].

While the FPP model captures some nice features of the environment, there are many planning problems which may exhibit features that do not naturally fit into the classical planning. To name one, the initial situations are often not fully known, e.g. uncertain information about the initial situation. Instead of proposing correspondingly adapted models with extra modelling complexity, however, it has been proven successful in tackling these more complicated scenarios by applying some suitable transformations to the existing FPP problems. This arguably explains why classical planning remains an active research area in the planning community. For example, in the

cases where the system is deterministic and all uncertainty only lies in the initial situation such as in [PG09], a translation can be performed by considering all possible initial situations to remove the uncertainty, thus obtaining a classical planning. Furthermore, there are other two important planning models which need attention, namely Markov Decision Processes (MDPs) [GR13] and Partially Observable Markov Decision Processes (POMDPs) [Mon82] (which are discussed in more detail in Section 2.4.3). Whereas MDPs generalise the model of FPP by allowing actions with stochastic effects and fully observable state, POMDPs further extends MDPs by allowing states to be partially observable through sensors that map the true state of the environment into observable tokens. After years of development by planning researchers, there is a variety of MDPs and POMDPs models with many highly efficient algorithms [GNT04].

Finally, regarding the planning language, the planning research community has also settled on standardised representations. The first and probably simplest language in use is Stanford Research Institute Problem Solver (STRIPS) [NF71]. In STRIPS, each state of the world is represented by a set of grounded atoms and a goal formula is built from the ground atoms using the normal connectives. For example, we can have that `in(london)` (resp. `have(report)`) is a grounded atom encoding the information of being in London (resp. having report), whereas `in(london)` $\wedge$ `have(report)` is a goal formula ensuring both being in London and having report. It is noted that the predicate `in` and `have` are explicitly declared by model programmers to take one variable. Meanwhile, each action is defined by an action description consisting of two main parts: a description of the effects of the action, and the conditions under which the action is applicable. For example, the following is the action description of flying from London to Shanghai.

`fly(london, shanghai)`

pre-conditions: `at(london)`, post-effects: `at(shanghai)`.

where the predicate `fly` and `at` are similarly declared by model programmers to take two variables and one variable, respectively. In practice, in order to easily compute the effects of action application in the environment, these effects are simply described by two lists, namely the delete list and add list. Whereas the delete list specifies those predicates that are no longer true, thus being deleted, the add list contains those predicates that are added and are regarded as being true. Therefore, the same action description of flying a plane from London to Shanghai can be revised and shown as follows.

`fly(london, shanghai)`

pre-conditions: `at(london)`, delete list: {`at(london)`}, add list: {`at(shanghai)`}.

However, due to the limited expressiveness of STRIPS language, Action Description Language (ADL) [Ped89] is proposed to support some other important features such as the condition effects. Later on, to provide a common formalism for describing planning problems in International Planning Competition (IPC), Planning Domain Definition Language (PDDL) [MGH$^+$98] is proposed as an official language in the planning competitions. In a nutshell, PDDL accommodates the STRIPS and ADL languages along with a number of additional syntactic constructs, e.g.

requirements feature which supports the articulation of the different level of expressiveness. In PDDL languages, the planning problems are expressed in two parts: one about the general domain; the other about a particular domain instance. Such a division has facilitated the empirical evaluation of a planner performance and the development of standard sets of problems in comparable notations. Finally, we note that the details of basic STRIPS formalism and PDDL representation are provided in Section 2.4.2 and Section 2.4.3 in more details.

1.4 Thesis Outline

The CAN language provides a powerful and flexible framework which is capable of modelling intelligent agents in complex and dynamic environments. However, the lack of abilities in the resulting CAN agents to create new plans, to adapt to the environment, and to manage intention interleaving limits its practical capability in an environment pervaded by uncertainty.

The aim of this thesis is to develop the planning extension of CAN agents to utilise planning for robust agent program execution, adaptive plan library, and efficient intention progression. To start with, we are interested in ways of extending the CAN agents such that planning ability can be employed to recover the execution failure when it needs most in Chapter 4. To this end, we propose a novel operational semantics for incorporating planning as an intrinsic planning capability that increases the robustness of a CAN agent by exploiting the full potential of declarative goals. In achieving this, we introduce a declarative goal intention to keep track of declarative goals used by planning and develop a detailed specification of the appropriate operational behaviour when planning is pursued, succeeded or failed, suspended, or resumed in CAN agents. Also, we prove that CAN agents and planning are indeed theoretically compatible for such a principled integration in both offline and online planning manner. Furthermore, we demonstrate the practical feasibility of this integration by a case study of a smart home.

In this thesis, we also want to propose an extension of CAN agents that allow the resulting agent to learn to adapt to a changing environment. Specifically, we present a plan library evolution architecture with mechanisms to incorporate new plans (plan expansion), e.g. from the automated planning tool, and drop old/unsuitable plans (plan contraction) in Chapter 5 to adapt to changes in a realistic environment. To achieve this objective, we follow a principled approach to a plan library expansion and contraction, motivated by postulates that clearly highlight the underlying assumptions, and supported by measures which are able to characterise plans in the library. The systematic specification of domain-independent characteristics (e.g. the quality of plans) of the plan library forms the basis for the plan library expansion and contraction reasoning. As such, we can define a plan library expansion operator and formally shows the benefits of expansion regarding the relevant characteristics. Meanwhile, a plan library contraction operator is also introduced. Unlike the plan library expansion operator, the contraction operator needs to not only take the earlier characteristics into account, but also balance the need for reactivity, the fragility

of the plan library, and the correctness and overall performance of the agent. To demonstrate the theoretical feasibility of our contraction operator, a concrete multi-criteria decision making is employed to instantiate such an abstract contraction operator.

While the principle integration of planning in CAN agents added an extra layer of robustness when no pre-defined plan worked or existed, there still would be execution failures due to the negative interaction between multiple intentions. Indeed, an agent pursuing multiple goals can encounter the so-called deadlock due to the careless interleaving. Equally, positive interactions can occur between intentions (e.g. common sub-intentions). In Chapter 6 we present another usage of planning in CAN agents to managing the concurrent intention executions. Specifically, the planning is used to exploiting overlapping intentions while resolving conflicts during the interleaved execution of intentions. As such, we show that CAN can think ahead about how to pursue its intentions in the possibly best and most sensible manner. Finally, we also implement our approach and evaluate such an approach empirically in a realistic manufacturing scenario to demonstrate its practical feasibility.

1.5 Overview

Finally, we organise the thesis as follows:

- In Chapter 2 we introduce notations and preliminaries, including an introduction to logic programming, CAN language, and FPP formalism.
- In Chapter 3 we provide a review of the literature on the current state-of-the-art CAN agents, e.g. the utilisation of FPP in CAN agents.
- In Chapter 4 we propose a framework with a strong theoretical underpinning for integrating planning within CAN agents based on their intrinsic relationship.
- In Chapter 5 we describe measures that characterise the performance and structure of plans, and provide rationales to guide the process of plan expansion and plan contraction.
- In Chapter 6 we propose a theoretical framework where planning is employed to manage the intention interleaving in an automated fashion to both guarantee the achievability of intentions and discover and exploit potential overlapping intentions.
- In Chapter 7 we conclude and discuss future work.

CHAPTER 2

PRELIMINARIES

In this chapter we start with some general mathematical notations along with preliminaries on the logic programming which will be used as a formal knowledge representation for Conceptual Agent Notation (CAN) agents. We then introduce some formal concepts and notations of Conceptual Agent Notation (CAN) agents, First-principles Planning (FPP), and some fundamentals of graph theory as a useful structural representation.

2.1 General Notation

We first introduce some mathematical set notations. A set is an unordered collection of distinct objects. The set theory begins with a fundamental binary relation between an object u and a set U. If u is a member of U, the notation $u \in U$ is used for membership. Since sets are also objects, the membership relation can relate sets as well. Let $\subseteq$ (resp. $\subset$) denote set inclusion (resp. strict set inclusion) for the membership among sets. There are also a number of operations on sets. Given two sets U_1 and U_2, then $U_1 \cup U_2 = \{u \mid u \in U_1 \ or \ u \in U_2\}$ denotes the union of U_1 and U_2 whereas $U_1 \cap U_2 = \{u \mid u \in U_1 \ and \ u \in U_2\}$ stands for the intersection of these two sets. The set difference of two sets, denoted as $U_1 \setminus U_2$, is the set of all members of U_1 that are not members of U_2, i.e. $U_1 \setminus U_2 = \{u \mid u \in U_1 \ and \ u \notin U_2\}$. For any non-empty set U_1 and U_2 such that $U_1 \cap U_2 \neq \emptyset$, we can easily have the following membership relations related to the operations: (i) $U_1 \subseteq (U_1 \cup U_2)$, (ii) $(U_1 \cap U_2) \subseteq U_1$, and (iii) $(U_1 \setminus U_2) \subseteq U_1$. The power of a set U, denoted as 2^U, is the set whose members are all of the possible subsets of U. For example, the power set of $\{1,2\}$ is $\{\emptyset, \{1\}, \{2\}, \{1,2\}\}$. Finally, let $|U|$ denote the cardinality of a set of U, i.e. the number of members in U. In addition, a total order over a set U is a binary relation, denoted $\preceq$, if the following hold for all u_1, u_2 and u_3 in U: (i) $u_1 \preceq u_2$ and $u_2 \succeq u_1$ then $u_1 \cong u_2$ (antisymmetry); (ii) if $u_1 \preceq u_1$ and $u_2 \preceq u_3$ then $u_1 \preceq u_3$ (transitivity); and (iii) $u_1 \preceq u_2$ or $u_2 \preceq u_1$ (connexity). Therefore, $u_1 \prec u_2$,

which is also called the strict order, if and only if $u_2 \preceq u_1$ does not hold. We also use standard mathematical symbols $\mathbb{N}$ to refer to the set of natural numbers, $\mathbb{R}$ the set of real numbers, and $\mathbb{R}_{\geq 0}$ the set of non-negative real numbers, For legibility, we use $\mathbf{u}$ as a compact notation for a vector $(u_1,\ldots,u_n)$.

2.2 Logic Programming

In this section we present the syntax and semantics of logic programming. The syntax of logic programming is based on three types of symbols: constant, variable, and predicate symbols. Following the convention in logic programming, variables start with uppercase letters and all others start with lowercase letters. If V is a variable and c is a constant, then we say V and c are terms. If p is a predicate and $t_1,\ldots,t_n$ are terms, then $p(t_1,\ldots,t_n)$ is an atom. A special atom $=(t_1,t_2)$ is an equation in which the predicate p is =, which is often written simply as $t_1 = t_2$. If p is an atom, then p (resp. **not** p) is a positive (reps. negative) literal. The negation symbol **not** in the negative literal **not** p is referred to as "negation as failure", that is, a negative condition **not** p is shown to hold by showing that the positive condition p fails to hold. Therefore, the close world assumption is used in this thesis unless it is specified otherwise. To avoid any confusion, the negations cannot be applied to any terms (e.g. constants and variables) directly. Let h be an atom and $b_1,\ldots,b_n$ literals which are defined as an atom or its negation. A (normal) clause (or rule) is of the form $h \leftarrow b_1 \wedge \ldots \wedge b_n$ with h called the head and $b_1 \wedge \ldots \wedge b_n$ called the body. The clause $h \leftarrow b_1 \wedge \ldots \wedge b_n$ is read as if $b_i \wedge \ldots \wedge b_n$, then h. Informally, if every b_i is true, then h must hold true. If each b_i is a positive literal, then a clause is also called a definite clause. A clause $h \leftarrow true$ is called a fact, and is written simply as h. A logic program is a finite set of clauses. Finally, in logic programming, a term, atom, clause, or a logic program is also called an expression.

Example 1. Consider a logic program which contains the following (definite) clauses and facts:

```
fly(X) ← bird(X) ∧ not flightless(X)
bird(pigeon)
bird(penguin)
flightless(penguin)
```

where `fly`,`bird`,`flightless` are predicates, `X` a variable, and `pigeon`,`penguin` constants. Intuitively, the clause above states that if an `X` is a bird and it is not flightless, then `X` should be able to fly. Given the goal of finding something that can fly, i.e. `fly(X)`, there are two candidate solutions, which solve the first subgoal `bird(X)`, namely `X=pigeon` and `X=penguin`. The second subgoal `not flightless(penguin)` of the second candidate solution `X=penguin` fails, because its positive condition `flightless(penguin)` holds. However, the second subgoal `not flightless(pigeon)` of the first candidate solution `X=pigeon` succeeds, because `flightless(pigeon)` does not hold,

i.e. "negation as failure". Therefore, `X=pigeon` is the only solution of the goal. In other words, it can be concluded from this logic program that the atom `fly(pigeon)` holds true. □

The form of reasoning we have intuitively applied in the above example is called substitution, or variable binding, e.g. `X=penguin`. A substitution θ is a finite set of form $\{V_1/t_1,\dots,V_n/t_n\}$ where each $V_1,\dots,V_n$ are distinct variables, and $t_1,\dots,t_n$ are terms such that $V_i \neq t_i$ for $i \in \{1,\dots,n\}$. We also call θ a ground substitution if $t_1,\dots,t_n$ are all constants, i.e. it is variable-free. Given an expression ϕ and a substitution $\theta = \{V_1/t_1,\dots,V_n/t_n\}$, the expression $\phi\theta$ stands for substituting each occurrence of V_i in ϕ with t_i for $i \in \{1,\dots,n\}$. A substitution θ is also called a unifier of two terms t_1 and t_2 if $t_1\theta = t_2\theta$. It is possible that no unifier for given two terms exists. For example, X and $p(X)$ cannot be unified. Often there are more than two unifiers existing for two terms. When more than two unifiers exist, we are interested in a particular type of unifier, namely most general unifier (mgu). Intuitively, a substitution θ is deemed a mgu of two terms t_1 and t_2 if θ itself is a unifier of these two terms and no more specific than any other unifier θ' of t_1 and t_2. By being no more specific than θ', it means that it is always feasible to substitute for some of the variables of θ and get θ'. Formally, a substitution θ is called a mgu of two terms t_1 and t_2 if θ is unifier of t_1 and t_2, and for any unifier θ' of t_1 and t_2, there always exists another unifier λ such that $t_i\theta' = (t_i\theta)\lambda$ where $i = 1,2$. For example, let two terms be $t_1 = p(X,Z)$ and $t_2 = p(Y,Z)$. We can have $\theta = \{X/Y\}$ as one mgu of t_1 and t_2. Given another unifier $\theta' = \{X/Y, Z/a\}$, we have $t_i\theta' = (t_i\theta)\lambda$ where $\lambda = \{Z/a\}$ and $i = 1,2$. We note here that there can be more than one most general unifier. However, such substitutions are the same except for variable renaming. For instance, both $\{X/Y\}$ and $\{Y/X\}$ are a mgu to $t_1 = p(X,a)$ and $t_2 = p(Y,a)$ with only variable name being different. Further to this, we assume the existing algorithm for finding the most general unifier of a set of expressions (e.g. in [BS01]) but will not discuss in any further detail in this thesis. The interested readers are referred to the work of [Rob92, Kow83] for a complete count of logic programming foundations.

To define the semantics of logic programming, a Herbrand model is used to discuss the truth or falsity of a logic program which is a finite set of clauses. The Herbrand base of a logic program is the set of ground atoms with regard to the set of all possible ground terms, i.e. containing no variables. This set of all possible ground atoms is also called the Herbrand universe. A Herbrand interpretation is a mapping from the Herbrand base to the set of truth values $\{\top,\bot\}$. For convenience, we will also refer a Herbrand interpretation ω by the set of atoms $\{p \in H \mid \omega(p) = \top\}$, i.e. we simply list only those atoms that are true, where H is the Herbrand base. A Herbrand interpretation is a Herbrand model of a definite clause $h \leftarrow b_1 \wedge \dots \wedge b_n$ if for every substitution θ such that $b_i\theta$ is a (set-theoretic) member of interpretation, $h\theta$ is also a member of the interpretation. In other words, when the body is true, it requires the head - or conclusion - to be true as well. A Herbrand interpretation is a Herbrand model of a definite clause logic program if it is a Herbrand model of every clause in the logic program. Definite clauses are guaranteed to have a unique minimal (by set inclusion) Herbrand model, called the least

Herbrand model. A definite clause program is said to entail an atom p iff p is a member of the least Herbrand model of this definite clause program. Thus, the least Herbrand model defines the semantics of definite clauses. Throughout this thesis we always assume that we are working with definite clauses, hence this least Herbrand model always exists. Further to this, we assume the standard semantics of logic programming for definite clauses but will not discuss these in detail.

2.3 CAN

We introduce the CAN [WPHT02, SP11] to formalise the behaviours of a classical Belief-Desire-Intention (BDI) agents. In a nutshell, CAN language, being a superset of AgentSpeak [Rao96], provides the first and foremost operational semantics of AgentSpeak to capture the behaviours of AgentSpeak formally.

2.3.1 Syntax

A CAN agent can be specified by a 3-ary tuple $\langle \mathcal{B}, \Pi, \Lambda \rangle$ where $\mathcal{B}$ represents its initial belief base, Π its plan library, and Λ its action description library. The syntax of a CAN agent can be constructed by three types of predicates, namely event predicate e, belief predicate b, and action predicate act. Similar to the logic programming, terms (resp. vector terms) in CAN are denoted as t (resp. $\mathbf{t}$). Therefore, we can write $e(\mathbf{t})$, $b(\mathbf{t})$, and $act(\mathbf{t})$ to denote the atoms for events, beliefs, and acts, respectively.

The belief base $\mathcal{B}$ of an agent is a set of ground belief atoms, which is also known as facts (e.g. `bird(penguin)`), which encodes what the agent believes about the world. Formally, if b is a belief predicate, and $\mathbf{t} = \{t_1, \ldots, t_n\}$ are terms, then $b(\mathbf{t})$ is a belief atom. If $b(\mathbf{t})$ and $c(\mathbf{s})$ are belief atoms, then $b(\mathbf{t})$, **not** $b(\mathbf{t})$, and $b(\mathbf{t}) \wedge c(\mathbf{s})$ are beliefs. A belief atom or its negation is also referred to as a belief literal where the negation is referred to as “negation as failure” same as in the logic programming. A ground belief atom is usually called a base belief which is a member of the belief base of the agent. In addition, given a belief formulas ϕ constructed from basic belief with the conventional logical connectives, we assume here that the entailment operators are available for checking whether the belief formulas ϕ is a logical consequence over the belief base of an agent (i.e. $\mathcal{B} \models \phi$). Also, operators (e.g. AGM belief revision [AGM85]) are assumed available to add a belief base b to a belief base $\mathcal{B}$ (i.e. $B \cup \{b\}$) and delete b from $\mathcal{B}$ (i.e. $B \setminus \{b\}$).

The plan library Π encodes the operational information of the domains for the agent to execute. It is a collection of plan rules of the form $e(\mathbf{t_1}) : \varphi(\mathbf{t_2}) \leftarrow P(\mathbf{t_3})$ with $e(\mathbf{t_1})$ the triggering event, $\varphi(\mathbf{t_2})$ the context condition, $P(\mathbf{t_3})$ the plan-body, and $\mathbf{t_i}$ a vector of terms ($i = 1,2,3$). For convenience, the expression to the left of the arrow is also called the head of the plan, which distinguishes itself from the plan-body part. In detail, the triggering event in the head of a plan specifies why the plan is triggered. Intuitively, when an agent is required to handle a new event goal or notices a change in its environment, it may trigger additions or deletions to its events or

beliefs. Let $b(\mathbf{t})$ be a belief atom and $e(\mathbf{s})$ be an event goal. Formally, a triggering event can be (i) a belief addition $+b(\mathbf{t})$ (i.e. addition of $b(\mathbf{t})$ to the belief base); (ii) a belief deletion $-b(\mathbf{t})$ (i.e. deletion of $b(\mathbf{t})$ from the belief base); (iii) an event goal $!e(\mathbf{s})$ (i.e. an event goal posted to the agent to respond to); or (iv) a test goal $+?b(\mathbf{t})$ (i.e. test $b(\mathbf{t})$ to be true or not). These forms of triggering events will be the part of basic building blocks for the plan-body (which is discussed later on).

The plan-body is also referred to as the agent program in CAN language as it lists the instructions to execute for the agent. The basic building blocks used in the plan-body P is defined by the following syntaxes: $P ::= \ act \mid ?\varphi \mid \ +b \mid \ -b \mid !e$. We now intuitively explain the meaning of each of these syntactic components, before giving their formal semantics in CAN in Section 2.3.2.1. The syntax act stands for a primitive action which represents the things the agent is capable of doing. The syntax $?\phi$, as a test goal, tests whether or not the belief ϕ can be entailed from the belief base $\mathcal{B}$. Unlike the context condition of a plan which can only check the information before the execution of a plan, the text $?\phi$ allows getting the latest information which might only be, e.g. available during the execution. The syntax $+b$ and $-b$ are respectively belief addition and belief deletion. The effect of $+b$ is that it adds the base belief b to the belief base $\mathcal{B}$ (i.e. $\mathcal{B} \cup \{b\}$) if it is not already there. Conversely, $-b$ removes the base belief b from the belief base $\mathcal{B}$ (i.e. $\mathcal{B} \setminus \{b\}$) if it is there. These two syntaxes $+b$ and $-b$ enable the agent to update its belief base proactively. The syntax $!e$, as an event goal, denotes a pending event to which an agent needs to respond. In order to distinguish an event goal $!e$ from a triggering event e, the "!" symbol is prefixed before an event atom. By allowing a new event in plan-body, it supports the so-called complex behaviour which requires more than simple sequences of actions to execute. In fact, an agent needs to address events, e.g. achieving an event goal, before further actions can be taken.

Next we present the various order relations available in CAN language when assembling the building blocks of plan-body introduced above. In CAN, the symbol ; is usually used for denoting sequencing relation between two agent programs. For example, we can have the syntax $P_1;P_2$ which specifies the order of execution of P_1 and P_2, i.e. P_1 followed by P_2. Also, the syntax $P_1;P_2$ also implies that the agent cannot start the execution of P_2 unless P_1 is achieved in full. However, it is often the case that one program P_2 cannot wait until another program P_1 is fully achieved in full. In other words, the agent need to pursue two program P_1 and P_3 concurrently. In CAN, it also supports concurrent execution of agent programs. Formally, the syntax $P_1 \parallel P_2$ is used to stand for concurrent execution of P_1 and P_2. However, the concurrency execution in CAN does not exactly requires the agent to, e.g. execute two actions simultaneously. Rather, it allows the agent to pursue the multiple programs in an interleaved fashion, e.g. one step in one program and next step in another program. Therefore, $P_1 \parallel P_2$ is often called interleaved concurrency of the agent program P_1 and P_2.

So far, it can be seen that the agent programs introduced solely tell the agent what to do through a set of procedures P, e.g. performing an action `drink(water)`. However, it is often the case that the agent may only care about a declarative description of state sought, e.g. `not`

`being(thirsty)`. While the procedural aspects of programs can be achieved efficiently in a dynamic environment, by omitting the declarative aspects of programs, the agent loses its ability to reason about its programs. For instance, without knowing what a state of affairs that an agent is trying to achieve, one cannot check whether the state is achieved, or check whether such a state is even impossible to reach. To address the lack of declarative aspect support in agent programs, CAN language also provides a goal program $goal(\varphi_s, P, \varphi_f)$ which states that the successful condition φ_s should be achieved through the procedural program P, failing when the failure condition φ_f becomes true and retrying (alternatives) as long as neither φ_s nor φ_f is true. In detail, the goal program $goal(\varphi_s, P, \varphi_f)$ first provides explicit procedure P which specifies how the agent might bring about the desired success condition φ_s. Furthermore, it also encodes when it is deemed impossible to accomplish, i.e. the failure condition φ_f. The importance of the success condition φ_s and the failure condition φ_f is that they decouple the success and failure of the desired state from the success or failure of its related plan. As a result, the goal program $goal(\varphi_s, P, \varphi_f)$ will not be dropped merely bcause a plan to achieve a state has failed. Similarly, a state cannot be assumed achieved just because the plan which is written to achieve it has executed fully.

A number of auxiliary agent programs are also used internally in the full program language, namely $nil \mid e : (|\varphi_1 : P_1, \cdots, \varphi_n : P_n|) \mid P_1 \rhd P_2 \mid goal(\varphi_s, P, \varphi_f)$. The first auxiliary program nil denotes a terminating program, i.e. nothing left to execute. Recall that the triggering event in its head triggers a plan in CAN language. Therefore, given an event e, the agent can retrieve a set of plans, e.g. $e \leftarrow \varphi_i : P_i$ ($i \in \{1, \ldots, n\}$), whose triggering event matches the given event e. Normally, this set of plans is called the relevant plans for the event e, which encodes all possible choices of plans to respond to an event e. To compactly denote this information, the program $e : (|\varphi_1 : P_1, \cdots, \varphi_n : P_n|)$ is introduced. It means that given an event e, a plan-body P_i can be executed if its related context condition φ_i holds where $i \in \{1, \ldots, n\}$. Regarding the auxiliary agent program $P_1 \rhd P_2$, it provides the failure recovery by executing P_2 only on failure of P_1. The failure handling mechanism $\rhd$, which is a distinguishing feature of CAN agents, specifies the kind of behaviours when the current plan does not go well as expected. Intuitively, it enables trying alternative plans for addressing an event if the current strategy failed. To implement this type of failure handling, it is typically accomplished by combining program $e : (|\varphi_1 : P_1, \cdots, \varphi_n : P_n|)$ and $P_1 \rhd P_2$. For example, the program $(\varphi_1 : P_1) \rhd e : (|\varphi_2 : P_2, \cdots, \varphi_n : P_n|)$ says that the current strategy $\varphi_1 : P_1$ is selected to address the event e while maintaining the rest of possible alternative plans (i.e. $\varphi_2 : P_2, \cdots, \varphi_n : P_n$) to consider if P_1 failed.

Finally, the action library Λ is a collection of actions a in the form of $a(\mathbf{x}) : \psi(\mathbf{x}) \leftarrow \phi^-(\mathbf{x}); \phi^+(\mathbf{x})$. We have that $\psi(\mathbf{x})$ is called the precondition, and $\phi^-(\mathbf{x})$ and $\phi^+(\mathbf{x})$ denoting a delete and add set of belief atoms, respectively. We note that we adopt STRIPS-like action formalism (discussed in details in Section 2.4.2) for simplicity. However, there is nothing preventing these actions from being much richer, such as powering the wheels on a rover.

2.3.2 Semantics

The semantics of the CAN agent state what are the legal execution of an agent. The operational semantics for a CAN agent are defined in terms of configurations C and transitions $C \to C'$. Intuitively, the configuration C is the current state which the agent is in, and it encapsulates, among others, its belief base and the current state of partially executed plans. A transition moves our agent from one configuration into another one. A transition $C \to C'$ denotes that executing a single step in configuration C yields C'. We write $C \to$ (resp. $C \nrightarrow$) to state that there is (resp. is not) a C' such that $C \to C'$. Also, we denote $\xrightarrow{*}$ as the transitive closure of $\to$. Intuitively, the transitive closure of $\xrightarrow{*}$ describes the reachability from one configuration to another configuration based on the given defined transition $\to$. To move one configuration to another, there are derivation rules specifying in which cases an agent can transition to a new configuration. A derivation rule consists of a (possibly empty) set of premises p_i and a single transition conclusion c, denoted by

$$\frac{p_1 \quad p_2 \quad \cdots \quad p_n}{c} \quad l$$

where l is a label for reference. Intuitively, it says that if p_i (e.g. a context condition of a plan) holds for $\forall i \in \{1,\ldots,n\}$, then a conclusion c (e.g. a plan can be selected) can be obtained. To simplify the explanation of the semantics of CAN, the semantics of CAN agent is usually specified by two different levels of transition forming two layers. The first type of transition is defined regarding the intention-level configuration specifying how to evolve a single intention. Such an execution at the intention-level will only affect the internal state of the agent, i.e. its belief base and the current intention. The second type of transition is defined as the agent-level configuration characterising how to execute a complete agent. Typically, the agent-level configuration captures the evolution of the intentions of an agent, e.g. adopting or dropping intentions. We now discuss each group separately for legibility and present the related derivation rules in the next section.

2.3.2.1 Intention-level Execution

We present the derivation rules characterising the transition of the intention-level configuration in the form of $\langle \mathcal{B}, \mathcal{A}, P \rangle$ in which $\mathcal{B}$ denotes the belief base, $\mathcal{A}$ the sequence of actions that have been executed by an agent, and P the program that is being executed (i.e. the current intention).

We first start with a simple derivation rule which characterises the belief addition in CAN agents. Recall that $+b$ as an agent program instructs the agent to adds the base belief b to the belief base $\mathcal{B}$ (i.e. $\mathcal{B} \cup \{b\}$) if it is not already there. Therefore, we can have the following derivation rule to formalise this instruction in an intention-level configuration where $+b$ is the current intention:

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, +b \rangle \to \langle \mathcal{B} \cup \{b\}, \mathcal{A}, nil \rangle} \quad \text{+b}$$

It can be noted that after the program of belief addition is executed, it becomes nil which indicates that the program terminates here now. Since adding a base belief is done by the set union operation, the belief base remains unchanged if such a base belief is already there. Therefore, there is no premise required in the derivation rule. Similarly, we can have the rule for belief deletion with set difference operation being performed in the belief base as follows:

$$\frac{}{\langle \mathcal{B}, \mathcal{A}, -b\rangle \rightarrow \langle \mathcal{B} \setminus \{b\}, \mathcal{A}, nil\rangle} \quad \texttt{-b}$$

We are now ready to look at a derivation rule which requires one premise to hold before reaching a conclusion, namely the test goal $?\phi$. Recall that the test goal $?\phi$ is used to check whether a belief condition ϕ holds according to the current belief base $\mathcal{B}$. If the belief condition ϕ holds (i.e. $\mathcal{B} \models \phi$), then the test goal succeeds (thus terminating as nil). Therefore, the following derivation rule can be given to account for the execution of test goal operation:

$$\frac{\mathcal{B} \models \phi}{\langle \mathcal{B}, \mathcal{A}, ?\phi\rangle \rightarrow \langle \mathcal{B}, \mathcal{A}, nil\rangle} \quad ?$$

Next we present a derivation rule for executing an action. Recall that an action in form of $a : \psi \leftarrow \phi^-; \phi^+$ first says that the agent can only execute this given action if the precondition holds $\mathcal{B} \models \psi$ in the current belief base. Secondly, the successful execution of such an action will delete and add the set of belief atoms ϕ^- and ϕ^+, respectively, from the belief base $\mathcal{B}$ resulting in $(\mathcal{B} \setminus \phi^-) \cup \phi^+$. Since any action needs to be instantiated (i.e. a ground atom) before it can be executed, we need to make sure that any variables used in the action must be replaced by constants, i.e. a substitution θ, Therefore, we have the following rule for the execution of an action where θ is the suitable substitution, $\mathcal{A} \cdot act$ records the execution of act, and nil denotes the termination state.

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

We now look at the final basic agent program $!e$, i.e. an event goal. Recall that the agent program $!e$ denotes a pending event that an agent needs to respond to. To deal with $!e$ (which can be either internal or external), there is a three-stage process to address the pending event. The first stage is to collect a (non-empty) set of the plans whose triggering events match the pending event subject to a mgu (i.e. $\theta = \text{mgu}(e', e)$). Such a set of plans is also called the relevant plans. The purpose of a mgu unification is to deter the rest of variables in the context condition and plan-body of the given plan from being instantiated. Deterring the instantiation of the context condition and plan-body until they have to allows BDI agents to respond flexibly to the current state of the environment. As standard, we maintain the set of relevant plans Δ of the form $\langle \varphi : P\rangle$ where φ (resp. P) is the context condition (resp. plan-body) of a plan whose triggering event e' matches the actual event goal e subject to a mgu θ. Therefore, we can have $\Delta = \{\varphi\theta : P\theta \mid (e' = \varphi \leftarrow P) \in \Pi \wedge \theta = \text{mgu}(e', e)$. Formally, we can have the following derivation rule

to define the process of retrieving a set of relevant plans for a given event where the pending event $!e$ is transitioning to the set of its relevant plans if such a set of relevant plans exists.

$$\frac{\Delta = \{\varphi\theta : P\theta \mid (e' = \varphi \leftarrow P) \in \Pi \land \theta = \text{mgu}(e', e)\}}{\langle \mathcal{B}, \mathcal{A}, !e \rangle \to \langle \mathcal{B}, \mathcal{A}, e : (\mid \Delta \mid) \rangle} \quad \texttt{event}$$

The second stage of responding to an event goal $!e$ is to select one applicable strategy P_i from the set of relevant plans $\Delta = e : (|\varphi_1 : P_1, \ldots, \varphi_n : P_n|)$. A strategy option P_i is applicable if the related context condition φ_i is believed true where $i \in \{1, \ldots, n\}$. In this case, the following derivation rule `select` constructs an auxiliary program in the form of $P\theta \rhd e : (\mid \Delta \setminus \{\varphi : P\} \mid)$, where $P\theta$ denotes the selected plan with bindings θ and $\mid \Delta \setminus \{\varphi : P\} \mid$ the new set of remaining plans. We can see that the rule `select` combines $e : (\mid \Delta \mid)$ and the construct $\rhd$. Also the new set of remaining strategies ensures that only strategy options that are not P will be considered if the current strategy P failed:

$$\frac{\varphi : P \in \Delta \quad \mathcal{B} \models \varphi\theta}{\langle \mathcal{B}, \mathcal{A}, e : (\mid \Delta \mid) \rangle \to \langle \mathcal{B}, \mathcal{A}, P\theta \rhd e : (\mid \Delta \setminus \{\varphi : P\} \mid) \rangle} \quad \texttt{select}$$

The third stage in handling an event goal $!e$ involves coping with the situation when the current strategy is unable to execute further. Indeed, this situation may happen, particularly in an environment which is dynamic and uncertain. For example, the precondition of an action act in a plan-body P_1 may not hold right before being executed, thus resulting in the failure of the current strategy. Formally, when the current non-empty strategy P_1 (i.e. $P_1 \neq nil$) in an agent program of the form $P_1 \rhd P_2$ has no next legal intention-level transition (i.e. $\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \not\to$), a new derivation rule $\rhd_\bot$ is introduced to try some alternative strategy in P_2, if applicable for execution (i.e. $\mathcal{B}, \mathcal{A}, P_2 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_2' \rangle$) to avoid the undesired outcome.

$$\frac{P_1 \neq nil \quad \langle \mathcal{B}, \mathcal{A}, P_1 \rangle \not\to \quad \langle \mathcal{B}, \mathcal{A}, P_2 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_2' \rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_2' \rangle} \quad \rhd_\bot$$

When the agent has selected an applicable strategy and it can transition to the end without failure, then it must be executed in full to completion. To this end, the following two derivation rules serve executing the current strategy one step (rule $\rhd_{\texttt{seq}}$) and to finish its execution in full (rule $\rhd_\top$). In detail, the rule $\rhd_{\texttt{seq}}$ says that if the current strategy P_1 can be progressed (i.e. $\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_1' \rangle$), then it should be continued (i.e. $\langle \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_1' \rhd P_2 \rangle$). Similarly, if the current strategy is successfully completed to handle an event, then the whole program including the recovery structure $\rhd$ completes shown in the rule $\rhd_\top$. Therefore, it can be understood that the failure handling mechanism does not intervene and it would only start to operate when the agent gets stuck with the current strategy (see the rule $\rhd_\bot$).

$$\frac{\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_1' \rangle}{\langle \mathcal{B}, \mathcal{A}, P_1 \rhd P_2 \rangle \to \langle \mathcal{B}', \mathcal{A}', P_1' \rhd P_2 \rangle} \quad \rhd_{\texttt{seq}} \qquad \frac{}{\langle \mathcal{B}, \mathcal{A}, (nil \rhd P_2) \rangle \to \langle \mathcal{B}', \mathcal{A}', nil \rangle} \quad \rhd_\top$$

We now look at the derivation rules to account for the sequencing order ; between agent programs. Recall that the program $P_1;P_2$ specifies that the execution of P_1 is followed by the execution of P_2. To be precise, the sequencing order has two-level operational meanings. Firstly, it says that the first program P_1 should evolve itself steps by steps until completion, provided it is possible. Secondly, the second program P_2 cannot start evolving itself unless P_1 is finished. The following two rules, namely seq and $\mathtt{seq}_\top$ defines these two-level operations, respectively:

$$\frac{\langle \mathcal{B},\mathcal{A},P_1\rangle \to \langle \mathcal{B}',\mathcal{A}',P_1'\rangle}{\langle \mathcal{B},\mathcal{A},(P_1;P_2)\rangle \to \langle \mathcal{B}',\mathcal{A}',(P_1';P_2)\rangle}\ \mathtt{seq} \qquad \frac{\langle \mathcal{B},\mathcal{A},P\rangle \to \langle \mathcal{B}',\mathcal{A}',P'\rangle}{\langle \mathcal{B},\mathcal{A},(nil;P)\rangle \to \langle \mathcal{B}',\mathcal{A}',P'\rangle}\ \mathtt{seq}_\top$$

The rule seq progresses a sequence by evolving its first part (i.e. $\langle \mathcal{B},\mathcal{A},(P_1;P_2)\rangle \to \langle \mathcal{B}',\mathcal{A}',(P_1';P_2)\rangle$) if it can be evolved (i.e. $\langle \mathcal{B},\mathcal{A},P_1\rangle \to \langle \mathcal{B}',\mathcal{A}',P_1'\rangle$). The rule $\mathtt{seq}_\top$ does it by progressing the second part of the sequence (i.e. $\langle \mathcal{B},\mathcal{A},(nil;P)\rangle \to \langle \mathcal{B}',\mathcal{A}',P'\rangle$) only when the first program is completed (i.e. $nil;P$) and the second part can be evolved (i.e. $\langle \mathcal{B},\mathcal{A},P\rangle \to \langle \mathcal{B}',\mathcal{A}',P'\rangle$).

Unlike the sequenced programs, a concurrent program $P_1 \| P_2$ may be evolved by evolving either parts independently, provided they can be evolved. However, in order to successfully terminate the concurrent program $P_1 \| P_2$, both parts need to be terminating. Therefore, we can have the following set of rules to capture the behaviours of the concurrent programs:

$$\frac{\langle \mathcal{B},\mathcal{A},P_1\rangle \to \langle \mathcal{B}',\mathcal{A}',P_1'\rangle}{\langle \mathcal{B},\mathcal{A},(P_1\|P_2)\rangle \to \langle \mathcal{B}',\mathcal{A}',(P_1'\|P_2)\rangle}\ \|_1 \qquad \frac{\langle \mathcal{B},\mathcal{A},P_2\rangle \to \langle \mathcal{B}',\mathcal{A}',P_2'\rangle}{\langle \mathcal{B},\mathcal{A},(P_1\|P_2)\rangle \to \langle \mathcal{B}',\mathcal{A}',(P_1\|P_2')\rangle}\ \|_2$$

$$\frac{}{\langle \mathcal{B},\mathcal{A},(nil\|nil)\rangle \to \langle \mathcal{B}',\mathcal{A}',nil\rangle}\ \|_{\mathrm{end}}$$

In detail, the rule $\|_1$ says that if the program P_1 in $P_1\|P_2$ can be evolved to P_1' (i.e. $\langle \mathcal{B},\mathcal{A},P_1\rangle \to \langle \mathcal{B}',\mathcal{A}',P_1'\rangle$), then $P_1\|P_2$ can be evolved to $P_1'\|P_2$ (i.e. $\langle \mathcal{B},\mathcal{A},(P_1\|P_2)\rangle \to \langle \mathcal{B}',\mathcal{A}',(P_1'\|P_2)\rangle$). The rule $\|_2$ can be explained in a similar way. Therefore, both rules $\|_1$ and $\|_2$ enable the agent to evolve as long as one of its concurrent programs can be progressed. The rule $\|_{\mathrm{end}}$ formalises the termination of a concurrent program $P_1\|P_2$ when both P_1 and P_2 are terminating.

Finally, we present the last handful of derivation rules that capture the behaviours of the declarative goal program $goal(\varphi_s,P,\varphi_f)$. Recall that the declarative goal $goal(\varphi_s,P,\varphi_f)$ in CAN agents provides an explicit procedural program P which specifies how the agent might bring about the success condition φ_s, and stops evolving $goal(\varphi_s,P,\varphi_f)$ if the failure condition φ_f turns out to be true. In other words, the failure condition φ_f encodes when it is deemed impossible for the agent to continue executing. Therefore, the first intuitive derivation rule for a declarative goal program $goal(\varphi_s,P,\varphi_f)$ is when either φ_s and φ_f holds true. The following two rules enable the agent to drop the declarative goal program if it becomes achieved (i.e. $\mathcal{B} \models \varphi_s$) or impossible (i.e. $\mathcal{B} \models \varphi_f$) where *?false* is a syntactic sugar to denote a failed program.

$$\frac{\mathcal{B} \models \varphi_s}{\langle \mathcal{B},\mathcal{A},goal(\varphi_s,P,\varphi_f)\rangle \to \langle \mathcal{B},\mathcal{A},nil\rangle}\ G_s \qquad \frac{\mathcal{B} \models \varphi_f}{\langle \mathcal{B},\mathcal{A},goal(\varphi_s,P,\varphi_f)\rangle \to \langle \mathcal{B},\mathcal{A},?false\rangle}\ G_f$$

When a declarative goal program is encountered during execution, and not already true or deemed impossible (i.e. $\mathcal{B} \not\models \varphi_s \vee \varphi_f$), then the agent will typically initialise the execution of a declarative goal program by setting the procedural program P which has no recovery plan available (i.e. $P \neq P_1 \rhd P_2$) in the such a declarative goal to be $P \rhd P$. Formally, the following rule $G_{\texttt{init}}$ formalises the initialisation process:

$$\frac{P \neq P_1 \rhd P_2 \quad \mathcal{B} \not\models \varphi_s \vee \varphi_f}{\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, P \rhd P, \varphi_f) \rangle} \quad G_{\texttt{init}}$$

Such an initialisation process has the following two important advantages. Firstly, it retains the first P to be executed to potentially reach the success condition as given in the original goal program. Secondly, it replicates the original procedural program P and stores itself as an alternative plan for failure recovery when the first program P gets blocked. Therefore, the agent can carry on repeating the execution of procedural program P as long as neither the success condition nor failure condition holds.

To respond to the first advantage of the initialisation, a new derivation rule $G_{\texttt{seq}}$ is defined for performing a single step of the first program on an already initialised program as follows:

$$\frac{P = P_1 \rhd P_2 \quad \mathcal{B} \not\models \varphi_s \vee \varphi_f \quad \langle \mathcal{B}, \mathcal{A}, P_1 \rangle \rightarrow \langle \mathcal{B}', \mathcal{A}', P_1' \rangle}{\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}', \mathcal{A}', goal(\varphi_s, P_1' \rhd P_2, \varphi_f) \rangle} \quad G_{\texttt{seq}}$$

To respond to the second advantage of the initialisation, a new derivation rule $G_{\rhd}$ is defined for situations when the first program which is currently pursued cannot continue further. To handle this situation, the very original procedural program replicated as P_2 in $P_1 \rhd P_2$ will be re-instantiated as the current strategy (i.e. $P_2 \rhd P_2$), in the hope that it could work in the new environment. Formally, the following derivation captures the re-instantiation process when the current procedural program gets stuck:

$$\frac{P = P_1 \rhd P_2 \quad \mathcal{B} \not\models \varphi_s \vee \varphi_f \quad \langle \mathcal{B}, \mathcal{A}, P_1 \rangle \nrightarrow}{\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f) \rangle} \quad G_{\rhd}$$

To be precise, if the current strategy P_1 is blocked (i.e. $\langle \mathcal{B}, \mathcal{A}, P_1 \rangle \nrightarrow$) and P_1 has a backup strategy (i.e. $P = P_1 \rhd P_2$), then P_2 will be re-instantiated for another round of attempt. Of course, this only happens when neither success nor failure condition holds (i.e. $\mathcal{B} \not\models \varphi_s \vee \varphi_f$). Once those conditions above hold, the declarative goal program $goal(\varphi_s, P_1 \rhd P_2, \varphi_f)$ is re-instantiated to be a new declarative goal program, namely $goal(\varphi_s, P_2 \rhd P_2, \varphi_f)$.

2.3.2.2 Agent-level Execution

We now present the semantics of agent-level execution, which sits on the top of intention-level execution, characterising the evolution of an agent. We start with the concept of an agent configuration. Formally, an agent configuration is defined by 5-tuple configuration $\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \rangle$ consisting a plan library Π, an action description library Λ, a belief base $\mathcal{B}$, the sequence of

actions $\mathcal{A}$ that has been executed so far, and the intention base Γ. The intention base by definition is a set of current intentions (i.e. the agent programs $P \in \Gamma$). In general, there are three operations that the agent-level execution needs to perform, namely (i) select an intention and execute a step; (ii) incorporate any pending external events; and (iii) update the set of intentions. In other words, an agent needs to take steps on some active intention, assimilate external events that have appeared (e.g. external requests), and discard the intentions that either completed or failed. In the following, we will present the three kinds of agent-level transitions and their related derivation rules.

The first step in an agent-level execution is to select an intention from the intention base, and evolve it one step by making a legal intention-level transition which is defined in Section 2.3.2.1. The following derivation rule captures the intention selection and execution.

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

The rule $A_{\texttt{step}}$ says that if an intention $P \in \Gamma$ is selected for executing one step becoming P' (i.e. $\langle \mathcal{B}, \mathcal{A}, P\rangle \to \langle \mathcal{B}', \mathcal{A}', P'\rangle$), then the intention base will be updated from Γ to $(\Gamma \setminus \{P\}) \cup \{P'\}$.

The second step in an agent-level execution is to incorporate external events which originate from the environment. For instance, the external event may account for the request from another agent that it must react to. Indeed, an intelligent agent should be able to handle an unexpected event while pursuing its current intentions. To assimilate an external event e, the agent simply needs to add it to its current intention base(i.e. $\Gamma \cup \{!e\}$). Therefore, we have the following derivation rule $A_{\texttt{event}}$ to incorporate a new external event as a new intention:

$$\frac{e \textit{ is a new external event}}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle \to \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \cup \{!e\}\rangle} \quad A_{\texttt{event}}$$

The third step in an agent-level execution takes care of terminating intentions which cannot execute further (i.e. $\langle \mathcal{B}, \mathcal{A}, P\rangle \nrightarrow$). Formally, the following derivation rule $A_{\texttt{update}}$ is defined to remove an intention (i.e. $\Gamma \setminus \{P\}$) that cannot make any further transition:

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, \mathcal{A}, P\rangle \nrightarrow}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma\rangle \to \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma \setminus \{P\}\rangle} \quad A_{\texttt{update}}$$

2.3.3 Scenario

In this section, we demonstrate some of these fundamental concepts of a CAN agent in Section 2.3.1 and Section 2.3.2 via a simplified robot cleaning example.

2.3.3.1 Scenario Description

We begin with conceptually describing the scenario of our simplified robot cleaning example (adapted from the original AgentSpeak work in [Rao96]). Let it be a traffic world where there are four adjacent lanes, namely lane `a`, `b`, `c`, and `d`. The physical layout of traffic world is that the

lane a is situated adjacent to the lane b, lane b adjacent to lane c, and lane c adjacent to lane d. In this traffic world, there is a cleaning robot whose jobs is to keep these four adjacent lanes clean. Such a cleaning robot can perform three basic actions, namely moving to the next adjacent lane, picking up the waste (if any) on the lane it is on, and depositing the waster in the bin. It is assumed that the environment is dynamic and pervaded by uncertainty. Therefore, there is a possibility that the waste can appear on any of the lanes at any time. The following graphic depicts such a traffic world cleaning scenario where the robot is initially on lane a, the waste on lane b, and the bin on lane d shown in Figure 2.1.

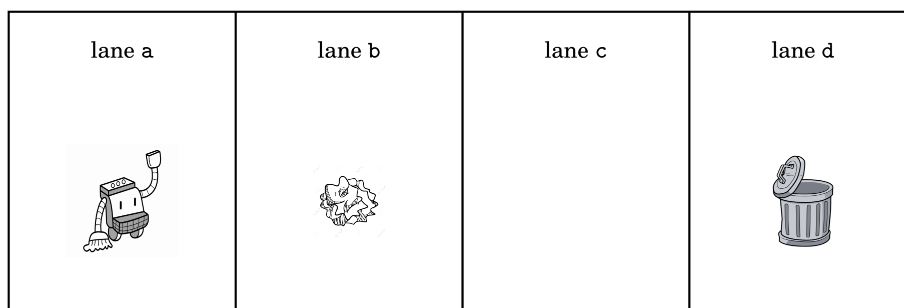

Figure 2.1: Clean Robot in a Traffic-world

In order to keep these four lanes clean, the cleaning robot will have to move around to pick up the waste (if any) and place it in the bin. When there is no waste on any lane, the robot should remain idle to save energy. In other words, it can be concluded that the robot should only start the cleaning operation when there is a waste on at least one lane. Once the robot is initiated to clean the waste due to the appearance of the waste, the robot will need to perform three different type of tasks. In detail, the first task is to move to the lane where the waste is on. If the robot and the waste happen to be on the same lane, no moving action is required. When the robot is situated on the same lane as the waste, the robot needs to perform the picking up action to collect the waste. When the waste is successfully collected, the robot then needs to move to the lane where the bin is and deposit the waste in the bin in the end.

2.3.3.2 Scenario Modelling in CAN Language

We now formally model the cleaning robot scenario in Section 2.3.3.1 in CAN formalism. We first start with the syntax which this scenario requires. Recall that the cleaning robot scenario contains objects, namely four lanes, one robot, waste, and one bin. Therefore, we can simply use these object names to denote their self-explanatory constants. In detail, we have constants such as a, b, c, and d to denote each corresponding lane, and constants robot, bin, and waste for the rest of objects in this traffic world. On top of these constants, we will have a set of predicates to encode the relations between these objects. The first predicate is location(X,Y) which defines that X is at location Y (e.g. location(robot,a)). The second predicate is adjacent(X,Y) which

```
1   // Initial beliefs
2
3   adjacent(a,b)
4   adjacent(b,c)
5   adjacent(c,d)
6   location(robot,a)
7   location(bin,d)
8   location(waste,b)
```

Figure 2.2: BDI Agent Belief Design in the Robot Cleaning Scenario

defines that something is adjacent to something else, e.g. adjacent(a,b). In fact, these two types of predicates are already sufficient for us to encode the physical layout of the traffic-world and the location of the robot and bin shown in Figure 2.1. The set of base beliefs is given in Figure 2.2

We now continue to introduce predicates to build up the plan library of the resulting robot. Firstly, we need a set of action predicates, namely move(Y,Z), drop(X,Y), pick(X), and stop. Intuitively, the action predicate move(Y,Z) defines the action of moving from the lane Y to lane Z whereas the predicate drop(X,Y) says that the robot can drop one object X into another object Y. The predicate pick(X) states the action of picking up the object X while the predicate stop represents an empty action which does nothing. These action predicates, along with the terms, give us a set of actions. For example, the action move(Y,Z) standing for moving from location Y to lcoation Z requires the precondition of location(Y), and results in deletion of base belief of location(Y) and addition of the base belief of location(Z). In this thesis, we assume that the meaning of these actions can be intuitively derived. Therefore, we leave the action library undefined. Secondly, we need suitable predicates for the triggering events in our plan rules. Recall that it is concluded in Section 2.3.3.1 that the presence of the waste initiates the robot. Therefore, we can have a triggering event of the form the belief addition, namely +location(waste,b) showing that there is a waste in location b. Furthermore, it is also mentioned in Section 2.3.3.1 that the robot needs to perform three different types of tasks, namely moving, collecting, and depositing. Therefore, we have the following event goals as triggering events, namely !go(X), !collect(waste), and !deposit(waste,bin). Intuitively, the event goal !go(X) stands for moving to a location X, !collect(waste) collecting the waste, and !deposit(waste,bin) depositing the waste in the bin. Finally, we introduce another belief predicate has(robot,waste) to confirm the robot succeeds in picking up the waste.

We now present the rest of design of our cleaning robot in this traffic world by showing its initial event goal(s), and plan library[1] shown in Figure 2.3. In this scenario, there is no initial goal, e.g. external events, provided by the BDI programmers. The set of plans for this cleaning robot is given on lines 5-10. The plan on line 5 is written to get triggered when some waste appears on a particular location. It instructs the agent to respond to two event goals, namely !collection(waste) and !deposit(waste,bin), in order. The plans on line 6 and 7 are the two

[1]Note that we use & for $\wedge$ and $<-$ for $\leftarrow$ in the actual agent programs.

```
1  // Initial goals
2
3  // Plan library.
4
5  +location(waster,X) : location(bin,Y) <- !collect(waste); !deposit(waste, bin)
6  +!collect(X) : has(robot,X) <- stop
7  +!collect(X) : not has(robot,X) & location(X,Y) <- !go(Y); pick(X)
8  +!go(X) : location(robot,X) <- stop
9  +!go(X) : location(robot,Y) & adjacent(Y,Z) <- move(Y,Z); !go(X)
10 +!deposit(X,Y) : has(robot,X) & location(Y,Z)<- !go(Z); drop(X,Y)
```

Figure 2.3: BDI Agent Plan Library Design in the Robot Cleaning Scenario

strategies to handling the event goal `!collection(waste)`, i.e. collecting the waste if it appears. First, if the robot has already picked up the waste (i.e. `has(robot,X)`), then the robot does nothing apart from the action `stop` according to plan on line 6. Second, if the robot does not have waste in hand and the waste is at another location, then the robot needs to move to that location first and subsequently pick up the waste shown by the plan on line 7. Similarly, the plans on line 8 and 9 specify the different strategies to move to a location based on where the robot is. Finally, the plan on line 10 encodes the strategy of handling the event goal `!deposit(waste,bin)`. It instructs the agent to deposit the waste in the bin by first moving to the location of the bin and then actually dropping the waste in the bin.

2.3.3.3 Scenario Execution in CAN

We now explain how our cleaning robot modelled in CAN agent functions regarding the derivation rules we introduced previously. In particular, we focus on the transition of the intention-level configuration. Suppose next that, at some point, some waste appears on a particular location (shown in red in initial beliefs on line 8) in Figure 2.2. In other words, a triggering event of the form of `+location(waste,b)` is present waiting to be addressed. In this case, the derivation rule `event` introduced in Section 2.3.2.1 can produce the following program containing the set of relevant plans available for handling such an event:

`+location(waste,b): (|location(bin,Y):!collect(waste); !deposit(waste,bin)|)` (1)

In this particular case, we can see there is only one relevant option to repsond to the event `+location(waste,b)`. Since the agent believes `location(bin,d)` to be true, the derivation rule `select` will evolve the set of relevant plans shown in (1) and yeild the following program after applying the unifier `Y/d`:

`!collect(waste); !deposit(waste,bin)` $\triangleright$ `+location(waste,b): (||)` (2)

where `+location(waste,b): (||)` stands for empty alternative strategies available for recovery if the current one failed. The following pictorial illustration depicts the evolution of the program `+location(waste,b)` shown in Figure 2.4. In detail, we can see that e = `+location(waste,b)`

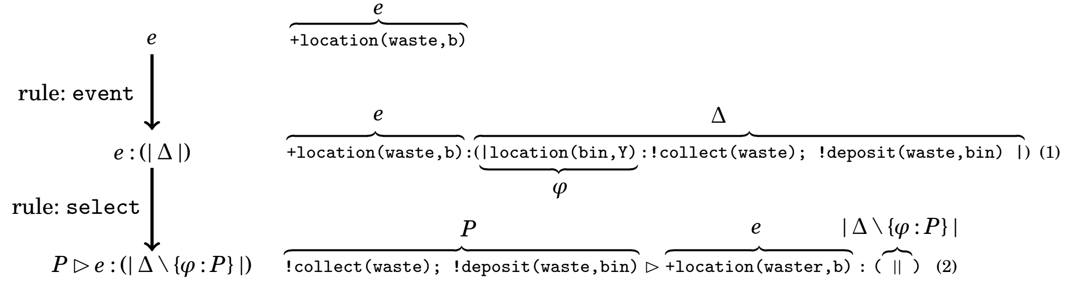

Figure 2.4: Semantic Evolution Flow of the Program +location(waste,b).

is evolved to $e : (| \Delta |)$ by the rule event where the set of relevant plans is denoted as Δ = location(bin,Y):!collect(waste); !deposit(waste,bin). Afterwards the program $e : (| \Delta |)$ is further evolved to be $P \rhd e : (| \Delta \setminus P |)$ through the derivation rule select where P = !collect(waste); !deposit(waste,bin).

The agent may next execute program !collect(waste); !deposit(waste,bin) in which both the first and second step involve resolving the event goal of collecting waste and depositing waste in the bin, respectively. To resolve the event goal, it in turn has to employ first the derivation rule event and then select. Suppose that the agent now uses the derivation rule event to respond to the first event goal !collect(waste). The program above in (2) can evolve to the following program in (3):

P_col; !deposit(waste,bin) ▷ location(waster,b): (||) (3)

where P_col $\stackrel{\text{def}}{=}$collect(waste):(|φ_1:stop,φ_2:!go(Y); pick(X)|), φ_1 = has(robot,waste), and φ_2 = not has(robot,X) ∧ location(waste,Y). The program P_col is actually the evolution of the program !collect(waste) shown in (2). Since the agent believes not has(robot,waste) and location(waste,b) to be true, then the program P_col keeps evolving into the following program shown in (4):

!go(b); pick(waste) ▷ collect(waste): (|has(robot,waste):stop|) (4)

It is noted that the agent is obligated to carry out P_col to full completion (if possible) before it can respond to the second event goal !deposit(waste,bin) in (3). We can see that in order to evolve the program shown in (4), the same process of using event and select derivation rules will be needed to apply again to repsond to the event goal !go(b). Since the agent believes that location(robot,a) and adjacent(a,b) to hold, the event goal !go(b) would evolve to the following program in (5):

move(a,b); !go(b) ▷ go(b): (|location(robot,b):stop|) (5)

Recall that once the agent has selected an applicable strategy, such a strategy needs to pursued to completion whenever possible. As such, the derivation rule seq ensures the current strategy to

be executed one step, namely the action `move(a,b)`. When executing the action `move(a,b)`, the derivation rule `act` updates the belief base and evolve the action `move(a,b)` to `nil`. Meanwhile, it results in the addition of base belief of `location(robot,b)` and the deletion of base belief of `location(robot,a)`. For brevity, we omit the rest of the evolution of remaining programs. However, it is not difficult to see that the set of derivation rules of intention-level transition nicely and succinctly define how to execute an agent step by step.

2.3.4 Other BDI Architectures

In this section, we briefly discuss a few other relevant BDI Agent-oriented Programming (AOP) languages that will be mentioned in this thesis. The first one is called Artificial Autonomous Agents Programming Language (3APL) [HBHM99] and its extended version A Practical Agent Programming Language (2APL) [Das08]. Similar to CAN language, 3APL is also a variant of AgentSpeak. It has been studied and shown in [HBHM98] that the AgentSpeak agent can be replicated by a matching 3APL agents. Therefore, 3APL has at least the same expressive power as AgentSpeak. In addition to being able to simulate AgentSpeak, 3APL also supports some form of failure handling along with standard plan rules, namely failure plan rules. Usually, such failure plan rules in 3APL are constructed to deal with failure and are given a higher priority than those standard plan rules. In fact, both CAN and 3APL are quite similar to each other regarding the features they provide and their formal style. However, unlike the 3APL addressing failure handling via pre-defined failure rule, CAN does it in a more semantic style by backtracking and trying alternative plans (if available). Concretely speaking, the failure handling in 3APL is considered as explicit knowledge provided by agent programmers to the agent whereas the failure handling in CAN is integrated as a part of already built-in reasoning engine free from actual agent programming. In addition, given the merit of declarative goals in CAN [WPHT02], the researches also extends 3APL to have this feature e.g. in [RDM05]. Finally, it is evident that both of CAN and 3APL give a succinct and full account of the operational semantics of a BDI AOP language. Later on, 2APL extends 3APL for implementing multi-agent systems. In achieving so, there are many new programming constructs proposed for a multi-agent setting, e.g. communication actions. Noticeably, in 2APL plan repair rules are applied only to repair failed plans whereas 3APL plan repair rules can be applied to revise any arbitrary plan. Furthermore, 2APL also proposes a new plan constructs to implement a non-interleaving execution of plans.

The second BDI language is Jadex [PBJ13] which is a Java-based BDI AOP language. The objective of Jadex is to allow for intelligent agent construction using sound software engineering foundation. Therefore, it extends Java with programming constructs to implement BDI concepts such as beliefs, goals and plans. Normally, an XML language is used for the specification of beliefs, goals and plan identifier as well as their initial values whereas the plan bodies are realised in the Java language. This Java-based representation in Jadex automatically differs itself from the logic-based representation in CAN and 3APL. Unlike using events to directly trigger the adoption

of plans like in CAN and 3APL, Jadex has explicit goals whose lifecycle consists of option, active, and suspended. Furthermore, Jadex also has the type of maintenance goal: an agent keeps track of the desired state, and will continuously execute appropriate plans to re-establish the maintained state whenever needed. However, Jadex still follows the similar reasoning cycle as CAN and 3APL, i.e. processing events, selecting relevant and applicable plans, and execute them.

Finally, we look at another different BDI framework which is formal logic-based, called X-BDI in [MLVC98]. They propose a logical formalism used to define the models of BDI that has an operational model that supports them. By being an operational model, it means that proof procedures are correct and complete concerning the logical semantics, as well as mechanisms to perform different types of reasoning needed to model agents. To begin with, it assumes that the beliefs of the agent are not always consistent. Therefore, it provides the capacity to minimally revise the agent program to ensure the consistency of the beliefs. Also, it assumes that the set of desires (i.e. a collection of formulas) are not always consistent and not always concurrently achievable as well. Therefore, it creates two intermediate subsets of desires before committing, i.e. reasoning intentions. The first subset of desires consists of desires such that their adoption conditions hold, but the current belief base does not support their final desired conditions. This first subset of desires is also called eligible desires. X-BDI then selects a subset of the eligible desires that are both consistent regarding their final desired conditions, and possible. By being possible, it means that there is a plan that can transform the belief base so that the final desired conditions become true. In X-BDI, such a possibility proof is verified through the logic abduction, which involves generating and applying a set of environment modification actions that results in the entailment of the final desired conditions.

2.4 Planning

In this section, we introduce the First-principles Planning (FPP) – devising a plan of actions to achieve some goals – which is also called classical planning.

2.4.1 Model for FPP

The conceptual model underlying FPP can be described as a 5-ary tuple $\langle S, s_0, S_G, A, f\rangle$ where

- S is a finite and discrete set of states;
- $s_0 \in S$ is the fully known initial state;
- $S_G \subseteq S$ is the non-empty set of goal states;
- $A(s) \subseteq A$ is the set of actions in A that are applicable in a given state $s \in S$; and
- $f(a,s)$ is the deterministic transition function where $f(a,s) = s'$ is the state that follows s after doing an action $a \in A(s)$.

The purpose of planning is to find which actions can be applied to which states in order to achieve the goal state when starting from some given initial state. Therefore, a solution or plan of this model is a sequence of actions $a_1,\dots,a_n$ that generates a state sequence $s_0,s_1,\dots,s_n$ where s_0 is an initial state and s_n a goal state. To be precise, the action a_i is applicable in the state s_{i-1} if $a_i \in A(s_{i-1})$, the state s_i follows state s_{i-1} if $s_i = f(a_i,s_{i-1})$, and s_n is a goal state if $s_n \in S_G$.

2.4.2 Languages for FPP – STRIPS

While the conceptual model for First-principles Planning (FPP) above provides an elegant mathematical formalism of the problem, it would be impossible for a planning problem to include an explicit enumeration of all possible states and state transitions. In what follows we will, for simplicity, stick to Stanford Research Institute Problem Solver (STRIPS) [NF71] formalism whose input includes (i) an initial state, (ii) a goal formula which is built from logic ground atoms, and (iii) a set of operators.

Let a state s be a finite set of ground atoms. An initial state s_0 is a state. A goal formula φ_g is built from the ground atoms using the normal connectives $\{\neg,\land,\lor\}$. Unlike the model in Section 2.4.1 which requires enumerating the set of all goal states, the goal formula φ_g is introduced so that the set of states that entails φ_g is the set of goal state, i.e. $S_G = \{s \mid s \models \varphi_g\}$. An operator has a precondition encoding the conditions under which the operator can be applied, and a post-effect encoding the outcome of applying the operator. An operator o is of the form $\langle pre(o), del(o), add(o)\rangle$ where $pre(o), del(o)$, and $add(o)$ are the precondition, delete-list, and add-list, respectively. The delete-list (resp. add-list) encodes the atoms which will be removed from (resp. added to) the state of the world after the operator has been applied. For convenience, the form of $\langle pre(o), del(o), add(o)\rangle$ for an operator o can be often given as a 2-tuple $\langle pre(o), post(o)\rangle$ in which $post(o) = add(o) \cup \{\neg l \mid l \in del(o)\}$ denotes a set of literals that conjoin the add-list and delete-list through taking the delete-list atoms as negative literals.

Let s_0 be the initial state, φ_g be the goal formula, and O a set of operators, an FPP planning problem is a 3-ary tuple $\langle s_0, \varphi_g, O\rangle$. The task of such a planning problem is to find a sequence of operators from O that reaches one of goal states $s_G \in S_G$ such that $s_G \models \varphi_g$ when executed from the initial state s_0. We now illustrate with an example the notions presented as follows.

Example 2. Suppose there is a cleaning robot in a two-grids world with the initial state s_0 ={`agent(left)`, `dirt(left)`, `dirt(right)`} as shown in Figure 2.5. The goal of this robotic vacuum cleaner is $\varphi_g = \neg$`dirt(left)` $\land\neg$`dirt(right)` $\land$ `agent(left)`. There are four operators available for this robotic cleaner, namely o_1 =`move(left)`, o_2 =`move(right)`, o_3 =`clean(left)`, and o_4 =`clean(right)`. For example, the operator o_1 = `move(left)` has an empty precondition $pre(o_1) = \top$ (i.e. always true), the delete-list $del(o_1) = \emptyset$ (i.e. delete nothing), and the add-list $add(o_1)$ = {`agent(left)`}. Meanwhile, the operator o_3 = `clean(left)` has a precondition $pre(o_1)$ =`agent(left)`, the delete-list $del(o_1)$ = {`dirt(left)`} (i.e. no longer dirty on the left),

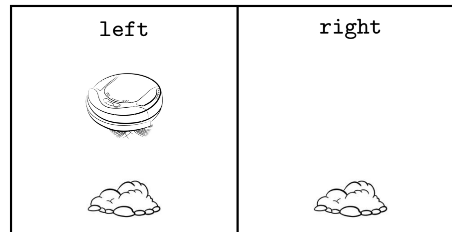

Figure 2.5: Robotic Cleaner in a Two-grid World

and the add-list $add(o_1) = \emptyset$ (i.e. add nothing). Similarly, we can easily give the specification of operator o_2 and o_4 regarding precondition, delete-list , and add-list.

When an operator is executed in a given state, the delete-list atoms will be deleted from the state if they are already in such a state while the add-list atoms will be included in the state if they are not already in such a state. Formally, the effects of applying an operator o to a state s can be described by the transition function defined as follows:

$$f(s,o) = \begin{cases} (s \setminus del(o)) \cup add(o) & if\, s \models pre(o) \\ undefined & otherwise \end{cases}$$

In a similar manner, the effects of applying a sequence of operators to a state can be defined as follows. Let $\langle o_1;\ldots;o_n\rangle$ be a sequence of operators and s_0 be a state. The effects of applying the sequence $\langle o_1;\ldots;o_n\rangle$ to s_0, denoted as $Res(s_0, \langle o_1;\ldots;o_n\rangle)$, is defined inductively as follows:

$$Res(s_0, \langle o_1;\ldots;o_n\rangle) = Res(f(s_0,o_1), \langle o_2;\ldots;o_n\rangle);$$
$$Res(s_0, \langle\rangle) = s_0.$$

Intuitively, $Res(s_0, \langle o_1;\ldots;o_n\rangle)$ specifies that the effects of applying a sequence of operators to a state s_0 is the effects of applying the first operator of the sequence to s_0, namely o_1, to obtain state s_1 (i.e. $s_1 = f(s_0,o_1)$) and so on, until state s_n is obtained by applying the last operator of the sequence to state s_{n-1}. We also call the state s_n the final state. It can be noted that state transitions can easily be computed using set operations, i.e. set addition and deletion.

We now can define what is a solution to an FPP planning problem in STRIPS formalism. Recall that a plan is a sequence of operators that can achieve the goal formula from an initial state. Therefore, a plan for an FPP problem $\langle s_0, \varphi_g, O\rangle$ is a sequence of operators $\langle o_1;\ldots;o_n\rangle$ such that $Res(s_0, \langle o_1;\ldots;o_n\rangle) \models \varphi$, i.e. the preconditions of operators in $\langle o_1;\ldots;o_n\rangle$ are met and the last state makes the goal formula (i.e. the goal state) hold. The following example illustrates what both the transition functions and a plan look like for an FPP problem introduced in Example 2.

Example 3. Recall that the initial state is s_0 ={`agent(left)`, `dirt(left)`, `dirt(right)`} and the goal formula φ_g = ¬`dirt(left)` ∧ ¬`dirt(right)` ∧ `agent(left)` in Example 2. Then, a possible plan for this FPP problem is the following sequence of operators $o_3;o_2;o_4;o_1$ where

o_3 = `clean(left)`, o_2 = `move(right)`, o_4 = `clean(right)`, and o_1 = `move(left)`. Briefly, this sequence of operators instructs the agent to clean the dirt on the left, move to the right, clean the dirt on the right, and move back to the left. To illustrate the transition function, for instance, the result of applying operator o_3 = `clean(left)` to state s_0 ={`agent(left)`, `dirt(left)`, `dirt(right)`} is the state s_1 ={`agent(left)`, ¬`dirt(left)`, `dirt(right)`}.

So far, what we have discussed is known as offline planning which generates a complete plan and then executes in full. An alternative is online planning which differentiates itself from offline planning by not fully elaborating a plan before execution, but instead to interleave planning and execution. Typically, online planning does so by calculating one or more "best" actions, executes these, and then continues another round of online planning from the newly arrived state. Indeed, online can often be more practical and has a broader scope in a highly dynamic and uncertain environment. In particular, online planning can deal with FPP problems whose models are not completely accurate (e.g. due to the actual dynamics of the environment). For example, when the agent expects a state s' after performing an action a in a state s, it is actually a different state s'' observed, provided that the state is fully observable in this case. Unlike the offline planning which may get stuck, e.g. due to the inapplicability of, e.g. actions, in the following action state s'' , the online planning can replan from the s'' instead. One of the special cases, which will be focused in this thesis, is to generate the next best action, execute, and then repeat. Formally, let an FPP problem be $\langle s_0, \varphi_g, O\rangle$. The online planning produces an incomplete plan o_1 towards achieving one of goal states that entails φ_g. If $s_1 \models \varphi_g$ holds (i.e. the goal state is reached) where s_1 is the actual state after executing o_1, then planning stops. Otherwise, it repeats the same process but for the new FPP problem $\langle s_1, \varphi_g, O\rangle$ until the goal state is reached. This particular case of online planning commonly employs an approximate method such as Monte-Carlo Tree Search (MCTS) [BPW+12]. Therefore, it allows not only to return "good enough" actions anytime [KE12], but also to replan when an unexpected situation is encountered while acting efficiently. Further to this, we assume the existing algorithms of online planning but will not discuss these in detail.

2.4.3 Other Planning Formalisms

In this section, we first briefly discuss a few other planning formalisms that will be mentioned in this thesis. Secondly, we will discuss the state-of-the-art of related planning techniques and languages. Note that further in this thesis we use planning as a black box. We communicate with the planning algorithm using a common language. Afterwards, the result of planning is applied in the Belief-Desire-Intention (BDI) framework to improve the capabilities of the agents. The benefit of this approach is that in the future we can adopt any innovations in the planning community by simply swapping the actual planner we use, as long as that planner uses the same common (planning) language.

The planning formalism we describe in Section 2.4.1 is based on a deterministic view of the environment. It assumes that if an action succeeds, it will transition the environment into one particular expected state. However, it is often necessary and beneficial to consider the so-called probabilistic planning. One popular formalism for modelling planning in this setting is the Markov Decision Processes (MDPs). MDPs generates the model underlying FPP in Section 2.4.1 by allowing actions with stochastic effects. In particular, it replaces the deterministic transition function $f(a,s)$ by transition probabilities $P_a(s' \mid s)$ for s' being the next state after doing the action $a \in A(s)$ in the state s where $A(s)$ stands for the set of applicable actions in state s. Effectively, it says that the environment transition between states stochastically, and the probability of transitioning from one state to another depends partially on the current state and partially on the action that the agent executes. The solution for MDPs is a function mapping the states into actions, i.e. take actions based on what state the agent is. These functions are also called policies.

Partially Observable Markov Decision Processes (POMDPs) further generalise MDPs by allowing states to be partially observable through sensors that map the true state of the world into observable tokens according to known probabilities. In one form of MDPs with a specific goal to reach, also called Goal MDPs, its task is to reach the goal with certainty given a known initial belief, actions, and observations that change the world and the beliefs. Unlike MDPs, POMDPs is no longer assumed to be fully known in the initial situation. Also, POMDPs no longer provide full information about the state of the world after executing each action. To get the formalism of Goal POMDPs, there are a few more components to added or modified from Goal MDPs. Firstly, there is a probability distribution b_0 over the states such that $b_0(s_0)$ stands for the probability of s_0 being the true initial state. In general, the probability distributions over states are also called belief states. Secondly, there is a sensor probabilities $P_a(ot \mid s)$ of receiving observation token ot in state s when the last applied action was a. It is required that the probabilities are defined for each state $s \in S$ and action $a \in A$, and that, given a state s and an action a, their sum is 1, i.e. $\sum_{ot \in Ot} P_a(ot \mid s) = 1$ where Ot is a finite set of (o)bservable (t)okens. Finally, it should be stressed that the goal states are assumed to be observable so that there is never uncertainty about whether the goal has been reached or not. The selection of the best action for achieving a goal in POMDPs depends on the observed execution $\langle a_1, o_1, a_2, o_2, \ldots \rangle$. This is because that the last observation no longer summarises the previous execution. However, it is shown that the belief state does. Therefore, the policy of POMDPs is a function that maps belief states to actions.

In the forms of planning considered so far, their focus on bringing about the states of affairs, i.e. the so-called “goal-to-be". Actions are only characterised in terms of their preconditions and effects (or stochastic transition functions). The choice and order for these actions for reaching the goal are computed automatically. However, the Hierarchical Task Networks (HTN) planning (e.g. in [EHN94]) provides an entirely different way of constructing plans. Unlike finding an action sequence that maps the initial situation into a goal state, the objective of HTN planning is to perform some set of tasks. Typically, these tasks are described at several levels. The tasks at

one level can be decomposed into tasks at a lower level until the primitive tasks are reached. These primitive tasks usually stand for real executable actions that do not decompose further. In order to decompose non-primitive tasks, pre-described methods are specifying when these tasks can be decomposed into what sub-tasks in which order. Therefore, we say that a problem of HTN planning is solved if HTN finds a decomposition for the given tasks that results in a consistent network of primitive tasks. The advantage of HTN is that it provides a convenient way to write problem-solving recipes that correspond to how a human domain expert might think about solving a planning problem. In Section 3.2, we can see that the similarities between HTN and BDI and integrations of these two have been intensively studied in BDI community.

We now discuss the state-of-the-art of those planning techniques mentioned in this chapter. To begin with, there are a number of classical planners over benchmarks from the International Planning Competition (IPC)[2], e.g. FF [HN01]. The scalability of planners has improved considerably over the last two decades and is still improving now. Regarding the MDPs and POMDPs, it is usually infeasible to finding a complete optimal policy as the size of the problem grows. In recent years, the focus has shifted to approximate methods such as MCTS, often referred to as online planners. Online planning methods are not aimed at computing partial or complete policies, but at the selection of the action to do next in a planning-and-execution cycle. Recent improvement has led to very competitive online planning algorithms, e.g. UCT in [KS06] for MDPs and PO-UCT in [SV10] for POMDPs. Regarding the HTN planning, due to its expert-led nature, it has been commonly applied in applications with success, e.g. business process management in [GFFOC13], with numerous existing planners.

Finally, along with the progress of planning techniques, the planning community has also long been employing the standardised Planning Domain Definition Language (PDDL). The creation and the adoption of this common language have fostered significant reuse of research, allowed more direct comparison of systems and approaches, and therefore has been supporting faster progress in the language. The main feature of PDDL is that it separates the model of the planning problem in two major parts: (i) domain description and (ii) the related problem description. Intuitively, the domain description is intended to express the general knowledge of a domain. For example, it usually includes, e.g. what predicates there are, what operators are available, and what the effects of actions are. The STRIPS formalism is typically used for presentations of, e.g. predicates and operators. Therefore, PDDL can be, to some extent, regarded as the extension of STRIPS. The problem description, however, presents a specific planning problem. It usually gives the initial state of the planning environment, i.e. a conjunction of true and false facts, and the goal formula, i.e. a logical expression over facts that should be true or false in a state of the planning environment. Ever since the creation of PDDL, it has been continuously extended to support many other advanced features. To name a few, PDDL2.1 introduced numeric fluents to model non-binary resources such as fuel-level whereas PDDL2.2 provided timed initial

[2]http://www.icaps-conference.org/index.php/Main/Competitions

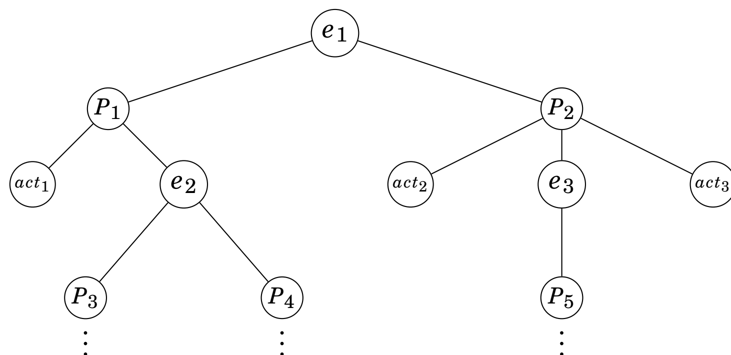

Figure 2.6: Examples of Hierarchical Plan Library Structure

literals to model exogenous events occurring at given time independently from plan-execution. The recent probabilistic track in IPC has adopted the language Relational Dynamic Diagram Language (RDDL), which allows an efficient description of MDPs and POMDPs by representing everything, e.g. observations and actions with variables. Finally, PDDL, as a common language in the planning community, confirms our view of applying planning as a black box. As long as the planners use the same common language, any improvement made to these planners will immediately be applied to our approaches which utilise them in this thesis.

2.5 Graph Theory: Fundamentals

Finally, we introduce some fundamentals in graph theory. It turned out that the graph provides an excellent vehicle to formalise the hierarchical structure manifested in the plan-library in BDI agents. In detail, each plan in BDI agents is composed of steps which include such as actions or event goals. The event goal can be addressed by a set of relevant plans, thus giving rise to a top-down decomposing structure. Figure 2.6 shows a simple hierarchy in the plan library. An event e_1 can be achieved by either of the two plans P_1 or P_2. The plan P_1 involves performing the action act_1 and handling the event goal e_2 whereas the plan P_1 consists of executing act_2, addressing the event goal e_3, and executing act_3. Therefore, it gives a natural hierarchy which is also directed in the plan library. In the following, we succinctly describe the AND/OR graph.

A directed graph is a tuple (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of directed edges. A multigraph is a tuple (N, L, E') where L is a set of labels and $E' \subseteq N \times L \times N$ is a set of multiedges such that for each $l \in L$ we have that $(N, \{(n, n') \mid (n, l, n') \in E'\})$ is a graph. We say n' is a child of n, written as $n' \in child(n)$ iff $(n, l, n') \in E'$ for some $l \in L$. Given nodes $n_1, n_{m+1} \in N$ in a multigraph, then a sequence of nodes and labels $(n_1, l_1, \ldots, n_m, l_m, n_{m+1})$ is a path from n_1 to n_{m+1} iff each n_j is unique and $(n_j, l_j, n_{j+1}) \in E'$ for $j = 1, \ldots, m$. We denote the length of a path of $(n_1, l_1, \cdots, n_m, l_m, n_{m+1})$ to be m. A multi-graph is acyclic if, for each $n \in N$, there exists no path from n to itself. A rooted multigraph is a tuple $(N, L, E', \bar{n})$ where (N, L, E') is a multigraph and

$\bar{n} \in N$ is a root node such that for each $n' \in N \setminus \{\bar{n}\}$, there exists a path from $\bar{n}$ to n'. We also say that a node n is a branch point if n has more than one child, while a leaf node is a node n without child nodes. An AND/OR graph $(N_\vee \cup N_\wedge, L_\vee \cup L_\wedge, E_\vee \cup E_\wedge, \bar{n})$ is a rooted acyclic multigraph where $N_\vee$ (resp. $N_\wedge$) is a set of OR-nodes (resp. AND-nodes), $L_\vee$ (resp. $L_\wedge$) is a set of OR-labels (resp. AND-labels), $E_\vee \subseteq N_\vee \times L_\vee \times N_\wedge$ (resp. $E_\wedge \subseteq N_\wedge \times L_\wedge \times N_\vee$) is a set of OR-edges (resp. AND-edges), and $\bar{n} \in$ is a root node. Intuitively, an AND-node is a solution if each of its child nodes is a solution, while an OR-node is a solution if it is a primitive solution, or at least one of its child nodes is a solution. In the context of BDI agents, we can see that the plan nodes are naturally AND-nodes as each of its children needs to be executed. The event goals are OR-nodes because it needs to select one of its child nodes to handle it. Finally, the actions are trivially OR-nodes as well because they are primitive solutions to execute. In Chapter 6, the formal AND/OR graph representation will be given to formalising the structure of the plan library in BDI agents.

Chapter 3

Literature Review

In this chapter we aim to provide a comprehensive overview of the existing approaches to improve the related features of Belief-Desire-Intention (BDI) agents, namely the robust program execution, adaptive plan library, and efficient intention progression. So we begin by discussing the recent works on including planning in BDI to ensure the robust program execution by (i) synthesising a new plan and (ii) applying lookahead planning on existing BDI plans. When appropriate, we will also discuss the approaches where they investigate the reuse of plan from planning when similar goals need to be achieved. With regard to efficient intention progression, we survey a much broader scope of relevant works on extending the basic capabilities of BDI agents using various techniques (e.g. optimisation) to support different aspects of agent reasoning, e.g. plan selection (which plan to select) and intention selection (which intention to progress).

3.1 Planning to Generate New BDI Plans

We have previously discussed the mechanism of BDI agents in Section 2.3.2. Often, practical BDI agents have avoided the use of First-principles Planning (FPP) in favour of a pre-defined plan library to limit the computational complexity. This allows for fast agent reasoning by relying on pre-defined recipes rather than on planning from scratch. However, it limits the autonomy and robustness of the resulting agent by preventing it from reasoning about alternative courses of action for the achievement of its design objectives. In particular, it causes difficulties in case of the execution failure of the agent programs. For example, when an agent selects a plan to achieve a given goal, it is possible that the selected plan may fail, e.g. the precondition of action in a plan no longer holds before being executed. In these cases, the agent typically will conclude that the goal has also failed. However, there may be potentially other plans that can successfully achieve

the same goal. Therefore, it is arguably more desirable for an intelligent agent to try alternative plans first than directly jumping to a conclusion of the failure of a goal.

Fortunately, to mitigate this problem, some works on BDI agents (e.g. Conceptual Agent Notation (CAN) in [WPHT02]) have already taken some preliminary steps. For example, when a plan failed to achieve a given goal, the failure recovery mechanism would try another applicable plan (if any) to achieve such a given goal (discussed in Section 2.3.1). If no alternative plans are available, then the failure is backtracked to higher-level goals. Such a form of failure handling is usually implemented in a backtracking manner. Alternatively, some other BDI agent (e.g. Artificial Autonomous Agents Programming Language (3APL) in [HBHM99]) possesses a set of plan repair rules to allows plans to be repaired to handle the potential failure. While these meta-level failure handlings can alleviate some of the limitations, the agent can still fail to achieve a goal if either all plans fail, or the failure falls out of the set of plan repair rules. Therefore, to address these shortcomings of BDI agents, a number of works have been done to include the planning capacity to synthesis a new plan to still achieve a goal. In the rest of this subsection, we give a comprehensive survey of these works. For the purpose of legibility, we group these work in clusters based on the type of planning models, namely a deterministic model of planning (i.e. Stanford Research Institute Problem Solver (STRIPS)-like planning), and a model of stochastic state transition (i.e. probabilistic planning).

3.1.1 STRIPS Planning

We now start with discussing the work which includes a STRIPS-like planning capacity, i.e. a deterministic planning system and environment. One of the very first pieces of work looking at FPP in a BDI agent is the work of the Propice-plan framework [DI99]. It is the combination of the IPP planner [KNHD97] and an extended version of the Procedural Reasoning System (PRS) agent system [IGR92] in which each plan is also given an expected declarative effect. The PRS agent is essentially the AgentSpeak without the operational semantics formalisation. In the framework of Propice-plan, an execution module modelled in PRS paradigm takes care of selecting plans from the plan library and executing them. If there is no applicable plan found, a plan module uses the IPP planner to obtain a new plan at run time. To formulate plans, IPP planner uses the plans of PRS agents as planning operators whose preconditions (reps. post-effects) are the contexts (reps. the declarative effects) of the corresponding plans. The goal state of such an FPP problem is the (programmer supplied) declarative effect of the achievement goal that failed due to no applicable plan available. When a solution is found by IPP, it will be returned to the execution module, which executes them by mapping these operators back into ground plans. Normally, we also called this type of plan returned by IPP to be an abstract plan as the operators of such plans are not primitive actions, e.g. STRIPS operators. Indeed, these abstract plans can only be executed using the existing procedural domain knowledge in BDI agents. Finally, we note that what this approach essentially does is to re-arrange existing plans. Thus, it is limited to the

amounts of planning problems it can address, and is not possible to come up with new ideas (i.e. new individual plans).

There is another similar piece of work [SSP09] which also seeks to obtain abstract plans in the form of hybrid plans using FPP in BDI agents. In contrast to the abstract plan which solely consists of operators corresponding to plans in [DI99], the work of [SSP09] produces plans made up of both abstract operators and primitive operators (i.e. primitive actions). Therefore, it not only reuses the existing procedural domain knowledge to find new plans other than those specified by the programmers, but also conforms to it, i.e. respecting the user-intent principles [KMS98]. In this work, the abstract operators are corresponding to the event goals in BDI agents. To transform an event goal into a planning operator, the authors first obtain the precondition of the corresponding operator by simply taking the disjunction of the context conditions of plans associated with such an event goal. However, it is not straightforward to obtain the post-effects of an abstract operator for an achievement goal. Therefore, it adopts a summarisation algorithm of [CDB07] to compute the definite effects of an event goal as its post-effects, given the structure of this goal and its relevant plans, conjoined with the post-effects of related primitive actions. When wishing to obtain a hybrid plan, FPP is given the information of the desired goal state, the initial state, and abstract operators along with primitive actions under the assumption that the proper integration of FPP and BDI agents is already done. Furthermore, the planning is solved exclusively in an offline fashion. Also, because the abstract operators do not encode the possible effects of an achievement goal, it is possible that when mapping back and executing, such possible effects can block the goals (or operators in the hybrid plan). Therefore, the authors also proposed a validation step to check the correctness of the plan obtained to ensure that a successful decomposition is possible.

To further improve the ability of an agent when achieving its goals, the work of [ML07] extended AgentSpeak that allows an agent to explicitly specify the state of the world that should be achieved by the agent. In order to transform the state of the world to meet the desired state, the agent uses FPP to form high-level plans through the composition of pre-defined plans already present in its plan library. This FPP planner is invoked by the agent through a regular AgentSpeak action in its existing language. The BDI agent may include this new planning action at any point within a standard AgentSpeak plan to call a planner. In detail, a planner invocation can be written as a standard AgentSpeak plan: +!`goal_conj`([$b_1,\ldots,b_n$]);`true` ← `plan`([$b_1,\ldots,b_n$]) where `goal_conj`([$b_1,\ldots,b_n$]) is a conjunction of base beliefs which must be made true in the environment and `plan`([$b_1,\ldots,b_n$]) the planning action for such a goal state. Such a planning invocation plan tells that the execution of the planner component can always be successfully triggered by an event +!`goal_conj`([$b_1,\ldots,b_n$]) and the planning action is bound to an implementation of a planner. Similar to work [DI99], it also adopts the approach of converting the plans of AgentSpeak into operators and takes the context of a plan as the precondition of the corresponding operator. However, unlike the work of [DI99] employing the pre-supplied

declarative effects of a plan as the post-effects of the corresponding operator, the work of [ML07] can only convert the plan whose body consists of only belief additions or belief deletions. Also, as the planning part creates new plans, the authors propose a plan reuse strategy [ML08] to reuse the new plans generated by planning from the work [ML07] by assigning a proper context to new plans from the planner. In the end, they settle down on a minimum context condition which they claim neither too restrictive, nor too general. Such a minimum context specifies the preconditions of the first operators, plus the preconditions of any subsequent operators that are not included in the effects of previous operators.

Finally, there are another noticeable work of [MZM04], called X^2-BDI, integrating planning not only to generate new plans for BDI agents, but also verify the possibility of potential desires before committing to them. Unlike the previous works we have discussed above, it differs itself due to the version of BDI framework it is based on, namely X-BDI [MLVC98]. Recall that an X-BDI agent has the traditional components of a BDI agent, i.e. a set of beliefs, desires, and intentions. However, unlike most of the BDI agent architectures (e.g. AgentSpeak), X-BDI agents do not include a library of pre-defined plans. In addition, every desire in an X-BDI agent is a goal conditioned to a body of a logic rule. And the body of such a logic rule specifies the preconditions that must hold in order for an agent to desire a goal. To select desires before committing them to intentions, X-BDI agents first select a set of eligible desires whose preconditions hold and whose goals do not hold yet. The second step is to select further a subset of the eligible desires, which is called the candidate desires that are also possible. By being possible for a set of desires, it means that there is a plan (consisting of primitive actions) that transforms the set of beliefs so that the desired goals become true. However, this type of desire selection suffers from significant inefficiencies. Therefore, instead of the slow logical abduction used in X-BDI for verifying the possibility of desires, the work of [MZM04] improves on X-BDI with a STRIPS planner based on Graphplan [BF97]. To do so, they provide a modified X-BDI along with a mapping from BDI mental states to propositional planning problems and from propositional plans back to mental states.

3.1.2 Probabilistic Planning

In Section 3.1.1, the planning formalisms employed in BDI agents hold a deterministic view of the environment. Such nature of determinism assumes that if an action succeeds, it will transition the environment into one particular expected state. However, there often exists an explicit model of probabilistic state transition in physical applications. For instance, when a dice is thrown, it is obvious that there is an equal chance of getting any number from 1 to 6. Therefore, it is also necessary for an agent to consider the effect of actions in the world state stochastically. However, the BDI model is not natively based on a stochastic description of the environment. In order to incorporate the type of probabilistic planning in BDI agents, therefore, it not only requires a careful examination of difference and similarities between these two, but also needs significant

work on modelling and reasoning uncertainty in the BDI paradigms beforehand.

Fortunately, to alleviate some of these issues, some promising works have been proposed in recent years. One of the first works is the work of [SWP02] which examines the relationship between BDI framework and one probabilistic planning technique, namely Partially Observable Markov Decision Processes (POMDPs), to conjoin the theoretical rigor of the POMDPs and the practical utility of BDI frameworks. To achieve such an objective, their focus is to verify the existence of mapping between BDI models and the counterpart POMDPs. While some of the components of POMDPs and BDI are trivially equivalent (e.g. states and actions), some mappings between components are somewhat convoluted. For the trivial equivalence, it is assumed that the BDI agents operate in an environment whose state transition function is explicitly known to build the state transition correspondence between these two. The key mapping is one between the desire and intention on the BDI side, and the reward on the POMDPs side. To establish such a mapping, they relate desires to rewards, and intentions to a combination of rewards and actions. In detail, the authors assume the desires to be a set of states on the BDI side, and distil the rewards on the POMDPs side in a way that they are defined only over states to build correspondence between rewards and desires. Meanwhile, they identify the concept of intentions in BDI framework with the rewards and actions in POMDPs. In detail, an intention is a stack of partially instantiated plans, which specify a sequence of actions to fulfil some desire of the agent. Therefore, there are both the action and desire aspects to intentions on the BDI side, which correspond to the actions and the rewards on the POMDPs side. Finally, using these equivalence, they provide preliminary empirical evidence that there is a trade-off between the optimality from a POMDPs problem versus the practicality by the domain knowledge encoded in BDI systems.

Later on, the work of [SP06] extends the work of [SWP02] by providing both theoretical and related algorithmic mapping between Markov Decision Processes (MDPs) and BDI framework. Similar to the work of [SWP02], they also assume extra information for BDI agents, e.g. the transition function is known. Unlike the work of [SWP02], their focus is to show how to map intentions in BDI architectures to policies in an MDPs and vice-versa, provided they work on the same state-space. To do so, they use the term “intention” to denote a state that an agent has committed to bringing about, and use the term “intention plan” or “ i-plan” to denote a sequence of actions built to reach a specific intention. Such i-plans are employed to correspond to a subset of pre-defined plans which consists of only actions. Regarding the conversion from the MDPs to BDI, the authors provide the pseudocode which maps policies into intentions. Intuitively, such a conversion collects all finite paths of a MDPs policy from a starting state to an ending state in general cases. The ending state is used as the head of a plan in BDI, whereas the starting state for this path is the context condition of the plan. And the sequence of actions in each path creates a body for such a plan. Converting a BDI agent to an MDPs, however, uses a set of i-plans to assign rewards to states in order to provide a policy for the underlying MDPs that will mimic the behaviour of an agent with the given i-plans. For an individual i-plan, it assigns a value to each

state-action pair related to this i-plan in such a way it reflects the gradient of increasing reward toward completion in each i-plan. Converting an entire plan library involves iterating over all plans. Finally, once the reward function is obtained, the resulting MDPs can be solved using the existing algorithm.

While both of the work of [SWP02] and [SP06] have successfully made progress of providing the useful insight of the relations between BDI and probabilistic planning, namely MDPs, the problem of actual integration of these two remains unaddressed. To this end, the authors of [BMH+16] proposes a pragmatic approach, called AgentSpeak$^+$, which integrates the probabilistic planning into the classical AgentSpeak agents. In this framework, they introduce the concept of epistemic states [ML11], which contains both the current uncertainty information and POMDPs. In particular, POMDPs embedded into such epistemic states is used to represent the domain knowledge about the partially observable environment and the uncertain effects of its actions. In order to call the POMDPs on-demand, a new action in AgentSpeak, namely `ProbPlan`, is introduced to be used in normal AgentSpeak plans to explicitly compute the optimal action to achieve a goal for a given epistemic state. Thus, it enables the agent to resorts to probabilistic planning to deal with the crucial part of its execution when needed, e.g. when the stake is high. Notably, this work also follows the idea of the hybrid plan in, e.g. [SSP09] by allowing POMDPs to contain both primitive actions and abstract actions (i.e. the summarisation of goals in the plan library). Finally, a prototype implementation of this framework is also developed that extends Jason [BHW07], which is an open-source implementation of AgentSepak agents.

Following the line of tight integration of POMDPs and BDI agents, the work [RM17] is proposed to combine the advantage of the online generation of reward-maximising courses of action from POMDPs and the sophisticated means-end reasoning (e.g. multiple-goal management) from BDI side. In detail, their key contributions are twofold. Firstly, they introduce the notion of the intensity of the desire for the achievement of goals, which is a mapping from goals to numbers representing the level of desire to achieve the goals. Hence, the agent can maintain desire levels when pursuing multiple goals and when new goals come up. Secondly, it allows the plan library in BDI part to store recently generated plans and reuse these plans if needed (similar to the plan reuse strategy in the work of [ML08] in STRIPS planning). Therefore, the agent can take advantage of the past "experience" – saving time and computation. In their experiments, it is also shown that the agent could perform actions up to 1.7 times faster (when executing only the first action of a policy) with an equivalent performance by reusing policies.

Finally, there is a different piece of work [KBM+16], which integrates online planning in MDPs with BDI agents to handle risk when selecting rational actions to achieve its goals. In particular, they allow the agent programmers to design agents that can set their risk aversion levels dynamically based on their changing beliefs about the environment. Their motivation starts with the recognition that pursuing high utility can often entail high potential costs. Therefore, it is vital for the agent to balance the trade-off between maximising expected utility (increasing

utility) and minimising potential costs (lowering risk). To do so, the authors first provide a novel method for calculating risk alongside utility in online planning algorithms. Secondly, they provide the decision strategy, guided by a set of principles, to a BDI agent about how to decide between multiple actions given both utility and risk assessments from an online planner. Finally, they extend the standard agent configuration to include our risk aversion value and introduce the derivation rule to change risk aversion value based on beliefs dynamically. Furthermore, their evaluation demonstrates that raising the risk aversion level of an agent will indeed cause it to take less risky actions. As such, it has a higher probability of successfully reaching its goal.

3.2 Applying Lookahead Planning to Existing BDI Plans

In Section 3.1, we have discussed different approaches which integrate various planning techniques to generate alternative courses of actions for the achievement of goals in BDI agents. We now look at another usage of planning, i.e. the so-called lookahead capacity, in BDI agents to reason about the consequences of choosing one existing plan over another for solving a given goal. Indeed, the ability to look ahead to guide choices in BDI agents is clearly desirable or even mandatory to ensure the successful achievement of goals in some situations. For instance, steps in a plan may not be reversible. It means that the wrong choice of a plan not only consumes important resource, but also may lead to situations from which the goal can no longer be solved. Therefore, by reasoning about the consequences of choosing one plan over another, the agent can guide its execution to avoid detrimental and troublesome situations.

Recall that in Section 3.1.1, we have discussed that the Propice-plan framework [DI99] provides the planning component to generate new plans when there is no applicable plan available in the plan library. As a matter of fact, the framework of Propice-plan also provides some lookahead capabilities to simulate and examine in advance a number of possible options available to the system ahead of execution. In detail, the lookahead capacity in Propice-plan can (i) advise the execution of the best option concerning the current state of the world, and (ii) anticipate some unsatisfied preconditions to come, and try to establish them with an adequate opportunistic strategy. The simulation is performed through the hierarchical expansion of BDI plans, guided by subgoals within plan bodies. Such an expansion is done in a similar way that how the initial state of the world is updated as methods are refined in Hierarchical Task Networks (HTN) planning. When the lookahead capacity failed the execution simulation of a goal due to some missing preconditions before the execution reaches this point, this lookahead component can also initiate an insertion of a proper instantiated plans to establish the missing preconditions.

It did not take long for the researchers in BDI community to recognise the many similarities between BDI programming languages and HTN planning. The very first work of explicitly incorporating HTN in BDI agents is the framework called the Cypress system [WMLW95]. In the Cypress system, the SIPE-2 [Wil90] HTN planning system is augmented with an extended

version of the PRS agent. This loosely coupled integration of SIPE-2 and PRS is combined with a new common representation language called ACT. This ACT language contains both the planning operators of SIPE-2 and the goal-directed reactive procedural rules of PRS agents. Cypress also includes translators that can automatically map ACT onto SIPE-2 and PRS structures, along with a translator that can map SIPE-2 operators and plans into ACT. The programmers specify the domain in the ACT language by default. The domain in ACT language can at runtime be translated into the language of PRS and SIPE-2 whenever needed. The Cypress system operates by employing the SIPE-2 to perform the lookahead function on PRS events to a suitable level of abstraction, which is domain-specific and given by the programmers. When SIPE-2 returns such an abstract plan, the PRS execution reasoning fills in the remaining details via the standard BDI decomposition. Therefore, it is argued by the authors that the solution obtained from SIPE-2 can be flexible as they consist of abstract entities whose exact refinements are dealt with by BDI agents.

While the Cypress framework acknowledges complementary advantages and close similarities of HTN planning and BDI paradigm, the contrasts and comparison of these two are not conducted until in the work of [SP04]. This work formally provides a mapping between HTN-based planning and BDI reasoning. In detail, they formalise that both BDI and HTN systems share a similar notion of decomposition and flexible composition of parts. The difference of these two, however, is that the decomposition in BDI is essentially employed for selecting goal-directed actions in a dynamic environment while the decomposition in HTN systems is to search a legit plan (which can be successfully executed later). Regarding the mapping, overall, the goal-plan hierarchy in BDI corresponds to a task network in HTN. In detail, the event goals of BDI are mapped to the abstract tasks of HTN, whereas the plans of BDI to methods in HTN. The hierarchy decomposition in BDI (resp. HTN) begins by having an event goal (resp. an abstract task) to be achieved by (reps. decomposed into) plans (resp. methods). The plan (resp. method) may have a sub-goal (resp. an abstract task) to achieve (resp. decompose). Therefore, there is a tree-like hierarchical structure formed in both two systems.

Thanks to the systematic similarity study provided in the work of [SP04], the work of [SP05a] and [SP05b] quickly follow up and integrate HTN planning in BDI agents. They propose a framework where BDI agent could use HTN planning in an environment when lookahead analysis is necessary to provide guaranteed solutions. In particular, BDI agents could use HTN lookahead to anticipate and avoid branches in the BDI hierarchy that would prevent the agent from achieving a goal. Unlike the Cypress system in [WMLW95], however, these two works focus on decomposing a goal entirely up to the level of primitive actions. Thus, the agent can be ensured whether a goal can have at least one successful decomposition in this way. In addition, while the Cypress treats the HTN and BDI as equal components under a common language ACT, these two works instead embed HTN in BDI systems. This implies that BDI agent is in control of HTN planning. Finally, to ensure a tight coupling between HTN and BDI, the invoking point of HTN

planning is written in the body of a standard BDI plan, providing the flexibility to the agent. Also, the program to be run by the HTN is derived from the existing BDI programs (i.e. the limited subsets of the program) to minimise the programming overhead. And the execution of the plan returned by HTN is done using the regular BDI execution following the advice from the planner on what plans to choose.

After the works of [SP05a] and [SP05b], there is also some progress made to provide a formal semantics of integration of HTN in BDI framework. The works of [SSP06] and [SP11], called CANPLAN, introduce a new language `Plan(P)` to CAN semantics, so that `Plan(P)`, where *P* is a plan-body program, is intended to mean "plan for *P* offline, searching for a complete hierarchical decomposition". In this way, the construct `Plan` in an CAN agent is bound to hierarchical lookahead planning on how to expand a plan to completion. By looking ahead rather than simply selecting the first applicable pre-defined plan, potential troublesome execution sequences could be avoided. The authors of [SSP06] and [SP11] also formally establish the equivalence between the `Plan` construct and HTN planning. Therefore, it is proved that the new construct `Plan` can indeed serve as an HTN planner in CANPLAN. Finally, there are two noticable complementary works [BLH+14] and [Sil17] which extend the CANPLAN framework. The work of [Sil17] develops a formal account of converting HTN hierarchies to obtain BDI goal-plan hierarchies. Its spirit is in line of converting MDPs policies into BDI plans studied in the work of [SP06]. Therefore, the plan library of a BDI agent can be enlarged from HTN domains. Meanwhile, the work of [BLH+14] extends the semantics of CANPLAN with actions that have probabilistic effects. However, while the semantics is sound in [BLH+14], the authors admit that such probabilistic HTN planning is hard to implement.

Finally, the work of [WBPL06] also attempts to merge BDI agents in Jadex framework and customised HTN-like planning to lookahead for plans along with a proof of correctness. Like in most BDI agents, goals in [WBPL06] are also represented as specific world states that the agent is trying to pursue. However, it significantly deviates from most traditional BDI agents in that the desires of an agent are represented domain-specific inverse utility functions that can guide the planning process. For example, the desire of an agent can be to keep the number of moved blocks low in a blocks-world domain. Also, each goal is assigned a unique function that reveals an approximate distance from the state to the goals. When given the representation of the world states, the HTN-like planner in this work takes into account the current goals of the agents and the functions specified by the desires to refine goals into actions. Planning then consists of decomposing the stack of goals into executable actions while trying to maximise the expected utility of the resulting plan using a heuristic based on the distance from the current state to the goals and the expected utility of these goals. Therefore, the emphasis of this work is on performance and the better utilisation of domain knowledge in BDI agents when integrating HTN planning.

3.3 Plan Selection

In the previous sections, we have mentioned numerous approaches to integrating planning in BDI agents for different purposes. In this section, we look at the question of what plan should be selected to achieve a given goal, i.e. plan selection. Recall that an essential feature of BDI agents is that it has a number of different means through which it can achieve a given goal. Often, the choice of means to achieve a goal is made by selecting pre-defined plans at run time based on the triggering event and the current beliefs of the agent. While this feature is handy, it remains unclear regarding which plan to adopt if several are applicable. In fact, it is often true that different means are likely to have different characteristics (e.g. cost and preference). Therefore, there is an intuitive question to ask about how to select the "most appropriate" applicable plan given the situation. Of course, the definition of being most appropriate is often domain-specific. Nevertheless, most current BDI languages typically lack the basic underlying representations for costs, preferences, time, and etc., which are necessary to implement such capabilities. In this section, we discuss the recent efforts made to build up these representations in BDI frameworks and how the relevant plan selection strategies work. Before we survey a large body of concrete plan selection mechanisms in BDI agents in recent years from different aspects, we notice that there are some aspects of these plan selection capabilities which can be already programmed in the current BDI agent languages. Most platforms provide some forms of hooks that allow the agent developers to control which plan is adopted. For example, the plan selection function S_O in [BHW07] is a user-defined function to customise plan selection for a particular application domain.

3.3.1 Meta-level Reasoning

In this section, we first look at works which approach the problem of plan selection via meta-level reasoning within the given BDI language itself. One of the first works is the work of [HBHM99] based on 3APL agents, which addresses the problem of plan selection via a meta-level structure. To do so, they first distinguish between an object-level which concerns the programming of agents in the agent language 3APL, and a meta-level which concerns the programming of control structures for agents. And the meta-level language is defined using transition systems of the agent in the object level. As such, this approach not only gives the advantages associated with the modularity of two different systems, but also allows for more freedom in specifying the various selection mechanisms for the agent. In detail, the control structure for the plan selection mechanism is based on an intuitive classification and order imposed on these rules. For example, the failure rule can be selected preferably to recover the failure instead of seeking optimisation. Later on, the work [DDBDM03] extends the work of [HBHM99] by breaking down the control structures for agents into individual programming constructs that can be used to select and apply plans. Therefore, a customised agent deliberation cycle can be programmed in terms of

these constructs. For example, the meta-statement $selrule(t_{sg}, t_{sr}, V_{ig}, V_{ir})$ says selecting a rule and a goal from the set of rules t_{sr} and the set of goals t_{sg}, respectively. Of course, the selected rule should be applicable to the selected goal, and they are assigned to variables V_{ig} and V_{ir}, respectively.

There is also another work of [Win05] which presents a meta-interpreter for the AgentSpeak language to provide easily prototyping extensions or changes to AgentSpeak language itself. By being a meta-interpreter, it implies an extra layer of interpretation over the underlying interpreter to extending the language or adding functionality. As such, by having the meta-interpreter to make the selection of plans explicit, the authors believe that they can override the provided defaults regardless of whether the implementation provides for this. However, this work tackles a plan selection problem in a slightly unconventional point of view, namely multiple solutions for context condition. In detail, the multiple solutions for context condition means that given a plan in the form of $e : \varphi \leftarrow P$, there may be two different ways of satisfying context condition c which gives different substitution θ_1 and θ_2 in given the current beliefs of the agent. Their plan selection supports the inclusion of multiple instances corresponding to different substitution to the same plan as applicable. However, it still remains silent regarding how to explicitly select one plan from this expanded set of applicable plans.

There is also a recent work of [LL15] which controls which relevant applicable plan to intend through the procedural reflection in the agent programming language meta-APL [DYAL14]. Similar to the works of [HBHM99] and [Win05], it also allows both the agent programs and the deliberation strategy of the agent to be encoded in the same programming language. Therefore, an agent programmer can not only write standard agent programs, but also customise the deliberation cycle to control which relevant applicable plans to select by exploiting procedural reflection. To do so, they introduce the so-called object rules to select an appropriate plan based on a reason. The syntax of an object rule is in the form of `reasons [:context]:` P where both `reasons` and `context` are beliefs, and P is a plan. Therefore, selecting a plan requires not only the context condition to be true, but also the relevant reasons to hold. However, it remains unaddressed regarding the sophisticated approaches to selection of plans based on the characteristics of plans, e.g. preference and cost. In the following section, we will discuss the recent works which handle the plan selection in precedence-based reasoning regarding the relevant characteristics of plans.

3.3.2 Precedence-based Reasoning

In this section, we have a look at works which select plans based on the precedence of specific plan characteristics. The work of [DW08] answers the question of how to select between different applicable plan instances from the aspect of maintenance in BDI agent-oriented software engineering. In a nutshell, their work is motivated to maintain the consistencies of a software system. Indeed, the system may find inconsistencies in the model when the software is modified due to a

range of causes, e.g. adding new functionality. For example, when an agent type is added, then consequently, other agents may need to be modified to communicate with this new agent. To avoid these inconsistencies, therefore, some secondary changes (also called change propagations) are needed to meet a changed environment. Building on their previous work [DWP06] applying repair plans to fix these consistency violations, they provide a cost-based plan selection mechanism of how to select these repair plans. The key merit of this work is the definition of the cost of plans which takes into account plan library hierarchy along with a scalable algorithm of cost calculation. To present an intuitive example of cost calculation, if repair plan P_1 involves 7 primitive actions whereas P_2 only needs 3 primitive actions. Then P_2 is viewed as cheaper than P_1. Therefore, when there are several applicable repair plans, the agent can select the cheapest plan among these plans.

There is also another plan selection strategy based on the cost and reward of plans in the work of [MLH$^+$14]. In this work, they focus on the uncertainty of beliefs of an agent, which leads to the uncertainty for determining the satisfiability of preconditions of plans. Therefore, the authors propose a plan selection strategy to choose plans that fulfil the maximum number of goals, the maximum degree of certainty, and resource-tolerance among the chosen plans. To do so, they first deal with plan selection to choose the best plan set with the maximum degree of certainty and achieving the maximum number of goals when no cost or reward information of plans is available. Then when cost and reward information is attached to plans, they can still choose the best set of plans that maximise expected profits (reward minus cost) while satisfying the others. Similarly, a recent work of [DEGL17] also proposed a utility-based plan selection in BDI agents in an environment with incomplete or uncertain information. To do so, they integrate the probability and utility into the BDI agents and select the most appropriate plan given all possible plans. In detail, each plan is assumed to have different known probabilities of successfully achieving the relevant goal. Then the utility of a plan regarding a state is the weighted average utility of all possible sub-plans according to their probabilities. Based on the utility of plans, therefore, the agent can select the plan with the highest utility to maximise the chance of achieving a goal.

Meanwhile, the work of [VTH11] explores the preference-based plan selection strategy in the BDI agents. To do so, they employ the preference language LPP [BFM06, BM07] to specify preferences. The preference of this work is expressed in terms of both properties of goals and resource usage of goals without having to know the details of how the goal is achieved. For example, the goal of booking a flight may have a property called `payment`, which specifies the payment method used. The vaules of this `payment` property can be, e.g. `credit` or `debit`. Any plan that achieves this flight-booking goal will result in the value being assigned to this property. In order to specify preferences over plans for a given goal, the preference of possible values for the related properties are specified instead. Regarding the resource usage of goals, the predicate `minimise(resource)` is introduced to express that the usage of the resource should be minimised whereas the predicate `usage(resource,amount,comparator)` to express the potential constraint

of resource usage. When reasoning preferences, unlike the fact that goal properties and their possible values are precisely known to the agent, the resource usage is dependent on the path chosen to achieve the goal. Therefore, the resource summaries of goals are obtained using the techniques from the work of [TP11]. Then an estimation function is introduced when attempting to satisfy preferences related to resource usage. Finally, since their approach is to express numerically how well a plan satisfies the preference formulas, they can sort the plans from most to least preferred and attempt the plans in that order to achieve the goal.

There is also another complementary work on preference-based plan selection proposed in the work of [PS13]. Typically, in BDI programs, it can be the common practice of assigning over-constrained context conditions to plans in order to ensure that the most preferred plan will be selected for use. However, an over-constrained context condition may limit the applicability of a given plan, e.g. where it could be of value as a back-up plan in other situations. Instead of overly constraining the context conditions of plans, it proposes an approach that maximises the applicability of plans while still being able to specify directly in a plan specification, aspects of the situation which would make the plan more or less desirable. To do so, they assign a quantitative value to each plan, using a local preference specification which allows dynamic calculation of this value based on both the current situation and attributes of the particular plan instance. Unlike the work of [VTH11] adapting another entire preference language for preference-based plan selection, the approach in [PS13] allows a straightforward declarative specification of the values of a plan. Furthermore, such declarative specifications at the plan level also make it straightforward to derive an explanation for a user as to why a particular plan was chosen in a particular situation.

We have discussed the work of [DW08], which arises from the aspect of software engineering, namely the maintenance. There is also another work of [TSP12] which proposes a plan selection based on the coverage from the perspective of software engineering. By the term of coverage for a plan, it intuitively implies how many world situations such a plan can be applicable in. To calculate the coverage of a plan, this work recasts the coverage problem as that of the model counting problem [GSS09]. To be precise, let a plan of a BDI agent be $e : \varphi \leftarrow P$. As standard in the model counting, we can have the model count of the propositional formula φ. Therefore, the coverage of a single plan is the model count of the propositional formula divided by the number of all possible worlds. In other words, the percentage of the state space in which a plan is applicable is considered as the coverage of a plan. Furthermore, they also describe the algorithm for calculating a measure of coverage of a plan considering the underlying goal-plan hierarchies. Indeed, an apparently high coverage of a plan may be compromised by the lower coverage of the sub-plans in the underlying tree. To utilise the coverage information for plan selection, it is intuitive to select a plan with the highest coverage measure to ensure the maximal success chance.

There is also some work starting from ethical aspects, e.g. moral values. The work of [CWDD17]

proposes a novel approach to plan selection in which societal, moral, and legal values of users influence decision-making. By using these values to select plans, the authors claim that a stable selection mechanism can be achieved because of the common base system of values among different people. The merit of this work is that it attempts to address the trust in the system by a human user in that the user can maintain a model of the system and can predict its future actions based on that model. To model the problem, they take two aspects into account: the goals and plans of the agent; and the values and their relationship. Regarding the value relationship, they follow the value hierarchy approach proposed by the work [Poe13]. It links values, norms, and design requirements or goals through a 'for the sake of' relationship. Similar to how the work of [VTH11] annotates the plans and goals with preference, this work extends BDI language by annotating plans with their effects on the value. To make the value-based decisions, they first take the constraint problem that has been generated from the goal-plan tree. Then plans are selected using a multi-criteria optimisation via a weighted sum in which each criterion measures the extent to which a particular value is currently satisfied. By employing an external constraint solver, it does not require changing the BDI languages or its implementation.

Finally, there is the work of [NL14] which proposes a plan selection mechanism which seeks the maximisation of some so-called softgoals, e.g. minimise time. Unlike many of the works we have discussed previously, their focus is to select the best plan by analysing the set of softgoals [BPG+04]. Each plan may contribute either positively or negatively to a softgoal, and can be characterised by a set of explicitly pre-defined contributions with a probability to a softgoal. In addition, similar to the work of [VTH11], there is a preference over the set of softgoals. For example, an agent may prefer the softgoal of saving money over the other softgoal of minimising time. However, instead of the strict ordering of preference in [VTH11], this work assigns a distribution over these softgoals to express the trade-off between different softgoals. The aim of their plan selection is to maximise the relevant contributions, considering the preferences over subgoals. To do so, they rely on the multi-attribute utility theory [KR76] to optimise the satisfaction of softgoals.

3.3.3 Learning-based Reasoning

Thus far, the works we have discussed above tackle the problem of plan selection by either the precedence of some characteristic (e.g. cost) or meta-level reasoning in the same language. While these approaches are useful, they are pre-programmed and do not take into account the experience of the agent. Therefore, the work of [SSPA10] provides an extended BDI framework that allows the agent to learn and adapt to the context conditions of plans. They argue that crafting fully correct context condition at design time can be a demanding and error-prone task. Also, fixed context conditions do not allow agents to adapt to potential variations of different environments. To address these limitations, they employ decision trees as the context condition of a plan, rather than original logical formulas. As such, for each plan, its decision tree (induced

based on previous execution) gives the agent information regarding how likely it is to succeed or fail in a particular world state. To select plans based on information in the decision tree, they propose a probabilistic plan selection function. Such a probabilistic plan selection function will select a most suitable plan according to the likelihood of success of each plan in different situations along with some measure of confidence in such a decision tree. Later on, they further extend it in the work of [SSP10] to include variables instead of propositional atoms and the recursive subgoaling. In doing this, they provide an approximate measure suitable for a recursive structure to replace the earlier measure of confidence which is based on a finite goal-plan tree.

Another similar work of [FN15] also proposes a plan selection approach to learn plans that provide possibly best outcomes. Similar to the work of [NL14], the ultimate goal of this work is also to maximise the agent satisfaction, considering the different side effects to softgoals and the preferences over these softgoals. However, unlike the work of [NL14], they do not require the programmers to explicitly provide probabilities of plan outcomes. They argue that the specification of probabilities of each possible plan outcome is hard to elicit and context-dependent. Furthermore, this specification may evolve over time. Instead, they require the factors (which can serve as contexts and are easy to identify) that can influence plan outcomes and the relationship between these factors and plan outcomes. Then they build a prediction model from the factors to plan outcomes based on recorded plan executions. To do so, there is initial learning to collect a sufficient amount of data. During this stage, plans may be selected randomly, for example. After the sufficient amount of data is obtained, a suitable existing machine learning algorithm can be applied to predict outcomes of plans. Finally, a function is proposed to transform plan outcomes into contributions associated with relevant softgoals.

3.4 Intention Selection

In the preceding sections, we have examined the existing works on plan selection in BDI agents. While useful, its focus is to decide what the best means is to use to achieve a given goal. However, a BDI agent typically pursues multiple goals in parallel due to its reactive nature (i.e. responding to new events while already dealing with other events). It means there are more decisions to be made for managing the concurrent execution of multiple intentions. For example, after the agent commits to applicable plans to multiple goals, the decision needs to be made about which intention is the best to execute next. Indeed, it is possible that the interleaving of steps in different intentions may result in conflicts, e.g. where the execution of a step in one plan makes the execution of a step in another plan impossible. Also, when an intention is selected to execute and there is a subgoal to achieve in this intention, the problem of plan selection for this subgoal may need to take into account other concurrent intentions. Indeed, an applicable plan which is suitable for one subgoal in one intention may not be compatible with the achievement of the rest of current intentions. For instance, an applicable plan may consume too much resource,

thus leaving insufficient resource for the rest of intentions. Unfortunately, most of the previous existing plan selection mechanisms fail to do so. For instance, the work of [SSPA10] clearly point out the assumption of the execution of a single intention. Similarly, the authors of the work [SSP06] also leave for future work accommodating parallel executions of goal and continues to omit this problem in the late journal version [SP11].

Similar to the support for the plan selection in BDI, the mainstream BDI programming languages also provide some preliminary operations that allow a developer to control which intention is scheduled for execution at the current cycle. For example, the intention selection function S_I in Jason [BHW07] allows a developer to customise intention selection function for a particular application domain. In addition, to avoid the potential conflicts between intentions, some atomic constructs are also available in languages such as Jason and A Practical Agent Programming Language (2APL) [Das08] to prevent the interleaving of steps in one intention from others. However, the hook for the intention selection function has normally required the programming in another language, e.g. Java. Regarding the non-interleaving atomic constructs, it may be either too difficult to know which intention should be kept separately from the execution of other intentions, or simply over-do it, thus missing the potential positive interactions between intentions. Therefore, a wealth of works have been released to incorporate these missing capabilities within BDI model in the following sections.

3.4.1 Summarisation-base Reasoning

In this section, we first look at one line of approaches which are based on information summarised from the goals and plans in the hierarchy of the plan library in BDI agents. One of the first works [TWPF02] starts from the intuition of potential resource conflicts when pursuing multiple goals in BDI agents. In general, different ways of accomplishing a goal may use different resources. Typically, in BDI agents, the plans to be used are chosen at runtime, based on the current context. As such, we cannot always say in advance precisely how many resources will be needed to achieve a given goal. To detect if a set of goals can be executed concurrently with no resource conflicts, the authors derive the possible and necessary resource summary information of relevant goals. In detail, the possible resource is the resource required by at least one plan of achieving the goal but not required by all plans, whereas the necessary resource is needed in every way to achieve a given goal. To simplify the problem, they assume that the agent designers can specify the necessary resource requirement for a plan via annotations. This necessary resource annotation for a plan essentially captures the necessary resource requirements for the actions in that plan. Based on the concrete necessary resource annotations for each plan, a handful of operators are introduced to compute the resource summaries goals according to the hierarchy of the plan library. Then the resource conflict reasoning can tell the feasibility of accomplishing all of the goals given the resources available. It also indicates when careful scheduling (e.g. reuse the resources) is necessary to ensure the achievement of all goals. In particular, it is useful to an agent when

deciding whether it can adopt a new goal or not with such resource conflict reasoning.

Similarly, there is also another work of [TPW03a] detecting and avoiding interference between a set of goals in BDI agents. Unlike the focus of potential resource conflict in [TWPF02], it concentrates on a particular type of negative interactions where the effects of one goal undo conditions that must be protected for successful completion of another goal. To do so, they obtain summary information about the definite and potential conditional requirements and effects of goals and their associated plans. Like the concept of necessary and possible resource in [TWPF02], a definite condition is a condition that will definitely be required at some point by every plan to achieve a given goal, whereas a potential condition required by at least one plan but not all. To facilitate the reasoning of interaction between goals, they also introduce the so-called preparatory effects and dependency links. To illustrate, if a plan P_1 brings about an effects ϕ which is the precondition of the following plan P_2, then there is a dependency link between the preparatory effect ϕ and the dependent plan P_2. With such information, their approach protects these dependency links to ensure the applicability of the dependent plans. They also want to protect the in-condition of a goal or a plan with the information of definite and potential in-conditions for goals and plans. Having calculated relevant summaries, they discuss ways of determining whether goals will definitely not interfere with each other and how to avoid potential interference via scheduling if possible.

Meanwhile, there is some work which is looking at exploiting positive interactions when pursuing a set of goals in parallel in BDI agents. For example, the work of [TPW03b] looks at the situations where there is potentially a common subgoal of multiple goals. To exploit this type of positive interaction, they propose a mechanism for identifying potential common subgoals and facilitating plan merging. Their approach is also based on their previous summarisation-based work [TPW03a]. In detail, the potential common subgoals are identified when maintaining summaries of definite and potential effects of goals and plans. The underlying motivation is that plans of different goals that can bring about the same effect could possibly be merged (i.e. executed one for all). To identify and facilitate plan merging, they store and monitor plans which could definitely and possibly be merged regarding the pursuit of the current goals.

3.4.2 External Tool-based Reasoning

Some researchers employ external tools to help the agent to pursue multiple intentions in parallel. To begin with, there is the work [BBJ+02] which employs a decision-theoretic task scheduler, called the Design-To-Criteria (DTC) scheduler, to automatically generate efficient intention selection functions for BDI agents. To do so, they first use a representation language called the TÆMS (Task Analysis, Environment Modelling, and Simulation) to represent the coordination aspects of intentions formally. Such a representation framework significantly improved the expressiveness of the language, thus facilitating the programming of certain types of application where quantitative reasoning is necessary. To be precise, not only does TÆMS provide quantitative

characteristics to tasks such as quality, cost, and duration, but also includes task relationships, e.g. enables, facilitates, and hinders. Furthermore, given the equivalence between methods (resp. tasks) and plans (resp. goals), they can obtain a TÆMS task structure library corresponding to the plan library in a BDI agent. Then for a given TÆMS task structure, the task scheduler DTC can produce alternative sequences in which an agent should execute the methods in that task structures to best satisfy the criteria (e.g. duration) and deadlines specified in the task structure.

There also exists a line of works which employ a stochastic approach, e.g. Monte-Carlo Tree Search (MCTS) [BPW+12] to scheduling intentions. For example, the work of [YLT14] proposes an approach to intention scheduling for BDI agents based on single-player MCTS that avoids conflicts between intentions. To do so, the input of single-player MCTS algorithm is a set of goal-plan trees representing the current set of intentions along with the current beliefs. And the output of the scheduling algorithm is the next step of one goal-plan tree to be executed at the current deliberation cycle. During each iteration of the MCTS algorithm, it consists of four phases, namely selection, expansion, simulation, and back-propagation, to guide the expansion of the search tree. Also, the node of the search tree records the previous and current steps in each goal-plan tree, the current environment, and some statistics (e.g. the number of times it has been visited). To demonstrate the performance of their approach, it compares its performance to that of summary information intention selection technique in the work of [TPW03a]. The experiment suggests that their stochastic approach is at least no worst than schedule using summary information.

Later on, there is another work of [YL16] which extends the work of [YLT14] in the following threefolds. First, not only does it avoid conflicts but also maximise fairness in the progression of the intentions in [YL16]. Secondly, it also allows the interleaving of primitive actions in different intentions. Thus, it differs from the work of, e.g. [TPW03a] and [YLT14], which only interleave intention at the plan level. Thirdly, a comprehensive evaluation is conducted to compare the performance of their stochastic approach to that of round-robin, non-interleaving, summary information-based, and coverage-based in both synthetic domain and realistic domain and both static and dynamic environments. The experimental results show their approach outperforms the rest of intention selection mechanisms regarding both the number of goal achieved and the variances in the goal achievement time.

Finally, it is worth mentioning there are two extended works, namely [YLT16a] and [YLT16b] based on the work of [YL16]. The work of [YLT16a] takes into account the deadlines of intentions with a straightforward extension to stochastic scheduling in [YLT16a]. Meanwhile, the work of [YLT16b] employs the stochastic scheduling in [YL16] to exploit the synergies between intentions. Instead of backtracking to recover from an execution failure, they propose an approach to appropriate scheduling the remaining progressable intentions to execute an already intended action which (hopefully) re-establishes a missing precondition. Also, they assume actions with stochastic effects to respond to a more realistic environment. To be precise, the intended outcome

of an action may or may not establish a precondition of a subsequent action in the plan. Their experiments show that their approach can avoid negative interactions between intention (as in [YL16]), while exploiting positive interactions to recover from execution failures.

3.4.3 Plan Selection Extended

Recall that we have discussed the work of [LL15] in Section 3.3.1 for plan selection through the procedural reflection. As a matter of fact, it also provides the support and freedom to the agent developers to customise the deliberation cycle to control which intentions to execute. The core idea to add to BDI language the ability to query the plan state (i.e. a collection of plan instances and their properties), and the actions which can manipulate the plan state. The large portion of their work is to provide a precise, declarative operational semantics for customised deliberation strategy which does not rely on user-specific functions. For example, the operation `scheduled(i)` specifies that a step of the plan instance with the ID `i` will be executed at the current deliberation cycle. To show the feasibility of their language, they replicate some typical of deliberation strategies found in the literature. For example, the meta-level rule `executable-intention(i)`→`schedule(i)` serves as the core rule to re-enable the previously progressed intention for execution again at the current cycle, provided it is still executable. Finally, an adaptive deliberation strategy is provided to avoid the conflicts between intentions while balancing the fairness of intention progression.

In the work of [TSP12], intention selection is also addressed along with the plan selection based on the notion of both coverage and overlap of plans and goals. The intuition of their work is that if there is a number of intentions waiting for selection to progress, the intention which has fewest possible successful execution (i.e. low coverage) is preferred for selection. In other words, the most vulnerable intention is prioritised for selection in case that the change of environment after selecting other intentions no longer enables it to be successfully achieved. Also, if there are more than two intentions with the same coverage, then the intention with the smallest overlap measure will be prioritised. Unlike the coverage, which measures how likely an intention will succeed regarding the environment, the concept of overlap measure quantifies how easy it can be recovered. As such, the agent can ensure that the most vulnerable and least recoverable intention will be selected preferably to ensure its successful execution.

Later on, the coverage-based intention selection is evaluated in the work of [WPS14]. To do so, they compare the coverage-based intention selection mechanism with the other two common intention selection mechanisms, namely round-robin and non-interleaved. Recall that the round-robin type of intention selection does a fixed number of steps on each intention in turn, whereas the non-interleaved processes each intention to completion in the order received. It is empirically shown that the coverage-based technique performs better under all circumstances, in particular in volatile environments where the plan library contains significant gaps regarding the coverage. Inspired by the nature of the coverage-based techniques, they also find out that the simple use of progressability checking when making intention selection amounts to a substantial improvement

in the number of successfully completed intentions. This is a substantial finding because this check can be readily applied to any other existing intention selection mechanisms. For example, it can enhance the round-robin with the progressability checking by selecting the first progressable intention and progress it until it becomes unprogressable and keeping the rest unchanged. Also, the experiments indeed show that the progressability checking enabled round-robin intention selection performs better than the plain round-robin.

3.4.4 Others

Finally, we notice that there are also some theoretical or architectural frameworks for deciding how goals interact and how an agent decides which goals to pursue. For example, the work of [PBL05] proposes a goal deliberation strategy, called Easy Deliberation, to allow the agent developers to specify the relationships between goals in an easy and intuitive manner in BDI agents. Similar to previous works on intentions, they also recognise that the goals of the agent can interact positively or negatively with each other. However, their contribution is to provide a suitable mechanism for handling goal relationships at the architectural level. Hence, the management of concurrent pursuit of goals can be left to the agent developers at the design phase. In achieving so, they begin with adopting an explicit representation of goals as described in [BPML04] which consists of a generic goal lifecycle and forms the basis for different goal types (such as achievement) in a different state (such as suspended). According to both the methodology Tropos [BPG^{+}04] and the engineering technique requirements KASO [LVL02], their strategy operates by the following two characteristics. Firstly, they identify the influence factors that drive the goal deliberation, namely cardinalities and inhibition arcs. The cardinalities tell the maximum number of active goals of a specific type, whereas the inhibition arcs specify the negative relationships between goals. Secondly, the deliberation process is initiated on two demands, namely when new goals are adopted or deactivated, and when goals are deactivated. The former case needs to decide if new goals can be activated and what are the implication to the current active goals. And the latter needs to decide the implication caused by the deactivated goal (e.g. some other goals inhibited by it). Despite the usefulness of this approach, the consideration of the conflicts is restricted to the goal level, and does not take into account the plans used to achieve the goals.

Similarly, the work of [ZRB16] also addresses the conflicting issue when the agent pursues multiple intentions in BDI agents. Arguably, their work extends the feature of atomic constructs which disallows the intentions interleaving in Jason agents. The key part of their approach is that the detection of conflicts is performed based on explicitly informing the conflicting plans in the agents. In other words, the developer is responsible for explicitly specifying the conflicts. In achieving so, they provide a detailed and expressive conflicting specification when writing agent programs. For example, the agent developer can inform the set of plans that conflict, e.g. conflict set $\{P_1, P_2, P_3\}$. To avoid the conflict at runtime, if a plan is already being executed, then only

plans that do not conflict with this plan can be instantiated and executed concurrently. Unlike any other work which adds the additional intention reasoning capacity, the authors believe that this approach has a high computational performance. In fact, their approach can be easily integrated into the existing BDI agent platforms, e.g. Jason, with two simple modifications. The first one is a simple annotation called `conflict`. And the second one is to add a further condition of checking no conflict plan currently executed before actually executing a plan.

3.5 Summary

We now close the section of the literature review with the following three tables. Firstly, Table 3.1 shows a detailed summary of all works on including planning in the BDI frameworks. Overall, it can be concluded that the planning seems well suited to be conjoined with BDI agents. A wide range of concrete planning techniques has been integrated into various BDI agent paradigms. HTN-like planning appears to be employed exclusively for the purpose of looking ahead on existing BDI plans according to Table 3.1. Meanwhile, various forms of FPP are integrated into BDI agents to create new plans when a path pursued via standard BDI execution turns out not to work. Furthermore, the community has also started to realise the importance of reusing plans from the planning tools when similar goals need to be achieved. It can be seen that the efforts of improving domain knowledge via the planning have started from both HTN side and FPP side (e.g. STRIPS). From the FPP side, the work of [ML07] leverages new plans from STRIPS-like planners whereas the work of [SP06] provides practical algorithms of converting MDPs policies into BDI rules. From the HTN side, the recent work of [Sil17] develops a formal account of coverting HTN hierarchies to obtain BDI goal-plan hierarchies.

Secondly, Table 3.2 shows a detailed summary of all plan selection works discussed in this chapter. Clearly, a large portion of works starts from some fixed characteristics of plans (e.g. cost). Based on these characteristics of plans, a suitable quantitative reasoning framework can be introduced. Also, the level of modification to the existing BDI languages also leads to different styles of approaches. For example, both works [VTH11, PS13] starts from the point of preference for plan selection. While [VTH11] adopts another expressive preference language to do so, [PS13] introduces a simple extension to the specification of the plans. Thus, there exists a natural correlation between the expressiveness of the plan selection extension and the level of modification to the BDI agents. Furthermore, to make the plan selection viable, either certain information is assumed (e.g. via annotation), or addition reasoning capacity is introduced to get the relevant information. For instance, the work of [TSP12] provides how to calculate the coverage and overlap of plans, whereas the work of [CWDD17] simple annotates the value changes to the plans.

Thirdly, Table 3.3 shows a detailed summary of intention selection works discussed in this chapter. Indeed, the intention selection is difficult as it needs to decide not only what set of

means to which it commits at any time, but also the way those means are progressed, given the goals of the agent and the plans it has to achieve them. A clear consensus from these works in Table 3.3 is that the intention selection at least needs to ensure the successful achievement of all current given goals. Therefore, a large number of works address how to avoid the potential adverse interactions between multiple intentions, e.g. in [TWPF02, TPW03a, ZRB16] and the whole line of applying MCTS method. Arguably, the difference between these approaches is in their degree of encapsulation. To illustrate, the work of [YL16] encapsulate all scheduling decisions (including which plan to select, which intention to progress) in a single MCTS process. Meanwhile, the approaches based on summary information functions as "a standing advisor" to determine whether the adaptation of a new goal would conflict with other existing goals in [TWPF02]. However, it is still up to the agent to decide whether to adopt a new goal or not.

Table 3.1: Summary of the works of incorporating planning in BDI discussed in this chapter

Problem	**Type**	**Approach**	**Reference**
Planning to Generate New BDI Plans	STRIPS Planning	[DI99]	PRS, abstract plan
		[SSP09]	CAN, hybrid plan
		[ML07, ML08]	AgentSpeak, abstract plan, and plan reuse
		[MZM04]	X-BDI, non-abstract plan
	Probabilistic Planning	[SWP02]	POMDPs versus BDI
		[SP06]	MDPs versus BDI
		[BMH+16]	POMDPs, AgentSpeak, and hybrid plan
		[RM17]	POMDPs, Plan Reuse
		[KBM+16]	MDPs, risk-aware planning
Applying Lookahead Planning to Existing BDI Plans	HTN Planning	[DI99]	HTN-like simulation
		[WMLW95]	HTN+ PRS in ACT language
		[SP04]	HTN versus BDI
		[SP05a, SP05b]	HTN in CAN
		[SSP06, SP11]	CanPlan
		[BLH+14]	CanPlan, Uncertainty
		[Sil17]	Plan reuse from HTN to BDI
		[WBPL06]	HTN in Jadex

Table 3.2: Summary of the works of plan selection in BDI discussed in this chapter

Plan Selection			
Type	**Feature**	**Approach**	**Reference**
Meta-level Reasoning		[HBHM99] [DDBDM03]	3APL, meta-level language
		[Win05]	AgentSpeak, meta-interpreter
		[LL15]	3APL, meta-APL
Precedence-based Reasoning	Cost	[DW08]	Maintainance, repair plan
	Cost, Reward	[MLH+14]	Uncertain beliefs
	Utility	[DEGL17]	Plans with known utility and success probability
	Preference	[VTH11]	Employ other preference language
		[PS13]	Extending plan specifications
	Coverage	[TSP12]	Model counting problems
	Moral value	[CWDD17]	Trust, constrain problems,
	Softgoals	[NL14]	Multi-attribute utility optimisation
Learning-based Reasoning		[SSPA10, SSP10]	Decision tree instead of logical formulas
		[FN15]	Extension of [NL14] with prediction for plan outcomes
Default Function		[BHW07]	Required another language (e.g. Java)

Table 3.3: Summary of the works of intention selection in BDI discussed in this chapter

Intention Selection			
Type	**Feature**	**Approach**	**Reference**
Summarisarion-based Reasoning	Resource conflict	[TWPF02]	Possible and necessary resource summary
	Interference	[TPW03a]	Definite and potential conditional requirements
	Positive interactions	[TPW03b]	Plan merging
External Tool-based Reasoning	Task scheduler	[BBJ+02]	DTC, TÆMS
	MCTS	[YLT14]	Plan level intention interleaving
		[YL16]	Action level intention interleaving, stochastic effects
		[YLT16a]	Deadline
		[YLT16b]	Failure recovery
Plan Selection Extended	Meta-level	[LL15]	Plan manipulation
	Precedence-base	[TSP12]	Coverage and overlap
		[WPS14]	Progressability check
Other		[PBL05]	Goal-level deliberation
		[ZRB16]	Conflict specifications
Default Function		[BHW07]	Require another language, e.g. Java

Chapter 4

Recovering Agent Program Failure via Planning

The bulk of this chapter has been published online in [XBML18a].

4.1 Introduction

The Belief-Desire-Intention (BDI) agent systems, where the agents are modelled based on their beliefs, desires, and intentions, provides a practical approach to developing intelligent agent systems. Typical BDI agents rely on user-provided plan library (i.e. a set of plan rules) to achieve goals, and online context-sensitive plan selection and goal expansion. These allow for the development of systems that are incredibly flexible and responsive to the environment. As a result, the agents modelled in BDI style are well suited in complex application domains, such as control systems [JB03] and power engineering [MDC+07]. While the use of a set of pre-defined plans simplifies the planning problem to an easier plan selection problem, obtaining a plan library that can cope with every possible eventuality requires adequate plan knowledge. This knowledge is not always available, particularly when dealing with uncertainty. Therefore, this limits the autonomy and robustness of BDI agent systems, often with deleterious effects on the performance of the agent when there is no applicable plan for achieving a goal at hand.

To illustrate the problem, consider the following running example (see Figure 4.1). In a smart home environment, there is an intelligent domestic robot whose job includes daily household chores (e.g. sweeping), security monitoring (e.g. burglary), and entertainment (e.g. playing music). The environment is dynamic and pervaded by uncertainty. When the robot does chores in the `lounge`, it may not be pre-encoded with plans to deal with an overturned clothes rack in the `lounge`, one of the doors to the `hall` being blocked unexpectedly, or urgent water overflow in a bathroom. Indeed, it is unreasonable to expect an agent designer to foresee all exogenous events and provide suitable pre-defined plans for all such eventualities. To address this weakness, the

Figure 4.1: Layout of a Smart House with a Domestic Robot

desirable behaviours of such a robot agent should be able to come up with new plans to deal with such unforeseen events at design time in order to act intelligently.

Fortunately, to alleviate (some of) these issues, a large body of work on integrating various planning techniques with BDI agents have been proposed in recent years, as reviewed in Section 3.1. For example, the work of [ML07] proposed integration of AgentSpeak and a classical First-principles Planning (FPP) in which a new planning action in AgentSpeak is introduced to incorporate this planner. This action is bound to an implementation of a planning component, and takes as an argument the desired world state along with the plan library and the current belief base to generate a new plan. The BDI agent designer may include this new planning action at any point within a standard AgentSpeak plan to call a planner. In the work of [SSP09], the authors provide a formal framework for FPP in BDI agent systems. This framework employs FPP to generate abstract plans, that is, plans that include not only primitive actions, but also abstract actions summarised from the plan library. It allows for flexibility and robustness during the execution of these abstract plans. However, most of the existing approaches (e.g. [ML07, SSP09, BMH+16]) which are reviewed in Section 3.1 integrate with FPP requiring the agent designer to define when the FPP is triggered. These ad-hoc styles of approaches limit the power of FPP to assist BDI agent systems to accomplish their goals as the points of calling FPP effectively can be unpredictable. Therefore, the goal of the contributions in this chapter is to advance the state-of-art of planning in BDI agents by developing a rich and detailed specification of the appropriate operational behaviour when FPP is pursued, succeeded or failed, suspended, or resumed. To achieve so, we introduce a novel operational semantics for embedding FPP in BDI agent systems. This semantics specifies when and how FPP can be called, and precisely articulates how a BDI agent system manages the FPP. Such a semantic approach of ours also responds to the lack of work in

strengthening the theoretical foundations of the BDI agent pointed out by the comprehensive survey [MS15] as one of the future directions for planning in BDI agents.

In this chapter we present a systematic study of the tight integration of FPP within a typical BDI agent programming language, namely Conceptual Agent Notation (CAN). The structure of this chapter is as follows. In Section 4.2, we introduce when the FPP can be utilised in CAN framework in an intrinsic manner. Specifically, we semantically enumerate all potential execution failure which the FPP can generate new plans to recover. Contradictory to the restricted approaches in the works (e.g. [ML07]) which requires the agent developers to specify when to trigger the FPP, our approach makes CAN agents self-aware of when they should call for help from FPP. It not only reduces the responsibilities of agent designers, but also, more importantly, ensures the maximal appropriate usage of FPP when needed. In Section 4.3, we then provide the strategy to recover these execution failures previously enumerated by calling FPP. To achieve so, we extend the intentions of BDI agents with declarative intentions and denote the original intentions as procedural intentions. This partition of the intention set in CAN agents allows fine-grained management where procedural intentions manage the existing agent programs telling how to achieve a goal, while declarative intentions instruct the embedded FPP what to achieve. In detail, we address how to recover all execution failure by adding the relevant declarative intentions for FPP to plan for with precise derivation rules. In Section 4.4, we discuss how the agent manages the declarative intentions and how the CAN agent executes the plan generated from FPP. The formal relationship between FPP and the CAN agent execution is established in Section 4.5. Finally, in Section 4.6, we offer an intricate scenario discussion, which supports the feasibility of the resulting framework and motivates the merits of the proposed framework to warrant future work on a fully implemented system.

4.2 Execution Failure in BDI

We now discuss how CAN agent systems and FPP can be integrated into a single framework. The resulting framework, called CAN(FPP), allows us to define agents that can perform FPP to provide new behaviours at runtime in an uncertain environment. We start by semantically enumerating the potential execution failure, namely the procedural execution failure and declarative execution failure in CAN agents in the basic configuration (i.e. how to evolve a single intention).

4.2.1 Procedural Execution Failure

We begin with the procedural execution failure which specifies the potential failure of all agent programs but not the declarative goal program $goal(\varphi_s, P, \varphi_f)$. We have two potential types of procedural execution failure, namely the coverage failure and precondition failure.

The **coverage failure** captures the type of failure when there is no applicable plan for the current (sub)goal. To be precise, all relevant plans are selected, i.e. $\Delta = \{\varphi\theta : P\theta \mid (e' = \varphi \leftarrow P) \in$

$\Pi \wedge \theta = mgu(e', e)\}$ to deal with an event goal e. However, there may not exist a relevant plan $\varphi\theta : P\theta$ such that its context condition holds for the current belief base, (i.e. $\nexists\ \varphi\theta : P\theta \in \Delta$ such that $\mathcal{B} \models \varphi\theta$). In this case, we say that a coverage failure occurs as there does not exist some plan that is applicable for every situation. Therefore, we can have the following derivation rule to capture this coverage execution failure where *?false* is a failed program and the label `cov` stands for coverage.

$$\frac{\Delta = \{\varphi\theta : P\theta \mid (e' = \varphi \leftarrow P) \in \Pi \wedge \theta = \text{mgu}(e', e)\} \quad \nexists\ \varphi\theta : P\theta \in \Delta \quad \mathcal{B} \models \varphi\theta}{\langle \mathcal{B}, \mathcal{A}, e : (\mid \Delta \mid) \rangle \xrightarrow{\text{cov}} \langle \mathcal{B}, \mathcal{A}, ?\mathit{false} \rangle} \quad F_{\texttt{cov}}$$

The **precondition failure** captures the type of failure when a precondition of an action does not hold before being executed. This type of execution failure can happen, e.g. due to the dynamic nature of the environment. For instance, before a robot proceeding passing through a door (i.e. action `gothrough(door1)`), the door was slam shut by, e.g. the pet. The following derivation rule is given to capture the coverage execution failure where the label `pre` stands for precondition.

$$\frac{a : \psi \leftarrow \phi^-; \phi^+ \in \Lambda \quad a\theta = act \quad \mathcal{B} \not\models \psi\theta}{\langle \mathcal{B}, \mathcal{A}, act \rangle \xrightarrow{\text{pre}} \langle \mathcal{B}, \mathcal{A}, ?\mathit{false} \rangle} \quad F_{\texttt{pre}}$$

4.2.2 Declarative Execution Failure

We have discussed the types of procedural execution failure in which the agent programs that describe how to achieve a given goal get blocked due to various reasons. We are now ready to have a look at the other type of failure, namely the declarative execution failure, which focuses on the special declarative goal program $goal(\varphi_s, P, \varphi_f)$. Recall that a declarative goal program $goal(\varphi_s, P, \varphi_f)$ states that the success condition φ_s should be achieved through the procedural program P, failing when φ_f becomes true, and retrying (alternatives) as long as neither φ_s nor φ_f is true (see [SP07]). We can see that the declarative goal amounts to a unique behaviour nature which is different from the normal procedural programs. Unlike the procedural program in which it is deemed successful if it has been executed successfully, the ultimate aim of a declarative goal is to achieve the success condition in it regardless of the execution state of the related procedural program. In the following, we focus on two types of declarative execution failure, namely procedural component failure and empty procedure failure.

The **procedural component failure** refers to the type of failure when the procedural component P which is used to achieve the successful state in a declarative goal program $goal(\varphi_s, P, \varphi_f)$ can no longer progress (i.e. blocked). For example, the reason of procedural component P in $goal(\varphi_s, P, \varphi_f)$ being blocked may be due to the coverage failure discussed in the procedural execution failure in Section 4.2.1. At first glance, it is tempting here to classify these situations into the procedural execution failure in Section 4.2.1. On closer inspection, unlike the procedural programs whose failure is inability to execute themselves in full, the purpose P in $goal(\varphi_s, P, \varphi_f)$ is to achieve the success condition φ_s rather than focusing on its own accomplishment. Therefore,

we assign them to the different type of failure, i.e. procedural component failure for a declarative goal program.

The following derivation rule captures the type of procedural component failure where the label `fail` denotes the failure of the procedural component (i.e. $\langle \mathcal{B}, \mathcal{A}, P \rangle \nrightarrow$) in a declarative goal.

$$\frac{\mathcal{B} \nvDash (\varphi_s \vee \varphi_f) \quad \langle \mathcal{B}, \mathcal{A}, P \rangle \nrightarrow}{\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, P, \varphi_f) \rangle \xrightarrow{\text{fail}} \langle \mathcal{B}, \mathcal{A}, ?false \rangle} \quad F_{\texttt{fail}}$$

The **empty procedure failure** is the type of failure in which there is no procedural program given to achieve a declarative goal. In other words, a declarative goal program $goal(\varphi_s, nil, \varphi_f)$ is initially written as a part of the plan-body program where $P = nil$ is syntactic sugar representing that there is no available procedural information on how to achieve the goal. Indeed, such a scenario can occur when either the procedural program was not known during the design time, or there are no efforts made to create pre-defined plans (e.g. due to the priority of other parts of plan library design tasks). Once the Belief-Desire-Intention (BDI) agent encounters such a declarative goal $goal(\varphi_s, empty, \varphi_f)$, it will return a failure, giving us:

$$\frac{\mathcal{B} \nvDash (\varphi_s \vee \varphi_f)}{\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, nil, \varphi_f) \rangle \xrightarrow{\text{empty}} \langle \mathcal{B}, \mathcal{A}, ?false \rangle} \quad F_{\texttt{empty}}$$

where the label `empty` stands for the empty procedural condition in the declarative goal.

4.3 Declarative Intentions in BDI

We have discussed and differentiated the distinct types of execution failure in BDI agents, which can potentially be recovered by First-principles Planning (FPP). In this section, we introduce the concept of declarative intentions (used by FPP) and its semantical operations regarding how to recover various types of execution failure. In a Conceptual Agent Notation (CAN) agent, the intention set Γ is limited to just procedural intentions. While valuable, procedural intentions only describe how to achieve a given goal and do not answer the question as to which goals FPP should be trying to achieve in the BDI agent. To address this shortcoming, we partition the intention set Γ in this work into two sets, namely procedural intention set Γ_{pr} and declarative intention set Γ_{de} such that $\Gamma = \Gamma_{pr} \cup \Gamma_{de}$ and $\Gamma_{pr} \cap \Gamma_{de} = \emptyset$. This straightforward extension allows us to keep track of both procedural intentions (executed by the BDI engine) and declarative intentions that tells us what we want to achieve (used by FPP). Each set of intentions Γ_i is furthermore partitioned into the subset of active intentions Γ_i^+ and the suspended intentions Γ_i^- where $i \in \{pr, de\}$. The key advantage of this detailed description of intention states is to provide a middle layer state, namely the suspended, to the intentions. Therefore, the agent can temporarily suspend its intentions before making decisions of whether it should permanently drop them or recover them before resuming. We also assume adding an element to Γ_i^+ ensures the element is removed from Γ_i^- and vice versa where $i \in \{pr, de\}$.

To define the elements in the declarative intentions, we introduce the concept of pure declarative goals. A pure declarative goal $goal(\varphi_s, \varphi_f)$ is obtained from the ordinary declarative goal $goal(\varphi_s, P, \varphi_f)$ in BDI agents by dropping the procedural component P. It is read as "*achieve* φ_s; *failing if* φ_f *becomes true*". This new goal structure encodes the minimum information of what FPP needs to achieve (i.e. successful condition φ_s) and when it is sensible to halt FPP (i.e. failure condition φ_f). To avoid confusion, we note that the special normal declarative goal $goal(\varphi_s, nil, \varphi_f)$ conceptually encodes the same information as the pure declarative goal $goal(\varphi_s, \varphi_f)$ due to that nil is an empty procedure. However, these two have profoundly different semantical behaviours, namely one executed by BDI engine and the other used by FPP. In the following, we will present the derivation rules in the agent configuration (i.e. how to execute a complete agent) to recover the relevant execution failure previously enumerated in Section 4.2. To do so, we will show how to obtain and add appropriate pure declarative goals based on other types of agent programs into declarative intentions (achieved by FPP) for each type of execution failure.

4.3.1 Procedural Execution Failure Recovery

We now consider the recovery strategies which add the appropriate pure declarative goals for the type of procedural execution failure presented in Section 4.2.1.

The first recovery strategy is to recover the coverage failure in which there is no applicable plan to achieve a given goal. In BDI agents, when no plan is available, then the goal is deemed failed (and potentially dropped by the agent). However, as one of the properties of goals held by a rational agent is that they should persist [WPHT02], it is rational to retry and pursue these goals if possible. To obtain such persistence, some BDI agent may temporarily suspend this goal and wait until one of the relevant plans becomes applicable somehow. Similarly, we also first temporarily suspend all relevant plans of a given goal in our approach. Unlike the passively waiting approach to maintaining the persistence of goals, however, we proactively make the relevant plan applicable by adding a pure declarative goal whose success condition is the precondition of one of the relevant plans into the declarative intention. In other words, we want FPP to establish the precondition of one of the relevant plans to ensure the continuing pursuit of the given goal. For simplicity, the failure condition in the newly added pure declarative goal can be empty (i.e. *nil*). After establishing the precondition of one of the relevant plans, we also utilise a motivational library $\mathcal{M}$ which is a collection of rules of the form: $\psi \rightsquigarrow P$, to resume the agent program P based on changes in beliefs. Therefore, through the motivational library, the agent can always be aware when to resume the temporarily suspended program, giving us the following derivation rule:

$$\frac{e:(\!|\,\Delta\,|\!) \in \Gamma^{+}_{pr} \quad \langle \mathcal{B}, \mathcal{A}, e:(\!|\,\Delta\,|\!)\rangle \xrightarrow{\text{cov}} \langle \mathcal{B}, \mathcal{A}, ?false\rangle \quad \varphi\theta : P\theta \in e:(\!|\,\Delta\,|\!)}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^{+}_{pr}, \Gamma^{+}_{de}, \mathcal{M}\rangle \rightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^{+}_{pr} \setminus \{e:(\Delta)\}, \Gamma^{+}_{de} \cup \{goal(\varphi\theta, nil)\}, \mathcal{M} \cup \{\varphi\theta \rightsquigarrow e:(\!|\,\Delta\,|\!)\}\rangle} \; R_{\text{cov}}$$

The rule R_{cov} is first to suspend all relevant plans of a given goal (i.e. $\Gamma^{+}_{pr} \setminus e:(\!|\,\Delta\,|\!)$) in the procedural intention set while adopting a precondition of a relevant plan to be a declarative

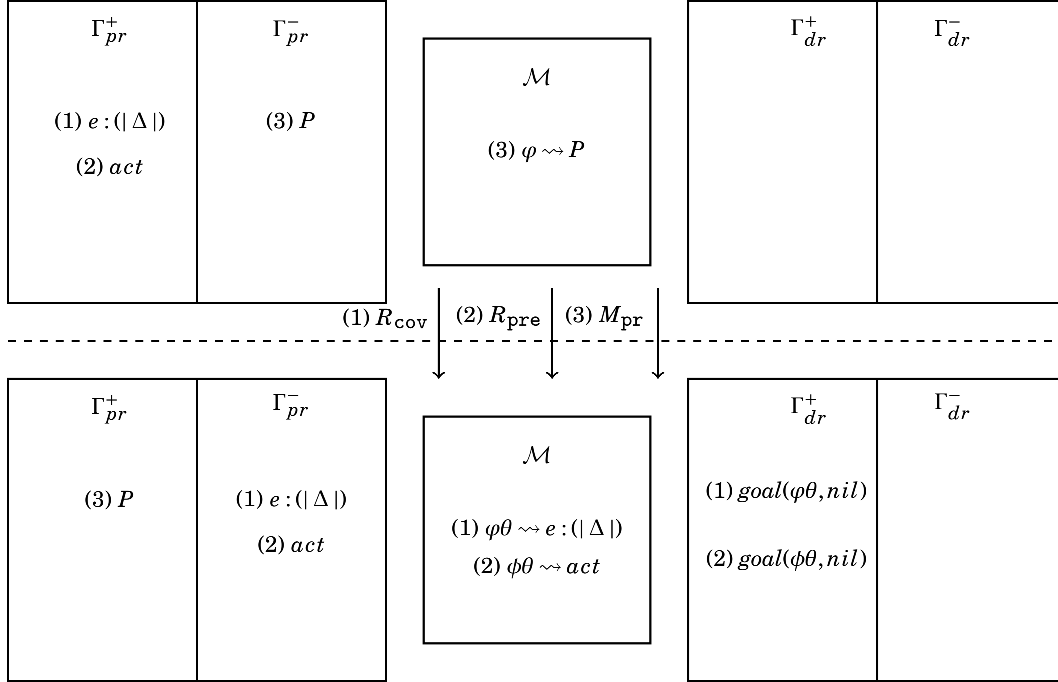

Figure 4.2: Diagrammatic Evolutions of the Rule (1) R_{cov}, (2) R_{pre}, and (3) M_{pr}

intention achieved by FPP (i.e. $\Gamma^+_{de} \cup \{goal(\varphi\theta, nil)\}$). After the adoption of such a pure declarative intention, it also adopts a new motivation rule $\varphi\theta \rightsquigarrow e : (\mid \Delta \mid)$ to resume the suspended agent program $e : (\mid \Delta \mid)$ once $\varphi\theta$ holds. As such, the BDI agents can automatically resume selecting an applicable plan to address the event e after the precondition of one of its relevant plans is established by FPP. The pictorial form of illustration of this rule is given and denoted by (1) in Figure 4.2. Also, it is noted we only mention $\mathcal{M}$ in the agent configuration when it needs modifying; for all other rules, the motivational library remains unchanged, thus omitted.

The second recovery strategy is to recover the precondition failure in which the precondition of an action does not hold right before being executed. To recover the precondition failure, similarly, the agent can temporarily suspend such a non-executable action in the procedural intention and adopt its precondition in the declarative intention before trying executing it again. We have the following rule R_{pre} with the pictorial form of illustration of this rule, denoted by (2), in Figure 4.2.

$$\frac{a : \psi \leftarrow \phi^-; \phi^+ \in \Lambda \quad a\theta = act \in \Gamma^+_{pr} \quad \langle \mathcal{B}, \mathcal{A}, act \rangle \xrightarrow{\text{pre}} \langle \mathcal{B}, \mathcal{A}, ?false \rangle}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr}, \Gamma^+_{de}, \mathcal{M} \rangle \to \langle \Pi, \Lambda, \mathcal{B}, \Gamma^+_{pr} \setminus \{act\}, \Gamma^+_{de} \cup \{goal(\psi\theta, nil)\}, \mathcal{M} \cup \{\psi\theta \rightsquigarrow act\} \rangle} \; R_{\text{pre}}$$

We now close this section by providing an extra derivation rule to reactivate a suspended procedural intention via a motivation rule with the pictorial form of illustration of this rule, denoted by (3), in Figure 4.2.

$$\frac{P \in \Gamma^-_{de} \quad \varphi\theta \rightsquigarrow P \in \mathcal{M} \quad \mathcal{B} \models \varphi\theta}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr}, \Gamma_{de}, \mathcal{M} \rangle \to \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr} \cup \{P\}, \Gamma_{de}, \mathcal{M} \setminus \{\varphi\theta \rightsquigarrow P\} \rangle} \; M_{pr}$$

4.3.2 Declarative Execution Failure Recovery

In this section, we discuss how to recover the declarative execution failure discussed in Section 4.2.2. The first recovery strategy is to recover the procedural component failure when the procedural component P which is used to achieve the success condition in a declarative goal $goal(\varphi_s, P, \varphi_f)$ can no longer progress further. When such a prodedural program P is blocked and either the success condition φ_s or failure condition φ_f holds, then the declarative goal $goal(\varphi_s, P, \varphi_f)$ is not accomplished. Recall that the ultimate accomplishment sign of a declarative goal is that the success condition holds. Therefore, instead of recovering for the failure of the actual procedural component, we decide to discard such a normal declarative goal first. Secondly, the relevant pure declarative goal is obtained by keeping the success and failure condition in the original normal declarative goal and then added it into the declarative intention for FPP to achieve. Thirdly, we also suspend the agent program (i.e. P'') which follows this normal declarative goal (i.e. $goal(\varphi_s, P, \varphi_f); P''$) and resume P'' when the success condition is achieved via the motivational library $\varphi_s \rightsquigarrow P''$. In contrast to the alternative convoluted behaviour, which hopes to achieve φ_s by recovering the procedural component P, our approach is direct and can potentially save the cost of execution. The following derivation rule captures our direct approach:

$$\frac{P' = goal(\varphi_s, P, \varphi_f); P'' \in \Gamma^+_{pr} \quad \langle \mathcal{B}, \mathcal{A}, P' \rangle \xrightarrow{\text{fail}} \langle \mathcal{B}, \mathcal{A}, ?false \rangle}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{pr}, \Gamma^+_{de} \rangle \rightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \langle \Gamma^+_{pr} \setminus \{P'\}, \Gamma^-_{pr} \setminus \{goal(\varphi_s, P, \varphi_f)\} \rangle, \Gamma^+_{de} \cup \{goal(\varphi_s, \varphi_f)\}, \mathcal{M} \cup \{\varphi_s \rightsquigarrow P''\} \rangle} \; R_{\texttt{fail}}$$

In rule $R_{\texttt{fail}}$, $\langle \Gamma^+_{pr} \setminus \{P'\}, \Gamma^-_{pr} \setminus \{goal(\varphi_s, P, \varphi_f)\} \rangle$ ensures the deletion of the normal declarative goal $goal(\varphi_s, P, \varphi_f)$ from the procedural intentions and the suspension of the agent program following $goal(\varphi_s, P, \varphi_f)$, namely P''. The adoption of a new pure declarative intention $goal(\varphi_s, \varphi_f)$ is achieved by $\Gamma^+_{de} \cup \{goal(\varphi_s, \varphi_f)\}$. Finally, $\mathcal{M} \cup \{\varphi_s \rightsquigarrow P''\}$ adds a motivation rule $\varphi_s \rightsquigarrow P''$ to pursue the agent program P'' which follows the original declarative goal program $goal(\varphi_s, P, \varphi_f)$. The pictorial form of illustration of this rule is given and denoted by (1) in Figure 4.3.

Similarly, we can recover the empty procedure failure in the same way as we do in procedural component failure. Therefore, we have the following derivation rule for empty procedure failure, and its pictorial form of illustration is given and denoted by (2) in Figure 4.3.

$$\frac{P = goal(\varphi_s, nil, \varphi_f); P' \in \Gamma^+_{pr} \quad \langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, nil, \varphi_f) \rangle \xrightarrow{\text{nil}} \langle \mathcal{B}, \mathcal{A}, ?false \rangle}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{pr}, \Gamma^+_{de} \rangle \rightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \langle \Gamma^+_{pr} \setminus \{P\}, \Gamma^-_{pr} \setminus \{goal(\varphi_s, nil, \varphi_f)\} \rangle, \Gamma^+_{de} \cup \{goal(\varphi_s, \varphi_f)\}, \mathcal{M} \cup \{\varphi_s \rightsquigarrow P'\} \rangle} \; R_{\texttt{emp}}$$

We also allow adding a pure declarative goal to the declarative intention set Γ_{de} in a proactive manner through the motivational library $\mathcal{M}$. Semantically, we need to add another derivation rule for the motivational library $\mathcal{M}$ so that a pure declarative goal can be added directly to Γ_{de} when the rule is triggered (first and second premise), and the program is a pure declarative goal (third premise) where the label `mot` stands for motivation. Similarly, its pictorial form of illustration is given and denoted by (3) in Figure 4.3.

$$\frac{\psi \rightsquigarrow P \in \mathcal{M} \quad \mathcal{B} \models \psi\theta \quad P = goal(\varphi_s, \varphi_f)}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{de}, \mathcal{M} \rangle \rightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{de} \cup \{P\}, \mathcal{M} \setminus \{\psi \rightsquigarrow P\} \rangle} \; R_{mot}$$

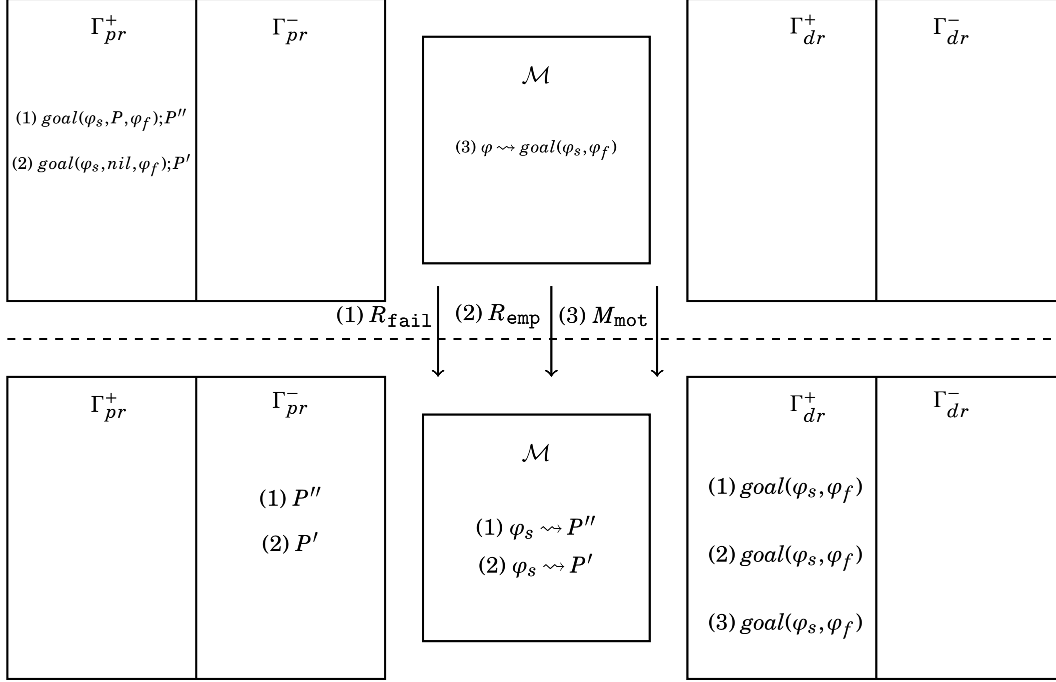

Figure 4.3: Diagrammatic Evolution of the Rule (1) $R_{\texttt{fail}}$, (2) $R_{\texttt{emp}}$, and (3) $R_{\texttt{mot}}$

Finally, we close this section by noting that all we have been doing is to add the appropriate pure declarative goals to the declarative intention set for FPP to achieve in order to recover some execution failure. In the following section, we show how to invoke FPP to address the pure declarative goals and how the BDI agents execute the solutions from FPP to actually recover execution failure.

4.4 First-Principles Planning in BDI

We now consider how FPP integrates with the BDI system and how BDI manages FPP. Firstly, when either φ_s or φ_f is true, the pure declarative goal $goal(\varphi_s, \varphi_f)$ has been completed. Therefore, it should be dropped from Γ_{de} shown in the following derivation rule:

$$\frac{G \in \Gamma_{de} \quad G = goal(\varphi_s, \varphi_f) \quad \mathcal{B} \models \varphi_s \vee \varphi_f}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{de} \rangle \xrightarrow{drop} \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{de} \setminus \{G\} \rangle} \; G_{drop}$$

Recall that in Stanford Research Institute Problem Solver (STRIPS), an FPP planning problem is a 3-ary tuple $\langle s_0, \varphi_g, O \rangle$ where s_0 represents the initial state, φ_g the goal formula, and O a set of operators (seen in Section 2.4.2). Let $goal(\varphi_s, \varphi_f)$ be a pure declarative goal in a BDI agent whose current belief is $\mathcal{B}$ and action library is Λ. For each planning to address the pure declarative goal $goal(\varphi_s, \varphi_f)$, we can have a corresponding FPP problem $\langle \mathcal{B}, \varphi_s, \Lambda \rangle$ where $\mathcal{B}$ is the initial state of this planning, φ_s the goal formula, and Λ a set of operators. Naturally, the

solution of an FPP problem $\langle \mathcal{B}, \varphi_s, \Lambda \rangle$ is denoted to be $sol(\mathcal{B}, \varphi_s, \Lambda)$. From now on, we will also distinguish between online planning, e.g. [KE12] and offline planning, e.g. [HN01]. Recall that offline planning generates a complete sequence of actions and executes them one by one to reach a goal state whereas online planning generates an incomplete plan, and interleaves execution and planning until a goal state is reached. In this thesis, we are interested in the online case when a single action is returned based on current belief states, and executed immediately. Formally, the offline solution of an FPP problem $\langle \mathcal{B}, \varphi_s, \Lambda \rangle$ can be written as $sol^{off}(\mathcal{B}, \varphi_s, \Lambda) = act_1; \ldots; act_n$ whereas the online solution $sol^{on}(\mathcal{B}, \varphi_s, \Lambda) = act$. Furthermore, we denote the action act generated by FPP as act^{FPP} to distinguish itself from actions written by BDI programers when necessary. In practice, this extra information can be easily enclosed by, e.g. annotation in Jason [BHW07]. In the following part, we will provide different derivation rules for accommodating each type of planning due to their aforementioned contrasting nature.

In offline planning, a complete sequence of actions to solve an FPP problem is first generated and then executed afterwards. The derivation rule for offline planning can be defined as follows:

$$\frac{goal(\varphi_s, \varphi_f) \in \Gamma^+_{de} \quad sol^{off}(\mathcal{B}, \varphi_s, \Lambda) = act_1; \ldots; act_n}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr}; \Gamma_{de} \rangle \rightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr} \cup \{goal(\varphi_s, act_1; \ldots; act_n, \varphi_f)\}; \Gamma^-_{de} \cup \{goal(\varphi_s, \varphi_f)\} \rangle} \quad P^{off}_F$$

The rule of P^{off}_F shows that the agent will adopt a new declararive goal whose procedural component is the sequence of actions generated by FPP (i.e. $goal(\varphi_s, act_1; \ldots; act_n, \varphi_f)$) to achieve the successful state φ_s if there exisits a complete offline solution to it (i.e. $sol^{off}(\mathcal{B}, \varphi_s, \Lambda) = act_1; \ldots; act_n$). The adoption of such a new declararive goal $goal(\varphi_s, act_1; \ldots; act_n, \varphi_f)$ takes the advantage of the existing declarative goal semantics in CAN language (seen in Section 2.3.2.1). In detail, it allows the agent to halt the execution of this sequence of actions generated by FPP if either φ_s or φ_f holds during the execution. Meanwhile, the rule of P^{off}_F also ensures the BDI agent to suspends this already planned pure declarative goal (i.e. $\Gamma^-_{de} \cup \{goal(\varphi_s, \varphi_f)\}$).

In online planning, the next action is generated in each planning phase and executed afterwards. This loop of "plan one action–execute one action" is iterated until the goal is reached. Therefore, the derivation rule for an online planning is defined as follows:

$$\frac{goal(\varphi_s, \varphi_f) \in \Gamma^+_{de} \quad sol^{on}(\mathcal{B}, \varphi_s, \Lambda) = act \quad P = act; activate(goal(\varphi_s, \varphi_f))}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr}; \Gamma^-_{de} \rangle \rightarrow \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma^+_{pr} \cup \{goal(\varphi_s, P, \varphi_f)\}; \Gamma^-_{de} \cup \{goal(\varphi_s, \varphi_f)\} \rangle} \quad P^{\text{on}}_F$$

The rule P^{on}_F says when an action act is generated for an FPP $\langle \mathcal{B}, \varphi_s, \Lambda \rangle$ (i.e. $sol^{on}(\mathcal{B}, \varphi_s, \Lambda) = act$), a new declarative goal $goal(\varphi_s, P, \varphi_f)$ is adopted where $P = act; activate(goal(\varphi_s, \varphi_f))$. The procedural component $P = act; activate(goal(\varphi_s, \varphi_f))$ first ensures the pursue of the action act which is returned from online FPP. When the action act is executed, it then calls the FPP again by reactivating the declarative intention $goal(\varphi_s, \varphi_f)$ via a construct *activate*. As such, FPP can take the new belief into consideration and plan for the next action. These two interleaved planning and execution will be repeated until the success condition is achieved if all possible.

The construct $activate(goal(\varphi_s,\varphi_f))$ in the rule P_F^{on} shares the same spirit with the motivation rules in the motivational library regarding the reactivating purpose. However, unlike that motivation rules which are conditioned on beliefs, the construct $activate(goal(\varphi_s,\varphi_f))$ will immediately activate the suspended declarative intention once it is encountered. Therefore, we have the following derivation rule to specify the behaviour of construct $activate(goal(\varphi_s,\varphi_f))$.

$$\frac{P \in \Gamma^+_{pr} \quad P = activate(goal(\varphi_s,\varphi_f)) \quad goal(\varphi_s,\varphi_f) \in \Gamma^-_{de}}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{pr}, \Gamma^-_{de}\rangle \to \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{pr} \setminus \{P\}, \Gamma^+_{de} \cup \{goal(\varphi_s,\varphi_f)\}\rangle} \; Re_{\mathrm{de}}$$

We have discussed how a BDI agent can execute the planning solution generated from FPP to recover the execution failure by addressing the related pure declarative goals. To readily exploit the existing semantics of CAN, we also encapuslate the planning solution within a new declarative goal (e.g. $goal(\varphi_s, act_1;\ldots;act_n,\varphi_f)$) which shares the same success or failure condition as the related pure declarative goal (e.g. $goal(\varphi_s,\varphi_f)$). However, it is still possible that the actions generated by FPP may be blocked, in particular for offline planning due to the dynamic environment. According to the rule $R_{\texttt{fail}}$ in our declarative execution failure recovery strategy in Section 4.3.2, if the procedural component P is blocked in $goal(\varphi_s,P,\varphi_f)$, the agent will trigger FPP again to plan to achieve φ_s by adding $goal(\varphi_s,\varphi_f)$ into the active declarative goal again. Therefore, a complete loop is reached by allowing the BDI to re-plan if the previous planning failed. Effectively, this approach creates a blind agent which will continue to pursue an intention until it believes the intention has actually been achieved.

However, such a blind agent is not always desirable, particularly given that the resource is bounded. Therefore, we also provide an alternative type of the agent which will disallow FPP recovering the actions generated by itself at some point. The rationale for this approach is threefold. First, the purpose of FPP is to recover the plans pre-defined by the agent developers in the first place. Secondly, despite the potential benefits of recovering the plan solution, such an approach could give rise to a huge behaviour space for BDI agents with a considerable cost in terms of planning. Thirdly, subject to the specific domains, the agent should know when it is time to cease pursuing a goal after a considerable effort has been made. For the purpose of legibility, in this thesis, we present a rule which overides the rule $R_{\texttt{fail}}$ and simply disallows FPP keeping recovering the failed actions generated by FPP (i.e. act^{FPP}).

$$\frac{P = goal(\varphi_s, act^{FPP}; P', \varphi_f) \in \Gamma^+_{pr} \quad goal(\varphi_s,\varphi_f) \in \Gamma^-_{de} \quad \langle \mathcal{B}, \mathcal{A}, act^{FPP}\rangle \xrightarrow{fail} \langle \mathcal{B}, \mathcal{A}, ?false\rangle}{\langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{pr}, \Gamma_{de}\rangle \to \langle \Pi, \Lambda, \mathcal{B}, \mathcal{A}, \Gamma_{pr} \setminus \{P\}, \Gamma_{de} \setminus \{goal(\varphi_s,\varphi_f)\}\rangle} \; Dis$$

The rule Dis says that when an FPP-generated action (i.e. act^{FPP}) is blocked for a pure declarative goal (i.e. $goal(\varphi_s,\varphi_f)$), it will discard the entire declarative goal which contains this blocked action (i.e. $\Gamma_{pr} \setminus \{P\}$ where $P = goal(\varphi_s, act^{FPP}; P', \varphi_f)$) and drop such a pure declarative goal from the declarative intention (i.e. $\Gamma_{de} \setminus \{goal(\varphi_s,\varphi_f)\}$). In fact, this rule includes the case of both online planning and offline planning. To be precise, $P' = activate(goal(\varphi_s,\varphi_f))$ in online planning while P' is the remaining actions generated by FPP (if any) in offline planning case.

Finally, we stress that the rule Dis is an extreme case when the agent does not allow FPP to recover the plan solution generated by FPP itself at all. A straightforward extension of the rule Dis can be obtained by allowing FPP to recover the plan solution generated by FPP itself up to a certain amount of times (which is the domain-specific). In practice, it is feasible to realise this more fine-grained control if an agent can keep track of the amount of planning for a certain pure declarative. However, it suffices for us to present the rule Dis from a theoretical point of view of integrating planning in BDI agents. Furthermore, we also point out that our approach of disallowing recovering plan solution in rule Dis only stops the effort of recovering the failure by FPP inclusively. In the case of procedural execution failure in Section 4.3.1, the agent still keeps the suspended procedural intention whose procedural failure amounts to the planning for the related pure declarative goal in the first. In the case of declarative execution failure, the agent still keeps the procedural intention which follows the blocked declarative goal. In fact, the motivation rule is still there intact. For example, once the motivation rule is activated (e.g. a desirable environment change), the agent can still continue to pursue the suspended procedural intention according to the rule M_{pr} in the case of procedural execution failure.

4.5 Formal Relationship between FPP and BDI

In this section, we formally study the relationship between FPP and the BDI execution. The following theorem establishes the link between $goal(\varphi_s, \varphi_f)$ and FPP in both offline and online setting so that $goal(\varphi_s, \varphi_f)$ can – to some extent – be seen as FPP. Recall that an offline (resp. online) solution for an FPP problem $\langle \mathcal{B}, \varphi_s, \Lambda \rangle$ is denoted as $sol^{off}(\mathcal{B}, \varphi_s, \Lambda)$ (resp. $sol^{on}(\mathcal{B}, \varphi_s, \Lambda)$).

Theorem 1. *For any agent,*

1. *For offline planning, we can have the transition* $\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, \varphi_f) \rangle \xrightarrow{*} \langle \mathcal{B}'', \mathcal{A}'', nil \rangle$ *if and only if* $sol^{off}(\mathcal{B}, \varphi_s, \Lambda) = act_1; \ldots; act_n$ *and* $\langle \mathcal{B}, \mathcal{A}, act_1; \ldots; act_n \rangle \xrightarrow{*} \langle \mathcal{B}'', \mathcal{A}'', nil \rangle$ *such that* $\mathcal{B}'' \models \varphi_s$*, provided there is no intervention from the outside environment and other concurrent intentions. The BDI agent can evolve a pure declarative goal* $goal(\varphi_s, \varphi_f)$ *to an empty program* nil *as long as the offline FPP returns a solution, namely* $act_1; \ldots; act_n$ *which can be successfully executed to solve the FPP problem* $(\mathcal{B}, \varphi_s, \Lambda)$.

2. *For online planning,* $\langle \mathcal{B}_0, \mathcal{A}, goal(\varphi_s, \varphi_f) \rangle \xrightarrow{*} \langle \mathcal{B}_k, \mathcal{A} \cdot act_1 \cdot \ldots \cdot act_k, nil \rangle$ *with* $k \geq 0$ *if and only if there exists a solution for each online planning, i.e.* $sol^{on}(\mathcal{B}_0, \varphi_s, \Lambda) = act_1$, $sol^{on}(\mathcal{B}_1, \varphi_s, \Lambda) = act_2$, $\cdots$, *and* $sol^{on}(\mathcal{B}_{k-1}, \varphi_s, \Lambda) = act_k$ *such that* $\langle \mathcal{B}_{j-1}, \mathcal{A} \cdot act_1 \cdot \ldots \cdot act_{j-1}, act_j \rangle \rightarrow$ *for* $j \in \{1, \cdots, k\}$ *and* $\mathcal{B}_k \models \varphi_s$*. The BDI agent will successfully execute (i.e. will make the success condition* φ_s *true) and evolve a pure declarative goal* $goal(\varphi_s, \varphi_f)$ *to* nil *if* φ_s *can be achieved after the repetition of planning and execution.*

Proof.
- The proof of *(i)* relies on the derivation rule P_F^{off}. Recall that rule P_F^{off} says that a successful progression of a pure declarative goal is to generate a solution for execution

to achieve the success condition in this pure declarative goal. In offline planning setting, the transition $\langle \mathcal{B}, \mathcal{A}, goal(\varphi_s, \varphi_f)\rangle \xrightarrow{*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$ first implies that the $goal(\varphi_s, \varphi_f)$ is progressable, i.e. there exists a complete sequence of actions generated from FPP (i.e. $sol^{off}(\mathcal{B}, \varphi_s, \Lambda) = act_1; \ldots; act_n \neq \emptyset$). Secondly, since the pure declarative goal can be successfully drop (i.e. nil) and we have assumed no external environment or concurrent intention interventions, then the returned FPP solution must be successfully executed (i.e. $\langle \mathcal{B}, \mathcal{A}, act_1; \ldots; act_n\rangle \xrightarrow{*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$) to achieve the goal state (i.e. $\mathcal{A}'' \models \varphi_s$). Hence, the right deduced from the left is proved. Let us now prove from right to left. If $sol^{off}(\mathcal{B}, \varphi_s, \Lambda) = act_1; \ldots; act_n \neq \emptyset$ holds, then $goal(\varphi_s, \varphi_f)$ can be progressed. Also if $\langle \mathcal{B}, \mathcal{A}, act_1; \ldots; act_n\rangle \xrightarrow{*} \langle \mathcal{B}'', \mathcal{A}'', nil\rangle$ holds, then the successful should be achieved, given the assumption of no external environment or concurrent intention interventions. Therefore, the pure declarative goal $goal(\varphi_s, \varphi_f)$ can be dropped with success. Combining these two, the left deduced from the right is proved. Therefore, the equivalence between the left and the right holds.

- The proof of *(ii)* can be given similarly as *(i)* but depending on the rule P_F^{on} instead. In detail, we present its proof by induction on the planning step k. So if $k = 0$, then $act_1 \cdot \ldots \cdot act_k = \emptyset$. It means that $\langle \mathcal{B}, \mathcal{A}, (goal(\varphi_s, \varphi_f))\rangle \not\longrightarrow$ is true if $sol^o(\mathcal{B}_0, \varphi_s, \Lambda) = \emptyset$, which holds trivially. Therefore, *(ii)* holds. Next, suppose the claim holds for all numbers less than some $k \geq 1$. We show that *(ii)* holds for k. Since we have, by the hypothesis, that there exists a solution $act_2 \cdot act_3 \cdots act_k$ such that $\langle \mathcal{B}_{j-1}, \mathcal{A} \cdot act_1 \cdot \ldots \cdot act_{j-1}, act_j\rangle \rightarrow$ for $j \in \{2, \cdots, k\}$ and $\mathcal{B}_k \models \varphi_s$ iff $\langle \mathcal{B}_2, \mathcal{A}, goal(\varphi_s, \varphi_f)))\rangle \xrightarrow{*_{k-1}} \langle \mathcal{B}_k, \mathcal{A} \cdot act_2 \cdot act_3 \cdots act_k, nil\rangle$ where $*_{k-1}$ stands for the $k-1$-step closure transition. Clearly, we now only need to discuss the transition from $\langle \mathcal{B}_0, \mathcal{A}, goal(\varphi_s, \varphi_f)\rangle$ to $\langle \mathcal{B}_1, \mathcal{A} \cdot act_1, goal(\varphi_s, \varphi_f)\rangle$. According to the rule P_F^{on} , the transition from $\langle B_0, \mathcal{A}, goal(\varphi_s, \varphi_f)\rangle$ to $\langle B_1, \mathcal{A} \cdot act_1, goal(\varphi_s, \varphi_f)\rangle$ can be obtained in these three steps, (1) adding act_1 for execution and the activate structure for $goal(\varphi_s, \varphi_f)$, and suspending $goal(\varphi_s, \varphi_f)$; (2) executing act_1; (3) activate $goal(\varphi_s, \varphi_f)$. If act_1 is a solution of FPP problem $\langle \mathcal{B}, \varphi_s, \Lambda\rangle$ for achieving the goal, then the problem is apparently solved already. In this case, the goal $goal(\varphi_s, \varphi_f)$ will be dropped immediately according to the rule G_{drop} as φ_s holds. Hence *(ii)* holds. If not, taking into consideration the hypothesis induction applying from 2 to k, *(ii)* holds still. Therefore, by induction, we have proved *(ii)*.

■

This theorem underpins the theoretical foundation that a successful execution resulting from our operational rules P_F^{off} and P_F^{on} for the pure declarative goal $goal(\varphi_s, \varphi_f)$ corresponds directly to a sequence of actions from FPP. On the one hand, *(i)* shows that if an offline planning step is able to start executing, then there is one solution for this FPP problem, provided there is no intervention from other concurrent intentions of the BDI agent and the external environment. In a sense, the offline planning in BDI is local as the possible interplay with the external environment

and other concurrent intentions are not taken into consideration. In contrast, *(ii)* avoids the local nature of offline planning and handle potential interactions while planning. Such online planning is integrated well with the – desirable – reactive nature of the BDI agents. It allows the agent to function in domains where the offline plan can be brittle due to the fast-changing environment. In the following section, we will provide the results on the practical feasibility of embedding FPP in BDI agents, which confirms and complements the theoretical results given above.

4.6 Feasibility Study

In this section, we demonstrate the practical feasibility of integrating a BDI agent system with FPP. We show how the cleaning task scenario from the introduction in Section 4.1 can be expressed using our CAN(FPP) framework. Without the loss of generality and for the simplicity of discussions, we consider the offline FPP and assume that the environment is dynamic (i.e. exogenous events can occur) and deterministic (i.e. the effects of actions can be precisely predicted). We stress though that the purpose of this discussion is not to present an actual fully developed CAN(FPP) system, but rather to motivate the merits of the proposed framework to warrant future work on a fully implemented system. Therefore, we briefly discuss a prototype system which we designed to verify the feasibility of our approach as a basis for this future work.

```
1   // Initial beliefs
2
3   dirty(hall)
4   location(lounge)
5   open(door1)
6   open(door2)
7   open(door3)
8   connect(door1, lounge, hall)
9   connect(door2, lounge, backyard)
10  connect(door3, backyard, hall)
11
12  // Initial goals
13
14  !clean(hall)
15
16  // Plan library
17
18  +!clean(X) : dirty(X) & location(X) <- vacuum(X); ? not dirty(X)
19
20  +!clean(X) : dirty(X) & location(Y) & connect(D, Y, X) & open(D)<- goal(at(X), move(D, Y,
X), nil); ? location(X); vacuum(X); ? not dirty(X)
```

Figure 4.4: BDI Agent in Domestic Cleaning Scenario

We recall that in a cleaning task scenario in Figure 4.1, a robot finished cleaning in the `lounge`, and needs to proceed to the `hall` to vacuum. There is a door labelled as `door1` connecting the `lounge` and the `hall`. The straight-forward route to the `hall` is to go through `door1` when

it is open. There are also two doors, namely door2 and door3 which connect the backyard with the lounge and the hall, respectively. The design of this robot has been shown by its belief base, initial goal and plan library in Figure 4.4. The initial beliefs of the robot are described on lines 3-10 and the initial goal to clean the hall is displayed on line 14 of Figure 4.4. In this case, the achievement goal !clean(hall) is added to the event set of the robot as an external event. At this point, two plans in the plan library on lines 18-20 are stored as plans P_1 and P_2, and BDI agent reasoning cycle begins. Both of plans P_1 and P_2 are relevant plans for the event !clean(hall). After validating and unifying the precondition given the current belief base, plan P_2 (see line 20) is identified as an applicable plan and becomes an intention in the procedural intention Γ_{pr} adopted for the execution. The execution of the body of P_2 starts from the execution of an ordinary declarative goal goal(at(hall), move(door1,lounge,hall), nil) which purses action move(door1,lounge,hall) to achieve the successful state at(hall) with empty failure condition nil. However, it is realistic to expect in a real life setting that some situation can block the execution of the robot (i.e. exogenous events can occur). For example, in a scenario where the door1 was slammed shut unexpectedly (i.e. open(door1)) amidst the execution of the action move(door1,lounge,hall). As a consequence, the action of move(door1,lounge,hall) would be undesirably halted, thus eventually causing the failure of the whole cleaning task.

To address this problem, the derivation rule $R_{\texttt{fail}}$ in Section 4.3.2 will elevate the pure declarative goal goal(at(hall), nil) into the declarative intention Γ_{de} with nil being no failure condition specified and suspend the rest of programs. Semantically, an FPP problem $\langle \mathcal{B}, \texttt{at(hall)}, \Lambda \rangle$ for goal(at(hall), nil) is to indicate that a first-principles planner will be triggered to generate a sequence of actions from the action library Λ to achieve the successful state at(hall) from the initial belief state $\mathcal{B}$. When a sequence of actions $act_1;\ldots;act_n$ is successfully generated in the offline fashion, the BDI agent starts to execute actions in $act_1;\ldots;act_n$ in turn in order to reach a goal at(hall). The goal is achieved if and only if $\langle \mathcal{B}, \mathcal{A}, act_1;\ldots;act_n \rangle \xrightarrow{*} \langle \mathcal{B}'', \mathcal{A} \cdot act_1;\ldots;act_n, nil \rangle$ such that $\mathcal{B}'' \models \texttt{at(hall)}$, provided without interplay with other concurrent intentions of the agent and the external environment. In practice, the BDI agent will need to pass along the successful state at(hall) it wants to achieve, the current belief $\mathcal{B}$, and a set of action Λ to the first-principles planner when calling the planner. We choose an offline first-principles planner called Fast-Forward planner[1] and employ the Planning Domain Definition Language (PDDL) [MGH+98] for specifying planning problems for the first-principles planner in this concrete example. Due to the syntactic knowledge difference, the transformation of knowledge (e.g. predicate, belief, and action) between BDI and PDDL is required to be conducted to generate PDDL planning problem specification. Afterwards, the first-principles planner deliberates and generates a plan solution if all possible. Finally, a sequence of actions is returned from the planner to reach the successful state at(hall), denoted as move(door2, lounge, backyard); move(door3, backyard, hall). It states the robot can move to the backyard through the door2

[1] https://fai.cs.uni-saarland.de/hoffmann/ff.html

first and proceed to the `hall` through the `door3`. The route is depicted pictorially in Figure 4.1. After arriving at the `hall`, we can see that the rest of agent programs will be resumed according to the derivation rule $R_{\texttt{fail}}$ and can indeed be progressed with success. For example, the test goal `?location(hall)` will be confirmed with success.

This case study on the blocked plan-body program highlights a number of key benefits offered by the CAN(FPP) systems. Compared to classical BDI agent, we are able to improve the robustness of the BDI agent systems to tackle the problems beyond their current reach (e.g. due to incomplete plans and dynamic environment). Compared to a pure FPP, our formal framework ensures maximums reactiveness for most of the subgoals (tracked in the procedural intention Γ_{pr}) and only plans on-demand for the pure declarative goals in the declarative intention Γ_{de}.

4.7 Conclusions

In Chapter 1, we emphasised how we wanted to look at planning extension of BDI agent systems. The BDI framework, by itself, is used to model intelligent agents in complex domains. However, by relying on a set of pre-defined plans, it exclusively limits the autonomy and applicability of the resulting agents, particularly when the execution failure occurs. To this end, we introduced CAN(FPP) framework with a strong theoretical underpinning for integrating first-principles planning (FPP) within BDI agent systems based on the intrinsic relationship between the two.

It is not a new idea to combine the power of planning with BDI to increase the robustness of the resulting agents as discussed in Section 3.1 and Section 3.2 in Chapter 3. However, most of the current approaches address the integration in a rigid and ad-hoc fashion. As such, while often improving the performance of the BDI agents, these approaches fail to utilise the full potential of FPP and overlook the relevant theoretical underpinnings of FPP in the existing BDI agents. This is particularly important to an agent which already has a solid theoretical basis.

In this chapter, we introduced a formal operational semantics that incorporates FPP, and that lends power to BDI agents when the situation calls for it. We do this by extending the CAN language (which is a classical and popular BDI programming language), and providing it with novel operational semantics to handle a tight integration with FPP. As such, a BDI agent can accomplish the goals beyond its own pre-defined capabilities. We start with enumerating all types of execution which are deemed either as a failure, or simply ignored by the current BDI but may cause some problems later on. These types of execution failure limit the robustness of the resulting agent when it may have the knowledge to recover them by generating new plans. Therefore, the integration and employment of FPP in BDI framework is naturally motivated. We see that each type of execution failure can be precisely captured by the derivation rule at the intention level. To efficiently manage the existing programs in BDI and FPP, we also introduce the concept of declarative intention, which is used by FPP, and it extends the intentions in BDI with new declarative intentions. With the newly extended intention set and a list of execution

failure captured by precise derivation rules, we provide the recovery strategy for each type of execution failure and articulate how BDI manages FPP for its best interests. To strengthen the theoretical foundation of the BDI agents, we have also established a theorem that the principled integration between FPP and BDI execution is indeed the one intuitively expected both in offline and online planning.

To conclude, we have considered a formal embedding of FPP in the popular BDI agent framework. By doing so, the BDI agents can build new plans when needed to achieve its designed objectiveness. We believe the work presented here lays a firm foundation for augmenting the range of behaviours of the agents by expanding the set of BDI plans available to the agent from FPP. More importantly, this contribution is a significant step towards incorporating different types of advanced planning techniques into BDI agent systems in a principled manner. In the next chapter we will advance the state-of-art of the hybrid planning BDI agents by proposing a novel BDI plan library evolution architecture to improve the adaptivity of the BDI agents which operates in a fast-changing environment. To achieve this, we introduce the plan library expansion and contraction scheme. The plan library expansion is to adopt new plans generated from the first-principles planner for future reuse. The contraction scheme is accomplished by defining the plan library contraction operator regarding the rationality postulates to remove undesirable plans (e.g. obsolete or incorrect plans).

CHAPTER 5

EQUIPPING BDI AGENTS WITH ADAPTIVE PLAN LIBRARY

The bulk of this chapter has been published online in [XBML18b].

5.1 Introduction

In the previous chapter we looked at an extension of Belief-Desire-Intention (BDI) that allows for the inclusion of a planning capability to recover the execution failure by generating new plans on demand. We discussed how First-principles Planning (FPP) could recover each situation of execution failure specified by a clear derivation rule. We also illustrate how BDI agents manage both its existing part of agent programs and the newly embedded FPP in a systematic manner. Such a line of extension increases the robustness of BDI agents by combining the power of planning on-demand to generate new plans. However, we did not look at how an agent can incorporate these new plans (e.g. from external planning tools) for future use and potentially delete some of its old plans to adapt to a changing environment.

Indeed, the intelligent agents should be able to adapt to a changing environment. The current approaches to implement BDI agents are not able to do so because the plan libraries of BDI agents are fixed and pre-defined. Nevertheless, real-world environments do often change over time, and realistic environments can be non-deterministic. Such non-deterministic nature of the environment makes it particularly difficult for an agent designer to foresee all eventualities. Hence, it is almost impossible to create plans in advance to deal with all obscure situations. When an intelligent agent ends up in a situation where its pre-defined response is inadequate or incorrect because the environment changed, or in a situation that was not foreseen at design time, it should be able to augment the range of behaviours (i.e. the BDI plans) in order to cope with changes in the environment. Furthermore, the plans can also become pro-error or no longer

useful, given that the environment can change over time. When some plans become unsuitable for the current state of the environment, the agent should be able to discard them for adaptivity.

To illustrate the problem, consider the example of a Mars Rover exploring the surface of Mars. The Rover, which is pre-designed with tasks (e.g. carrying out science experiments), must utilise a high degree of autonomy due to the high-latency communication channels to Earth. For example, when an off-nominal event (e.g. a blue rock) is detected, it would increase science exploration if the Rover can respond to such "science opportunities". This autonomous behaviour implies that new plans (e.g. navigation plan) may need to be generated. Furthermore, since the Rover will always return to the lander to deliver samples to the ascent vehicle, this lends an opportunity to the Rover to (potentially) use the knowledge obtained during navigation to a sampling site on its return. If the Rover has safely navigated to a location, "remembering" the route it took (i.e. adding such a navigation plan to its plan library) and then returning by the same path are promising features highlighted in [BMH08]. However, some of these "*remembered*" plans may fail because of the fast-changing Martian surface. Therefore, the future planetary Rovers demand more adaptive agent systems which can both add and delete plans intelligently.

However, we can see that most of these planning extensions overlooked the potential adoption of new plans generated by FPP, and continue to treat the plan library as a fixed and pre-defined set of plan rules. For example, to compensate for the inadequacy of a plan library in an uncertain environment, the authors of [BMH+16] proposed the AgentSpeak+ framework, which extends AgentSpeak with a mechanism for probabilistic planning named Partially Observable Markov Decision Processes (Partially Observable Markov Decision Processes (POMDPs)) [KLC98]. However, once the goal was met, the (potentially valuable) plan obtained from the POMDPs was simply forgotten and discarded. Other promising works, such as [DI99, ML07, SSP09], proposed the integration of classical planners and BDI agents to generate new plans. Unfortunately, none of them considers expanding the set of pre-defined plans by adopting these useful and hard-fought new plans generated from external planning tools. There is one work considering the reuse of new plans (achieved by adding them to the plan library) found in [ML08]. Still, this work solely focuses on leveraging new plans by deriving optimal context conditions, thus approaching the plan library expansion in an ad-hoc manner.

In this chapter, we design BDI agents which can adapt to the changing environment by dropping the assumption of fixedness of the pre-defined plan library for the mainstream of BDI agents. To do so, we investigate the structure of a pre-defined plan library and define a generic framework that enables a BDI agent to incorporate new plans from, e.g. automated planning tools for unforeseen situations. We will refer to this step as plan library expansion. However, merely adding plans is not enough for an agent. As the agent ages, some plans may become unsuitable, hampering its reactive nature which is crucial to the success of BDI agents. To illustrate such a case, an approach to an event (e.g. the need to enter another room) which worked in the past (e.g. turning a handle) may no longer work in the future (e.g. the handle has been removed, and

a button needs to be pressed instead). Therefore, there is a need for plan library contraction as well. Indeed, plan library contraction is an altogether more significant – albeit challenging – problem than expansion. Unlike plan library expansion which adds plans because of the need, plan library contraction needs to determine which plans are no longer deemed valuable and so can be removed without damaging the internal structure of the plan library. Therefore, it relies on both qualitative and quantitative measures associated with each plan in the library. For example, a plan may be flagged for the potential deletion because it became obsolete (e.g. a low number of calls) or because it became incorrect (e.g. a high failure rate). However, due to the nature of a plan library, care must be taken when deleting plans to avoid undesirable side-effects. For instance, it could damage the agent more to delete an incorrect (sub)plan which is relied upon by another highly successful plan.

To achieve these objectives in this chapter, we follow a principled approach to a plan library expansion and contraction, motivated by postulates that clearly highlight the underlying assumptions, and supported by measures which are able to characterise plans in the plan library. The contributions of this chapter are, therefore, threefold. Firstly, we provide a systematic specification of domain-independent characteristics (e.g. the quality of plans) of the plan library as the basis for the reasoning of the plan library expansion and contraction. Secondly, we define a plan library expansion operator and formally shows the benefits of expansion regarding relevant characteristics. Thirdly, we introduce an operator for plan library contraction which takes the earlier characteristics into account, and which balances the need for reactivity, the fragility of the plan library, and the correctness and overall performance of the agent. To demonstrate the feasibility of the proposed plan library contraction operator, we present a multi-criteria argumentation-based decision making to instantiate a contraction operator exemplified in a planetary vehicle scenario.

The remainder of this chapter is organised as follows. In Section 5.2, we introduce a set of measures that characterise the performance and structure of plans. These new measures are intuitive and domain-independent. Together, they form a set of binary relations as the basis to construct the plan library expansion and contraction operator. In Section 5.3, we give a principled definition of plan library expansion operator with some postulates and shows that the expansion of a plan library will only improve the overall performance of the resulting BDI agents. Indeed, as we will see, the expansion of a plan library will never cause a decrease of the number of goals an agent can respond to, and the number of relevant plans it has to address a given goal. Finally, in Section 5.4, we propose some postulates for a plan library contraction operator. In addition, we also present a concrete instantiation of such a contraction operator in a Mars Rover scenario and prove that this instantiation indeed is a contraction operator.

5.2 Plan Library Analysis

In this section, we establish some measures to capture the characteristics of plans (e.g. the performance of plans, and the relationships between them) in a BDI agent system. This section will provide the foundations for understanding both how to compute them, and how they can be used for the plan library expansion in Section 5.3 and contraction reasoning in Section 5.4.

5.2.1 Measuring Performance of Plans

In this chapter, we use $\mathcal{P}$ to stand for a set of plans and $\mathcal{T}$ a set of time points. We start by introducing notations for plan execution as follows:

Definition 1. A function $\mathcal{S} : \mathcal{P} \times \mathcal{T} \to \{\top, \bot, \varnothing\}$ is called a status function.

A status function records the success and failure of plans during agent execution while the agent is running. For example, $\mathcal{S}(P_1, 3) = \top$ means that plan P_1 succeeded at time point 3 while $\mathcal{S}(P_2, 5) = \bot$ says that plan P_2 failed at time point 5. Finally, $\mathcal{S}(P, t) = \varnothing$ if it didn't succeed or fail at time point t (e.g. still in execution). In principle, the success (denoted as $\top$) or failure (denoted as $\bot$) of a plan is determined by its primitive actions as the actions are ultimate means to interact with the environment. In this chapter, we also assume that an action fails if it cannot be executed or side-effects taking place after execution, which agrees with the type of precondition failure and effect failure discussed in Section 4.2. Therefore, in order for a plan P to succeed, all primitive actions in P need to succeed. Otherwise, we can say that the plan P fails.

We now introduce the execution frequency of plans, which measure how many times a plan has been completely executed over a given set of time points, and success rate, which is based on the execution frequency to capture the relative performance quality of each plan.

Definition 2. An execution frequency function $\delta : \mathcal{P} \times \mathcal{T} \times \mathcal{T} \to \mathbb{N}$ is defined for each $P \in \mathcal{P}$ and each $t_1, t_n \in T$ such that $t_1 \leq t_n$ as follows:

$$\delta(P, t_1, t_n) = |\{\mathcal{S}(P, t_i) \neq \varnothing \cdot \; i = 1, \cdots, n\}|$$

Definition 3. A success rate for plan P is defined as:

$$\Phi(P, t_1, t_n) = \frac{\delta^s(P, t_1, t_n)}{\delta(P, t_1, t_n)}$$

where $\delta^s(P, t_1, t_n) = |\{\mathcal{S}(P, t_i) = \top \cdot \; i = 1, \cdots, n\}|$ stands for the successful *execution frequency*.

In Definition 2, the execution frequency simply collects the number of times a plan has led to either success or failure over a given set of time points between t_1 and t_n. Furthermore, to avoid confusion when combined with the set cardinality, we use $\cdot$ instead of | in $\{\mathcal{S}(P, t_i) \neq \varnothing \cdot \; i = 1, \cdots, n\}$ to denote the set of t_i ranging from 1 to n such that $\mathcal{S}(P, t_i) \neq \varnothing$. The similar expression is adopted throughout this chapter when necessary. In Definition 3, the success rate is the percentage of the

execution frequencies that are successful over a given set of time points. We stress that argument $\mathcal{T} \times \mathcal{T}$ allows the agent to have a capture of the quality of plans which can be based on both the overall performance (i.e. $\delta(P, t_0, t_n)$) and the latest performance (e.g. $\delta(P, t_{n-2}, t_n)$). Such a timely capacity is vital as the realistic environment is highly dynamic. Therefore, simply having the overall success rate may prevent the agent from being aware of the recent abruptly growing failure of some plans.

5.2.2 Relationships Between Plans

We have introduced execution frequency and success rate to provide a performance abstraction to plans in BDI agents. While they are beneficial, however, it says nothing about the inherent structural characteristics (i.e. the relationships between plans) of these plans. Let e^P be a set of relevant plans $\{P_1, \cdots, P_n\}$ for achieving an event e. The first thing we are interested in is to know, and to compute the relevancy of individual plan P, i.e. how many alternative relevant plans a BDI agent possesses to respond to an event e in different situations.

Definition 4. A relevancy function $\Upsilon : \mathcal{P} \to \mathbb{N}$ is defined to be $\Upsilon_{\mathcal{P}}(P) = |e^P| - 1$ where $P \in e^P \subseteq \mathcal{P}$.

Definition 4 defines a relevancy measure of a plan to the number of the relevant plans of the event it responses minus one. For example, when there is one plan P to address an event e, we can see that the relevancy of P is 0 (i.e. no other relevant plan for addressing e except for itself).

Besides the concept of relevancy, we are also interested in replaceability, which is when there are two or more plans applicable in the same situation to get the same result (i.e. post-effects). Intuitively, the "greater" the replaceability of a plan P is, the higher the chance that such a plan P can be recovered in the event of its failure, thus providing flexibility and robustness to the whole system. To introduce the concept of what it means by being replaced and the degree of replaceability, we reply on the concept of overlapping in work of [TSP12] and the concept of summarisation in [YSL16]. In a nutshell, the overlap of P and $\{P_1, \cdots, P_n\}$, denoted as $\mathcal{O}(\{P, P_1, \cdots, P_n\})$ in [TSP12], measures the number of situations (i.e. possible worlds) that both P and $\{P_1, \cdots, P_n\}$ can be applicable. It tells whether two or more plans can be applicable in the some same situations. For example, $\mathcal{O}(\{P, P_1, \cdots, P_n\}) \neq 0$ shows that the situation for both P and a set of plans $\{P, P_1, \cdots, P_n\}$ to be applicable exists. Meanwhile, the summarised post-effects of a plan P (i.e. $post(P)$ denoted in [YSL16]) provides a means to check if some plans can achieve the same result regarding the necessary and possible post-effects. The summarised necessary post-effects are those which are always true after successfully executing any decomposition of plans while possible post-effects are those that may result from some decomposition of plans. We use $post(\{P_1, \cdots, P_n\}) \models post(P)$ to represent that the post-effects of executions of $S = \{P_1, P_2, \cdots, P_n\}$ can ensure the post-effects of executions of P to be true. Therefore, we can have the following definition of what it means being replaceable for a plan P and the measure of the degree of replaceability of a plan P.

Definition 5. A plan P can be replaced by a set of plans $S = \{P_1, P_2, \cdots, P_n\}$ where $P_i \neq P$ ($i \in \{1,\ldots,n\}$), denoted as $P \rhd_r S$, if the overlap of P and $\{P_1,\cdots,P_n\}$, namely $\mathcal{O}(\{P,P_1,\cdots,P_n\}) \neq 0$ and $post(\{P_1,\cdots,P_n\}) \models post(P)$.

The concept of replaceability reveals whether a given plan P can be replaced by a set of different other plans S. Furthermore, it can be seen that if $P \rhd_r S$, then $P \rhd_r S'$ should naturally hold for any S' such that $S' \supseteq S$. As such, there is an indefinite number of the set of plans to replace a given plan P as long as there is one set of plans to replace P. To ensure a one-to-one mapping to plans to their replaceability, therefore, for any given plan P, we are interested in a particular set of plans S such that $P \rhd_r S$ and $\forall S'' \subsetneqq S$, $P \not\rhd_r S''$. Of course, $P \not\rhd_r (S \setminus P')$ holds when either $\mathcal{O}(\{P\} \cup S'') = 0$ or $post(S) \not\models post(P)$. To abuse the notation, we denote S can *minimally* replace P as $P \rhd_{mr} S$. We are now ready to formally define the degree of replaceability of plans as follows:

Definition 6. A degree of replaceability for plan P is a function $\varsigma_{\mathcal{P}} : \mathcal{P} \to \mathbb{N}$, defined to be $\varsigma_{\mathcal{P}}(P) = |\{S \cdot P \rhd_{mr} S\}|$ where $P \rhd_{mr} S$ stands for that S can minimally replace P.

We can see that the degree of *replaceability* for P is the number of sets of plans S that can minimally replace P in Definition 6. Finally, we close the section by noting that what we have done so far is to define the relevant measures of BDI plans at the individual plan level. In the following section, we will extend the measures for a given set of plans (e.g. a plan library) to prepare for plan library expansion and contraction.

5.2.3 Summary Information

We have defined the performance and structural information for each individual plan in BDI agents. We can now summarise both performance and structural information of an individual plan to characterise the plan library of a BDI agent system as a whole.

Firstly, we describe how the performance information (e.g. execution frequency) of each plan is summarised to indicate the performance of a plan library. We apply a mean aggregation method to provide an average performance of execution frequency and success rate of a plan library.

Definition 7. An execution frequency is a function $\delta : 2^{\mathcal{P}} \times T \times T \to \mathbb{R}_{\geq 0}$ defined as follows:

$$\delta(\Pi, t_1, t_n) = \frac{\sum_{P \in \Pi} \delta(P, t_1, t_n)}{|\Pi|}.$$

where $\Pi \subseteq 2^{\mathcal{P}}$ and $\delta(P, t_1, t_n)$ denotes the execution frequency of plan P between time t_1 and t_n.

Definition 8. A success rate is a function $\Phi : 2^{\mathcal{P}} \times T \times T \to \mathbb{R}_{\geq 0}$ defined as follows:

$$\Phi(\Pi, t_1, t_n) = \frac{\sum_{P \in \Pi} \Phi^s(P, t_1, t_n)}{|\{P \in \Pi \cdot \delta(P, t_1, t_n) \neq 0\}|}$$

where $\Pi \subseteq 2^{\mathcal{P}}$, $\delta(P, t_1, t_n)$ refers to execution frequency of plan P between time points t_1 and t_n, and $\delta(P, t_1, t_n) \neq 0$ means P has to be executed at least once between time t_1 and t_n.

Secondly, we summarise the structural information of a plan library by counting how many events a plan library accounts for. The intuition of it is that the capability of a BDI agent is essentially the number of different types of events or goals it can handle. Therefore, we can define the degree of functionality to formalise such an intuition as follows:

Definition 9. A degree of the functionality is a function $\mathcal{F} : 2^{\mathcal{P}} \to \mathbb{N}$ defined as follows:

$$\mathcal{F}(\Pi) = |\{e_P \cdot P \in \Pi\}|$$

where $\Pi \in 2^{\mathcal{P}}$ and e_P is the triggering event of plan P.

Recall that the measure of relevancy of a given plan quantifies the number of relevant plans of the event it addresses, while the degree of replaceability counts the number of sets of plans which are available to replace such a plan. We now are ready to introduce our four novel ordering relations corresponding to the three summaries we established above (i.e. execution frequency, success rate, and functionality), and one extra ordering relation based on relevancy and replaceability. Such a set of orderings for a set of plans are given as follows:

Definition 10. A set of binary relations $\succeq$ over $2^{\mathcal{P}}$ with regard to execution frequency δ, success rate Φ, functionality $\mathcal{F}$, and relevancy Υ and replaceability ς measure, is a 4-tuple

$$\langle \succeq_{activeness}, \succeq_{success}, \succeq_{functionality}, \succeq_{robustness} \rangle$$

where $\forall \Pi, \Pi' \in 2^{\mathcal{P}}$

- $\Pi \succeq_{activeness} \Pi'$ iff $\delta(\Pi, t_1, t_n) \geq \delta(\Pi', t_1, t_n)$;
- $\Pi \succeq_{success} \Pi'$ iff $\Phi(\Pi, t_1, t_n) \geq \Phi(\Pi', t_1, t_n)$;
- $\Pi \succeq_{functionality} \Pi'$ iff $\mathcal{F}(\Pi) \geq \mathcal{F}(\Pi')$;
- $\Pi \succeq_{robustness} \Pi'$ iff $\nexists P \in \Pi \; s.t. \; P \in \Pi'$, $\Upsilon_{\Pi}(P) \leq \Upsilon_{\Pi'}(P)$, and $\varsigma_{\Pi}(P) \leq \varsigma_{\Pi'}(P)$;

For any binary relation, we have that $\Pi \simeq \Pi'$ if $\Pi \succeq \Pi'$ and $\Pi' \succeq \Pi$ while $\Pi \succ \Pi'$ if $\Pi \succeq \Pi'$ and $\Pi \not\preceq \Pi'$. For a plan library Π, if Π has a higher execution frequency than Π', denoted as $\Pi \succeq_{activeness} \Pi'$, then it is interpreted as that Π is believed to be more active than Π'. The second ordering $\Pi \succeq_{success} \Pi'$ means that Π has a higher success rate than Π', and Π is believed to be more successful than Π'. The third ordering $\Pi \succeq_{functionality} \Pi'$ means that Π can respond to more types of events than Π' can. Finally, the fourth ordering $\Pi \succeq_{robustness} \Pi'$ shows that for every plan $P \in \Pi$, Π has both more relevant and replaceable plans for P than Π' does.

Now that we have defined all relevant measures, we look into how we can expand and contract a plan library sensibly and predictably based on the summaries of a given plan library.

5.3 Plan Library Expansion

In this section, we propose some postulates for the plan library expansion. These postulates rationalise the desired behaviours of our plan library expansion and serve as a premise for further reasoning, e.g. the properties of the proposed plan library expansion scheme. For illustration and simplicity, we will first consider using a single plan to represent inputs, and then extend to the general case where we use any set of plans to represent general inputs.

5.3.1 Formal Expansion Framework

We start with the definition of an expansion operator $\circ$: Given a plan library Π and a plan P, $\Pi \circ P$ denotes the expansion of Π by P with $\circ$ if and only if it satisfies the following postulates:

EO1 $\Pi \circ P$ is a plan library.

This postulate ensures that the expansion is still a plan library.

EO2 $P \in \Pi \circ P$ and $\Pi \subseteq \Pi \circ P$.

This postulate states that the new plan is obtained after the expansion and the result of plan library expansion $\Pi \circ P$ indeed subsumes the knowledge of the previous plan library Π.

EO3 If $P \in \Pi$, then $\Pi \circ P = \Pi$.

This postulate indicates that the plan library expansion $\Pi \circ P$ should only consider a new plan P which is initially not included in Π.

EO4 $(\Pi \circ P) \circ P' = (\Pi \circ P') \circ P$ for any plan P and P'.

This postulate says that the order of inputs should not influence the outcome of expansion.

Proposition 1. If an operator $\circ$ satisfies the postulate **EO1**, **EO2**, and **EO4**, we have $\Pi \circ \{P, P'\} = (\Pi \circ P) \circ P' = (\Pi \circ P') \circ P$.

This proposition shows that the expansion of a set of plans is equivalent to a sequence of expansions by a single plan. In addition, the order of expansion makes no difference either.
Now we can give the following representation theorem for these postulates.

Theorem 2. *Given an operator $\circ$, $\Pi \circ P$ satisfies **EO1-E04** precisely when*

$$\Pi \circ P \succeq_{functionality} \Pi \text{ and } \Pi \circ P \succeq_{robustness} \Pi.$$

This theorem formally confirms that the expansion of a plan library Π by P will never cause a decrease of functionality or robustness. In other words, after an expansion of a plan library, it would only be more events or goals that an agent can respond to, more relevant plans for some goals, and more replaceable plans for some plans.

Finally, in order to extend these postulates to the case that new input is not restricted to only one plan, we simply need to replace a single input plan P with a set of plans $\mathcal{P}$.

5.4 Plan Library Contraction

In this section, we give a principled definition of a plan library contraction operator. We then present a concrete instantiation of such an operator in a Mars Rover scenario. We then close this section by showing that this instantiation satisfies the postulates of a contraction operator.

5.4.1 Formal Contraction Framework

We start with the definition of a contraction operator ∇: Given a plan library Π, $\nabla(\Pi)$ denotes the contraction of Π by ∇ iff it satisfies the following postulates:

CO1 $\nabla(\Pi)$ is a plan library.

This postulate ensures the result of contraction is a plan library.

CO2 $\nabla(\Pi) \subseteq \Pi$.

This postulate says the result of a contraction operator is a subset of the original plan library.

CO3 Given a set of plans $\mathcal{P}$, if $\mathcal{P} \subseteq \Pi \setminus \nabla(\Pi)$ and $\mathcal{P} \subseteq \Pi' \subseteq \Pi$, then $\mathcal{P} \subseteq \Pi' \setminus \nabla(\Pi')$.

This relativity postulate states that if a set of plans $\mathcal{P}$ are contractible in the plan library Π (i.e. $\mathcal{P} \subseteq \Pi \setminus \nabla(\Pi)$), then they must be deemed as contractible in any subset Π' (i.e. $\mathcal{P} \subseteq \Pi' \setminus \nabla(\Pi')$) which includes them (i.e. $\mathcal{P} \subseteq \Pi' \subseteq \Pi$).

CO4 $\nabla(\Pi) \succeq \Pi$ where $\succeq \in \{\succeq_{activeness}, \succeq_{success}\}$.

This postulate restricts the behaviour of the contraction by saying that the contraction $\nabla(\Pi)$ should not witness the decrease of both *execution frequency* and *success rate* of Π.

CO5 $\forall P \in \Pi \setminus \nabla(\Pi)$, then $\varsigma_{\nabla(\Pi)}(P) > 0$.

This postulate takes care of the fragility of the plan library by ensuring that there are still plans left in $\nabla(\Pi)$ which can replace deleted plan P.
With these postulates, we have the following results that characterise contraction operators that satisfy some of postulates **CO1-CO5**.

Proposition 2. If an operator ∇ satisfies the postulate **CO1-CO3**, then the following holds:

(1) $\nabla(\Pi') \subseteq \nabla(\Pi)$ *if* $\Pi' \subseteq \Pi$	ordered set inclusion
(2) $\nabla(\Pi \cap \Pi') \subseteq \nabla(\Pi) \cap \nabla(\Pi')$	intersection set inclusion
(3) $\nabla(\Pi \setminus \Pi') \subseteq \nabla(\Pi) \setminus \nabla(\Pi')$	difference set inclusion
(4) $\nabla(\Pi) \cup \nabla(\Pi') \subseteq \nabla(\Pi \cup \Pi')$	union set inclusion

Proof. The first condition shows that the contraction preserves the set inclusion order. If $(\Pi \setminus \nabla(\Pi)) \cap \Pi' = \emptyset$, then we have $\Pi' \subseteq \nabla(\Pi)$. Given **CO2** (i.e. $\nabla(\Pi') \subseteq \Pi'$), we have $\nabla(\Pi') \subseteq \nabla(\Pi)$. If $(\Pi \setminus \nabla(\Pi)) \cap \Pi' = \mathcal{P} \neq \emptyset$, we have $\Pi' \setminus \mathcal{P} \subseteq \nabla(\Pi)$. Given **CO3**, we have $\mathcal{P} \subseteq \Pi' \setminus \nabla(\Pi')$, which means $\nabla(\Pi') \subseteq \Pi' \setminus \mathcal{P}$. Hence we have $\nabla(\Pi') \subseteq \nabla(\Pi)$. In the second condition, it shows that the contraction on intersection of Π and Π' are a subset of the intersection of the contraction results of $\nabla(\Pi)$ and $\nabla(\Pi')$. Notice $\Pi \cap \Pi'$ is a subset of both Π and Π'. Therefore, $\nabla(\Pi \cap \Pi') \subseteq \nabla(\Pi)$ and $\nabla(\Pi \cap \Pi') \subseteq \nabla(\Pi')$ according to the first condition. Hence $\nabla(\Pi \cap \Pi') \subseteq \nabla(\Pi) \cap \nabla(\Pi')$. Similar arguments can be give to the third and fourth condition by noticing $\Pi \setminus \Pi' \subseteq \Pi$ (i.e. difference set inclusion) and $\Pi' \subseteq \Pi$ in third condition, and $\Pi \subseteq \Pi \cup \Pi'$ (i.e. union set inclusion) and $\Pi' \subseteq \Pi \cup \Pi'$ in fourth condition. ■

5.4.2 Instantiation of Plan Library Contraction

In this section, a concrete multi-criteria argumentation-based decision making is proposed to instantiate the abstract contraction operator presented in Section 5.4.1. We stress though that the purpose of this instantiation is not to signify its supremacy over other potential instantiations, but rather to verify the existence and feasibility of our contraction operator. Also, the benchmark comparison of different instantiated contraction operators is beyond the scope of this chapter.

The multi-criteria argumentation-based decision making is a general-purpose decision framework which combines the multi-criteria decision [OMT07] with knowledge-based qualitative argumentation theory [AP06]. Argumentation serves to support or attack whether a particular candidate is better than another based on knowledge processed by an agent. The framework employed in this work is formally stated in [FEGS14] and is introduced in a self-contained fashion in this section. Whenever there is any potential conflict with the previous terminologies, it will be pointed out to avoid the confusion. The framework in [FEGS14] is conceptually composed by three components, namely $\langle X, \mathcal{K}, \mathcal{R} \rangle$. The first component X is the set of all possible candidates (e.g. a set of plans) presented to the decision maker. The second component is epistemic knowledge $\mathcal{K}$, denoted as a 5-tuple $\langle \mathcal{C}, >_{\mathcal{C}}, \lambda, \varpi, ACC \rangle$. It allows the agent to reason and compare candidates among each other and decide which is/are best candidate/s to be chosen. A set of non-cyclic (i.e. linear) criteria $\mathcal{C}$ (e.g. success rate) is used to compare the elements in X. The strict order of the element of $\mathcal{C}$, denoted as $>_{\mathcal{C}}$, is given such that $(C_i, C_j) \in \mathcal{C}$ means that the criteria C_i is preferred than C_j. In order to quantify the linear preference, each $C_i \in \mathcal{C}$ has a subinterval that states the preference among all criteria (e.g. $C_1 = \textit{execution frequency} \sim [0, 0.25]$). A set of clauses (ζ, μ)

is computed where ζ in the form of $q \leftarrow p_1 \wedge \cdots \wedge p_k$ ($k \geq 0$) says the conclusion q is supported by $p_1 \wedge \cdots \wedge p_k$, where $q, p_1, \cdots, p_k$ are literals, and $\mu \in [0,1]$ which express a low bound for the necessity degree of ζ. A set of uncertain clauses (i.e. $\mu \in (0,1)$) is denoted as λ while the set of certain clauses (i.e. $\mu = 1$) denoted as ϖ. Uncertain clauses with the same conclusion will be combined to form arguments. A user-specified aggregation function ACC aggregates necessity degrees of arguments which support a same conclusion q to build accrued structures. Finally, the decision rules $\mathcal{R}$ will be used to select final candidates, denoted as $sol(\langle X, \mathcal{K}, \mathcal{R}\rangle)$, based on those accrued structures. There are two decision rules[1] shown as follows:

DR1 : $\{W\} \overset{X}{\Leftarrow}$ *{ws(W,Y)}, not{ws(Z, W)}*.

A candidate $W \in X$ will be chosen if W is worse (ws) than another candidate Y and there does not exist Z which is worse than W.

DR2 : $\{W, Y\} \overset{X}{\Leftarrow}$ *{sm(W,Y)}, not{ws(Z, W)}*.

Both W and $Y \in X$, deemed as equivalently bad i.e. same (sm), will be chosen if there is no Z which is worse than W and Y.

Following the methodology of the formalism in [FEGS14], we consider a BDI agent which has a set of plan $\mathcal{P}$ and a set of criteria $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ where $C_1 = \delta(P, t_0, t_{current})$ is the overall execution frequency from initial time point t_0 to current time point $t_{current}$ (i.e. the moment the plan contraction starts), $C_2 = \delta(P, t', t_{current})$ the latest execution frequency of P from a chosen recent time point t' to $t_{current}$, C_3 the overall success rate, and C_4 the latest success rate.

In this chapter, we assume that the agent prioritises the success rate criteria over execution frequency criteria and prefer the latest information. Therefore, the criteria order $>_{\mathcal{C}} = \{(C_4, C_3), (C_2, C_1), (C_4, C_2), (C_3, C_1)\}$. Since plans that are without any replaceable plans cannot be deleted and the formalism in [FEGS14] is not concerned with how the possible candidates are obtained to present to the decision-maker, we will employ postulate **CO5** to filter these plans out. Therefore, we define a specific contraction operator, denoted as ∇^{abm}, and propose a generic algorithm which implements ∇^{abm} as shown in Algorithm 1:

Definition 11. Let $\mathcal{P}$ be a set of plans. We define a contraction operator $\nabla^{abm} = sol(\langle X, \mathcal{K}, \mathcal{R}\rangle)$ where $sol(\langle X, \mathcal{K}, \mathcal{R}\rangle)$ is the selected candidates of decision problem $\langle X, \mathcal{K}, \mathcal{R}\rangle$ defined previously, and $X = \{P \in \mathcal{P} \mid \varsigma_{\mathcal{P}}(P) > 0, C_4(P) < \eta\}$ where η represents the success rate tolerance threshold and $C_4(P)$ the value of criteria C_4 (i.e. latest success rate) of P.

The set of all possible candidates X will not include any plans which do not have replaceable plans (i.e. $\varsigma_{\mathcal{P}}(P) > 0$) and only have plans with success rates lower than η for potential removal (i.e. $C_4(P) < \eta$) shown in step 2 of Algorithm 1. In the following section, all the concepts involved in the multi-criteria argumentation based decision making in [FEGS14] will be exemplified in

[1]We modify the rules to choose the worse (ws) plans compared to the original work on finding the better ones.

Algorithm 1: Computation for Contraction Operator ∇^{abm}

```
1 function ∇^abm(⟨X, C, >_C, R⟩)
2     X = {P ∈ P | ς_P(P) > 0, C_4(P) < η}          /* filtering */
3     C = {C_1, C_2, C_3, C_4} defined previously
4     >_C = {(C_4,C_3),(C_2,C_1),(C_4,C_2),(C_3,C_1)}
5     Compute uncertain and certain clauses (λ, ϖ)
6     Build arguments (defined in [FEGS14])
7     Apply rules R to select the acceptable candidates
8     return solution of selection
```

a Mars Rover example to demonstrate how it can effectively assist the selection of plans for deletion.

5.4.3 Planetary Vehicle Example

In this section, we present an example of planetary vehicle to illustrate the employment of a concrete plan contraction operator based on the multi-criteria argumentation-based decision making. Assume that one of the missions of Mars Rover is to use scientific instruments mounted to the robotic arm of the Rover to investigate and analyse Martian terrain. This requires the Rover to drive up to a designated target (i.e. terrain navigation plan), position themselves to reach the target (i.e. Rover positioning plan), and deploy the arm onto the target to perform the investigation (i.e. arm deployment plan). After remembering several routes from the navigation planner it took to a designated crater wall, the Rover has plans P_1, P_2, and P_3 to navigate to it again if needed. Plan P_4 and P_5 are Rover positioning and arm deployment plan, respectively. Consider the set of plans $\Pi = \{P_1, P_2, P_3, P_4, P_5\}$ where replaceability $\varsigma_\Pi(P_1) = \varsigma_\Pi(P_2) = \varsigma_\Pi(P_3) = |\{P_1, P_2, P_3\}| - 1 = 2$, $\varsigma_\Pi(P_4) = \varsigma_\Pi(P_5) = |\{P_4\}| - 1 = |\{P_5\}| - 1 = 0$, and $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ where C_1 is the overall execution frequency (**oef**), C_2 the latest execution frequency (**lef**), C_3 overall success rate (**osf**), and C_4 latest success rate (**lsr**). We set the lower bound tolerant success rate threshold $\eta = 0.85$.

Table 5.1 shows the possible candidates and their respective values for each criterion (in C_1, C_2, C_3, and C_4) and their respective values normalised to interval $[0,1]$ (in C'_1, C'_2, C'_3, and C'_4). It is noted that both plan P_4 and P_5 have been filtered out due to no replaceable plans available. In other words, plan P_4 and P_5 are protected from being discarded. To briefly explain all criteria in the context, let us have a look at plan P_1. On the one hand, $C_1 = 80$ for plan P_1 means that the plan P_1 has been executed 80 times from the initial time point to the current time point while $C_2 = 70$ for plan P_1 indicates that 70 out of these 80 execution are executed recently, called the latest execution frequency. On the other hand, $C_3 = 0.8$ for plan P_1 shows the success rate of executing P_1 from the initial time point to the current time point is 80% compared to only 50% successful rate in the latest period of time given in $C_4 = 0.5$ for plan P_1. To

obtain the normalisation (for criterion C_1), we have initial value C_1 for plan P_1 is 80, C_1 for P_2 20, and C_1 for P_3 70. After the normalisation, we have C'_1 for plan P_1 is 80/80 = 1, C'_1 for plan P_2 is 20/80 = 0.25, and C'_1 for plan P_3 is 70/80 = 0.88. Similarly, the remaining criteria can be interpreted and their respective normalised values can be obtained.

Candidates	C_1	C_2	C_3	C_4	C'_1	C'_2	C'_3	C'_4
P_1	80	70	0.8	0.5	1	1	1	0.63
P_2	20	5	0.6	0.8	0.25	0.07	0.75	1
P_3	70	10	0.7	0.75	0.88	0.14	0.88	0.94

Table 5.1: Criterion Values and Normalised Criterion Values

We now, following the approach from [FEGS14], we explain in details how a set of uncertain and certain clauses $\langle\lambda,\varpi\rangle$ can be computed presented in Figure 5.1. The necessity degrees of the clauses belonging to (λ,ϖ) were calculated as follows. **Step 1**: Normalise the criteria values (see C_i) to interval $[0,1]$ for all of the criteria (see C'_i) where $i \in \{1,\ldots,4\}$. **Step 2**: Compare the candidates among each other regarding the normalised criteria. The candidate which appears as the first argument has a worse criteria value than the one that appears as the second argument. The necessity degree of the clause is calculated as the absolute value of the remainder of the normalised criteria values. **Step 3**: Divide the necessity degree obtained in the previous step by the number of criteria provided to the decision-maker, i.e. by 4 in this case. Let us take an example of $(oef(P_2,P_1),0.19)$ in Figure 5.1 to explain how these three steps above work. Firstly, step 1 has been accomplished shown in Table 5.1. After comparing plan P_1 and P_2 regarding the criterion overall execution frequency (oef), we have that the first argument is P_2 given its worst criteria value and the second argument P_1. The necessity degree of the clause $oef(P_2,P_1)$ is obtained by dividing the absolute value of the remainder of the normalised criteria values (in the step 2) of P_1 and P_2 by the number of criteria in the step 3, i.e. $\frac{|0.25-1|}{4} = 0.19$. Therefore, we have an uncertain clause $(oef(P_2,P_1),0.19)$. **Step 4**: Assign the subinterval to each criteria according to $>_C$, i.e. $C_1 = oef \sim [0,0.25]$, $C_2 = lef \sim [0.25,0.5]$, $C_3 = osr \sim [0.5,0.75]$, and $C_4 = lsr \sim [0.75,1]$. **Step 5**: Map the necessity degrees obtained in the previous step to the subinterval assigned to the criteria in the clause. **Step 6**: For each clause (ζ,μ) such that ζ is a rule of either $ws(W,Y) \leftarrow C_i(W,Y)$ or $\neg ws(W,Y) \leftarrow C_i(W,Y)$, we set μ to be the upper bound value of the subinterval assigned to C_i where $i \in \{1,2,3,4\}$. Before we explain how to obtain uncertain clauses related to $ws(W,Y)$ and $\neg ws(W,Y)$, it is noted here that we abuse the notation $\neg$. In fact, the $\neg$ in $\neg ws(W,Y)$ is called strong negation (unlike the negation as failure introduced in Section 2.2 in Chapter 2). The strong negation used in the head of logic rules is to represent the conflictive or contradictory information. Unless specified, the notion of the strong negation is employed exclusively for this concrete multi-criteria argumentation-based decision making. The interested readers are

referred to the work of [GS04] for detailed explanations and usages of strong negation, which is beyond the scope of this thesis. For the uncertain clause such as $(ws(W,Y) \leftarrow oef(W,Y), 0.24)$, it specifies a rule with a necessity degree 0.24 that if $oef(W,Y)$ holds, then $ws(W,Y)$ holds (where ws stands for "worse than"). Meanwhile, the similar clause $(\neg ws(W,Y) \leftarrow oef(Y,W), 0.24)$ can be understood intuitively as the "negate" form of $(ws(W,Y) \leftarrow oef(W,Y), 0.24)$. Finally, the set of clauses can obtained with certainty that if $sm(W,Y)$ holds (i.e. W and Y are same), then $\neg ws(W,Y)$ holds (i.e.W is not worse than Y), denoted as $(\neg ws(W,Y) \leftarrow sm(W,Y), 1)$. Similarly, we have $(\neg ws(W,Y) \leftarrow sm(Y,W), 1)$. Therefore, we can this set of certain clauses denoted as $\varpi = \{(\neg ws(W,Y) \leftarrow sm(W,Y), 1)\ (\neg ws(W,Y) \leftarrow sm(Y,W), 1)\}$.

$$\lambda = \left\{ \begin{array}{cc} (oef(P_2,P_1),0.19) & (ws(W,Y) \leftarrow oef(W,Y),0.24) \\ (oef(P_2,P_3),0.16) & (\neg ws(W,Y) \leftarrow oef(Y,W),0.24) \\ (oef(P_3,P_1),0.03) & (ws(W,Y) \leftarrow lef(W,Y),0.49) \\ (lef(P_2,P_1),0.48) & (\neg ws(W,Y) \leftarrow lef(Y,W),0.49) \\ (lef(P_2,P_3),0.27) & (ws(W,Y) \leftarrow osf(W,Y),0.74) \\ (lef(P_3,P_1),0.47) & (\neg ws(W,Y) \leftarrow osf(Y,W),0.74) \\ (osr(P_2,P_1),0.56) & (ws(W,Y) \leftarrow lsr(W,Y),0.99) \\ (osr(P_3,P_1),0.53) & (\neg ws(W,Y) \leftarrow lsr(Y,W),0.99) \\ (osr(P_2,P_3),0.53) & (lsr(P_1,P_2),0.84) \\ (lsr(P_1,P_3),0.83) & (lsr(P_3,P_2),0.77) \end{array} \right\}$$

$$\varpi = \{(\neg ws(W,Y) \leftarrow sm(W,Y), 1)\ (\neg ws(W,Y) \leftarrow sm(Y,W), 1)\}$$

Figure 5.1: A Set of Uncertain and Certain Clauses

The arguments of the form $\mathcal{A} = \langle u, h, \mu \rangle$ is presented in Figure 5.2 and is built from the uncertain clause program in which u is a set of uncertain clauses from λ, h the conclusion it supports (e.g. $ws(W,Y)$), and μ its necessity degree. For example, the argument $\mathcal{A}_1 = \langle \{(ws(P_2,P_1) \leftarrow oef(P_2,P_1), 0.24), (oef(P_2,P_1), 0.19)\}, ws(P_2,P_1), 0.19 \rangle$ is built from two uncertain clauses , namely $(ws(P_2,P_1) \leftarrow oef(P_2,P_1), 0.24)$ and $(oef(P_2,P_1), 0.19)$, both of which support conclusion $ws(P_2,P_1)$. The necessity degree of the argument $\mathcal{A}_1 = 0.19 = min\{0.24, 0.19\}$. Finally, we aggregate the arguments which support the same conclusion h into accrued structures. For example, $\mathcal{A}_1$, $\mathcal{A}_7$, and $\mathcal{A}_{13}$ support the same conclusion $ws(P_2,P_1)$ to build the accrued structure $[\Psi_1, ws(P_2,P_1), 0.82]$ in which $\Psi_1 = \mathcal{A}_1 \cup \mathcal{A}_7 \cup \mathcal{A}_{13}$. To calculate the necessity values for accrued structures, it will use the ACC function defined as $ACC(\mu_1, \cdots, \mu_n) = [1 - \prod(1-\mu_i)] + Kmax(\mu_1, \cdots, \mu_n)\prod(1-\mu_i)$ with $K = 0.1$. For example, for , the accrued structure $[\Psi_1, ws(P_2,P_1), 0.82]$, we have the necessity values $0.82 = [1-(1-0.19)\times(1-0.48)\times(1-0.56)] + 0.1\times max\{0.19, 0.48, 0.56\}\times(1-0.19)\times(1-0.48)\times(1-0.56)$. As it can be observed, 12 aggregated arguments shown in Figure 5.2 can be built to support the reasons by which a candidate should be deemed worse than another one. Those aggregated

arguments warranted (shown in bold) because of their greater necessity values than their negated counterparts will be used to compute the final chosen candidate(s) with decision rules $\mathcal{R}$. In this particular case, only decision rule *DR1* can be applied. For candidate P_1, precondition of *DR1* can be warranted and there is no warranted accrued structure supporting a conclusion of the kind $ws(Z,P_1)$ to warrant the restriction of rule *DR1*, hence P_1 selected for deletion.

In summary, through this planetary vehicle, we have successfully demonstrated in great details how the plan library contraction operator can be instantiated to be a concrete multi-criteria argumentation-based decision-making process. Furthermore, we can see that choosing plan P_1 is quite evident for the so-called human common sense reasoning since it has the worst latest success rate (i.e. 0.5) which is most important preference criterion, despite having a best overall success rate. The worst latest success rate may imply that P_1 is no longer suitable for the current Martian surface-navigation task, thus ready to be dropped by the Rover. In the next section, we will formally prove that such a multi-criteria argumentation-based decision-making process indeed satisfies the postulates of a plan library contraction operator.

5.4.4 Theorem

Theorem 3. $\nabla^{abm} = sol(\langle X, \mathcal{K}, \mathcal{R}\rangle)$ *is a contraction operator* ∇ *satisfying* ***CO1-CO5***

Proof. Postulates **CO1** (i.e. $\nabla^{abm}(\Pi)$ is a plan library) and **CO2** (i.e. $\nabla^{abm}(\Pi) \subseteq \Pi$) hold as plans are simply selected and removed from the original plan library Π. Postulate **CO3** (i.e. relativity of contraction) holds for ∇^{abm} due to two computation steps of uncertain clauses λ. The normalisation of criteria values (in **Step 1**) and the absolute value of the remainder of the normalised criteria values (in **Step 2**) imply that a plan is deemed worse than the others is in a relative sense in a given set of plans. Postulate **CO4** says that the contraction should not witness the decrease of both activeness and the success rate of the plan library. It holds for ∇^{abm} because the selected plans to be removed are those which are deemed worse either in success rate criterion or both success rate and execution frequency criteria than other plans. Therefore, at least the success rate of all plans will be increased after contraction. Hence **CO4** holds. Postulate **CO5** (i.e. the protection of fragility of the plan library) holds for ∇^{abm} because we exclude plans which do not have any replaceable plans beforehand show in step 2 in Algorithm 1. Therefore, there are still plans which can replace deleted plans after the contraction. ■

5.5 Conclusion

In this chapter, we described measures that characterise the performance and structure of plans, and provided rationales to guide the process of new plan adoption (i.e. plan library expansion) and unsuitable plan deletion (i.e. plan library contraction) to obtain an adaptive agent for a fast-changing environment. Specifically, we introduce the notation of execution frequency to capture how often a plan is applied, and such execution frequency can further be used to define

$$\mathcal{A}_1 = \langle\{(ws(P_2,P_1) \leftarrow oef(P_2,P_1),0.24),(oef(P_2,P_1),0.19)\},ws(P_2,P_1),0.19\rangle$$
$$\mathcal{A}_2 = \langle\{(\neg ws(P_1,P_2) \leftarrow oef(P_2,P_1),0.24),(oef(P_2,P_1),0.19)\},\neg ws(P_1,P_2),0.19\rangle$$
$$\mathcal{A}_3 = \langle\{(ws(P_2,P_3) \leftarrow oef(P_2,P_3),0.24),(oef(P_2,P_3),0.19)\},ws(P_2,P_3),0.19\rangle$$
$$\mathcal{A}_4 = \langle\{(\neg ws(P_3,P_2) \leftarrow oef(P_2,P_3),0.24),(oef(P_2,P_3),0.19)\},\neg ws(P_3,P_2),0.19\rangle$$
$$\mathcal{A}_5 = \langle\{(ws(P_3,P_1) \leftarrow oef(P_3,P_1),0.24),(oef(P_3,P_1),0.03)\},ws(P_3,P_1),0.03\rangle$$
$$\mathcal{A}_6 = \langle\{(\neg ws(P_1,P_3) \leftarrow oef(P_3,P_1),0.24),(oef(P_3,P_1),0.03)\},\neg ws(P_1,P_3),0.03\rangle$$
$$\mathcal{A}_7 = \langle\{(ws(P_2,P_1) \leftarrow lef(P_2,P_1),0.49),(lef(P_2,P_1),0.48)\},ws(P_2,P_1),0.48\rangle$$
$$\mathcal{A}_8 = \langle\{(\neg ws(P_1,P_2) \leftarrow lef(P_2,P_1),0.49),(lef(P_2,P_1),0.48)\},\neg ws(P_1,P_2),0.48\rangle$$
$$\mathcal{A}_9 = \langle\{(ws(P_2,P_3) \leftarrow lef(P_2,P_3),0.49),(lef(P_2,P_3),0.27)\},ws(P_2,P_3),0.27\rangle$$
$$\mathcal{A}_{10} = \langle\{(\neg ws(P_3,P_2) \leftarrow lef(P_2,P_3),0.49),(lef(P_2,P_3),0.27)\},\neg ws(P_3,P_2),0.27\rangle$$
$$\mathcal{A}_{11} = \langle\{(ws(P_3,P_1) \leftarrow lef(P_3,P_1),0.49),(lef(P_3,P_1),0.47)\},ws(P_3,P_1),0.47\rangle$$
$$\mathcal{A}_{12} = \langle\{(\neg ws(P_1,P_3) \leftarrow lef(P_3,P_1),0.49),(lef(P_3,P_1),0.47)\},\neg ws(P_1,P_3),0.47\rangle$$
$$\mathcal{A}_{13} = \langle\{(ws(P_2,P_1) \leftarrow osr(P_2,P_1),0.74),(osr(P_2,P_1),0.56)\},ws(P_2,P_1),0.56\rangle$$
$$\mathcal{A}_{14} = \langle\{(\neg ws(P_1,P_2) \leftarrow osr(P_2,P_1),0.74),(osr(P_2,P_1),0.56)\},\neg ws(P_1,P_2),0.56\rangle$$
$$\mathcal{A}_{15} = \langle\{(ws(P_3,P_1) \leftarrow osr(P_3,P_1),0.74),(osr(P_3,P_1),0.53)\},ws(P_3,P_1),0.53\rangle$$
$$\mathcal{A}_{16} = \langle\{(\neg ws(P_1,P_3) \leftarrow osr(P_3,P_1),0.74),(osr(P_3,P_1),0.53)\},\neg ws(P_1,P_3),0.53\rangle$$
$$\mathcal{A}_{17} = \langle\{(ws(P_2,P_3) \leftarrow osr(P_2,P_3),0.74),(osr(P_2,P_3),0.53)\},ws(P_2,P_3),0.53\rangle$$
$$\mathcal{A}_{18} = \langle\{(\neg ws(P_3,P_2) \leftarrow osr(P_2,P_3),0.74),(osr(P_2,P_3),0.53)\},\neg ws(P_3,P_2),0.53\rangle$$
$$\mathcal{A}_{19} = \langle\{(ws(P_1,P_2) \leftarrow lsr(P_1,P_2),0.99),(lsr(P_1,P_2),0.84)\},ws(P_1,P_2),0.84\rangle$$
$$\mathcal{A}_{20} = \langle\{(\neg ws(P_2,P_1) \leftarrow lsr(P_1,P_2),0.99),(lsr(P_1,P_2),0.84)\},\neg ws(P_2,P_1),0.84\rangle$$
$$\mathcal{A}_{21} = \langle\{(ws(P_1,P_3) \leftarrow lsr(P_1,P_3),0.99),(lsr(P_1,P_3),0.83)\},ws(P_1,P_3),0.83\rangle$$
$$\mathcal{A}_{22} = \langle\{(\neg ws(P_3,P_1) \leftarrow lsr(P_1,P_3),0.99),(lsr(P_1,P_3),0.83)\},\neg ws(P_3,P_1),0.83\rangle$$
$$\mathcal{A}_{23} = \langle\{(ws(P_3,P_2) \leftarrow lsr(P_3,P_2),0.99),(lsr(P_3,P_2),0.77)\},ws(P_3,P_2),0.77\rangle$$
$$\mathcal{A}_{24} = \langle\{(\neg ws(P_2,P_3) \leftarrow lsr(P_3,P_2),0.99),(lsr(P_3,P_2),0.77)\},\neg ws(P_2,P_3),0.77\rangle$$
$$[\Psi_1, ws(P_2,P_1), 0.82],\ \mathbf{[\Psi'_1, \neg ws(P_2,P_1), 0.85]},\ \Psi_1 = \mathcal{A}_1 \cup \mathcal{A}_7 \cup \mathcal{A}_{13},\ \Psi'_1 = \mathcal{A}_{20}$$
$$[\Psi_2, \neg ws(P_1,P_2), 0.82],\ \mathbf{[\Psi'_2, ws(P_1,P_2), 0.85]},\ \Psi_2 = \mathcal{A}_2 \cup \mathcal{A}_8 \cup \mathcal{A}_{14},\ \Psi'_2 = \mathcal{A}_{19}$$
$$[\Psi_3, ws(P_2,P_3), 0.73],\ \mathbf{[\Psi'_3, \neg ws(P_2,P_3), 0.79]},\ \Psi_3 = \mathcal{A}_3 \cup \mathcal{A}_9 \cup \mathcal{A}_{17},\ \Psi'_3 = \mathcal{A}_{24}$$
$$[\Psi_4, \neg ws(P_3,P_2), 0.73],\ \mathbf{[\Psi'_4, ws(P_3,P_2), 0.79]},\ \Psi_4 = \mathcal{A}_4 \cup \mathcal{A}_{10} \cup \mathcal{A}_{18},\ \Psi'_4 = \mathcal{A}_{23}$$
$$[\Psi_5, ws(P_3,P_1), 0.77],\ \mathbf{[\Psi'_5, \neg ws(P_3,P_1), 0.84]},\ \Psi_5 = \mathcal{A}_5 \cup \mathcal{A}_{11} \cup \mathcal{A}_{15},\ \Psi'_5 = \mathcal{A}_{22}$$
$$[\Psi_6, \neg ws(P_1,P_3), 0,77],\ \mathbf{[\Psi'_6, ws(P_1,P_3), 0.84]},\ \Psi_6 = \mathcal{A}_6 \cup \mathcal{A}_{12} \cup \mathcal{A}_{16},\ \Psi'_6 = \mathcal{A}_{21}$$

Figure 5.2: Arguments and accrued structures

the concept of the plan success rate to capture how well a plan has been performing. Besides these performance-based measures, we also introduce the measures that reflect the unique structure nature of the pre-defined plan library of BDI agents, namely functionality, relevancy, and replaceability. These measures are all based on intuitive and general concepts in BDI, e.g. how plans relate to each other and how the agent performs when dealing with specific events or goals. As such, the measure strategies and rationales we provide in this work are generic and do not require additional information from the BDI agent developer or the domain, beyond what is required in the typical BDI agent development. The merit of our framework is that we are one of the first works which formally define an evolving capacity of the BDI agents through changes to the set of existing BDI plans, thus increasing the adaptivity of BDI agents. Regarding the plan library expansion, the plan library expansion operator we proposed would only increase the

functionality and replaceability of the resulting plan library. Therefore, the agent can potentially respond to more events with more relevant plans, and has a better chance of replacing some plans. It confirms the intuition of planning-based extensions to BDI agents, and encourages the usage of new plans from advanced planning tools. With our plan library contraction operator, the agent not only can drop the worst or oldest plans to increase its performance, but also protect the fragility of the plan library by ensuring there are always other plans to replace deleted plans. Finally, our instantiated plan library contraction operator in a planetary vehicle example (which excludes the expansion operator) demonstrates the feasibility and applicability of plan library contraction operator in our approach.

CHAPTER 6

Efficient Intention Progression via Planning

The bulk of this chapter has been published online in [XMBL19].

6.1 Introduction

In the previous two chapters, we discussed the extensions of Belief-Desire-Intention (BDI) by embedding planning to recover the execution failure to increase robustness, and adding or deleting plans to adapt to the changing environment. Both of these extensions have tried to overcome the drawbacks associated with the pre-defined plan libraries in the mainstream BDI implementations. In other words, their primary focus is the plans in BDI agents. In this chapter, we look at another issue related to intentions – a crucial part of BDI agents – which are one of the least studied areas in BDI theory [HLPX17]. In particular, we will address the problem of intention interleaving, where we are interested in identifying and exploiting overlapping programs (e.g. common actions) between different intentions.

A desirable property of any agent-based system is that the system should be reactive in a dynamic and complex domain. Such reactive nature requires the agent to be able to respond to new events even while already dealing with other events. Indeed, one of the crucial features of BDI agents is their ability to pursue multiple intentions in parallel, i.e. multitasking. To this end, intentions are often executed in an interleaved manner. For example, the agent can execute one step of an intention at each cycle in a round-robin fashion [BHW07]. However, the interactions between interleaved steps in different intentions may result in conflicts, i.e. the execution of a step makes it impossible to execute a step in another concurrent intention. Therefore, it is critical for the agent to avoid negative interference between intentions, i.e. conflict resolution [TPW03a, YL16], when an agent is pursuing multiple intentions in parallel. However, to avoid execution inefficiency, the agent also should capitalise on positive interactions between intentions,

in particular, the positive interactions when intentions overlap with each other. In this thesis, we regard the intention progress which exploits the positive interactions as a form of efficient intention progression regarding the cost of execution in BDI agents. Indeed, opportunities for positive interactions between intentions can enable the agent to reduce the effort (e.g. resources) it exerts to accomplish its intentions. For example, the agent with the overlapping intentions can merge these intentions to reduce the actual overall execution cost. Such the mechanism of intention merging effectively allows the agent to skip some of its plan steps in its plan, i.e. kill two birds with one stone, while still managing to achieve all intentions. Finally, to avoid any confusion, we stress that the notion of "efficient intention progress" in this thesis shall be interpreted as an algorithm to reason how to accomplish intentions with as *less* actual resources the agent needs as possible rather than a fast real-time algorithm which only reasons how to progress intentions. In other words, our "efficiency" measures the physical cost when the agent interacts with the external environment via action execution rather than the traditional sense of the high speed of internal computational reasoning.

To illustrate the problem, consider a BDI implementation for a Mars Rover agent. The agent has a goal to transmit soil experiment results and a goal to transmit image collection results. The agent could perform the goals sequentially by *establishing the connection* with the Earth, sending soil experiment results, *breaking the connection*, then *establishing the connection* with the Earth, sending image collection results, and *breaking the connection*. Alternatively, it could *establish the connection* with the Earth, send both the soil experiment and image collection results, and *break the connection*. Clearly, the second approach specifies a kind of agent which is able to discover and exploit the commonality of different intentions. Unlike the traditional agent, such a type of agent is capable of manifesting a more sensible and intelligent behaviour whenever appropriate. For example, a domestic Artificial Intelligence (AI) assistant can identify the online order commands from the users and merge several separate orders into one order, thus reducing the package receiving efforts for users. While, unlike conflict resolution, exploiting commonality of intentions is not necessary for the agent to perform its tasks correctly, it can be of vital importance in a resource-critical domain such as in the autonomous manufacturing sector [SWH06] (in Section 6.4 we will present a manufacturing scenario of using machining operations to make holes in a metal block).

Within the BDI community, there are few papers which address these issues. One motive for this is that there has been a focus on a simple intention selection mechanism that favours highly efficient reasoning cycle above all else as we have discussed in Section 3.4. Still, recently, a number of approaches on dealing with positive interactions between multiple intentions in parallel have been released. In works of [TPW03b, TP11], for instance, the authors propose a way to detect and exploit positive intention interactions by reasoning about definite and potential preconditions and post-effects of plans and goals. However, this approach is limited to intention merging at the plan level to avoid duplicate plan executions, thus ignoring the merging of individual steps (e.g. actions)

within plans. As a result, the approach needs to adopt a conservative strategy where the merging is allowed if and only if the definite and potential effects of one plan is completely subsumed by the others to preserve correct intention execution. However, some conflicts between intentions may not be resolved by the appropriate ordering of plans and can only be resolved by appropriate interleaving of steps within plans. Furthermore, post-effects are not commonly defined for plans in BDI implementations and thus must be inferred, which prevents this approach from being used in, e.g. domains pervaded by uncertainty.

Apart from the work of [TPW03b, TP11], we are not aware of any other existing work on intention interleaving with the focus on discovering and exploiting identical sub-intentions in BDI agents. There are also some other works which exploit the positive intention interaction for intention resolution in BDI agents. For example, the work of [YLT16b] studied the robust execution of BDI agent programs by exploiting synergies between intentions. Instead of backtracking to recover from an execution failure, they proposed an approach to appropriate scheduling the remaining progressable intentions to execute an already intended action which re-establishes a missing precondition. In contrast, we address the problem of intention resolution, where we guarantee the achievability of intentions by avoiding all negative interaction between intention in this work. Another noticeable work [WR11] combines work on both intention and planning. However, their purpose is to split the original actions into several stages of intention (i.e. refinement of action) to solve unary planning problems. In fact, a large number of works integrate automated planning techniques into BDI agents to generate plans at runtime, as surveyed by Meneguzzi and De Silva [MS15]. However, our work is one of the few which formally integrates planning techniques into BDI agents to managing intention interleaving.

This leaves the agent with a brittle mechanism to detect potential overlapping intentions and attempting to schedule its actions to take advantage of them. Instead, in this chapter, we show that within a BDI context, as a high-level agent modelling language, many of these intention issues can be resolved through planning in an automated fashion. We accomplish this by showing how intentions (particularly the complete intention execution traces, each of which leads to the successful execution of an intention) can be modelled as the search space of a PDDL problem description [MGH$^+$98]. Subsequently, planning is employed to identify a *conflict-free* (i.e. correct execution) and *maximal-merged* (i.e. minimal effort) execution trace. The approach we introduce is agnostic to the actual planner being used, thus implying our approach can be used with modern highly efficient online planners (e.g. [KE12]) to execute plans until it is necessary to replan. Furthermore, since we rely on third-party implementations, we immediately benefit from any improvement made to these planners.

The contribution in this chapter is another extension to the BDI framework where planning is used to exploit overlapping intentions while resolving conflicts during the interleaved execution of intentions. To achieve so, we first formalise the intention of a BDI agent as an AND/OR graph to capture the unique hierarchical structure of a plan library in Section 6.2.1. In the

context of AND/OR graphs, we define the concept of execution traces of an intention to identify every unique way in which a given intention can be achieved. When the agent is interleaving different intentions, it is essentially interleaving elements in the execution traces of different intentions. Therefore, the potential execution trace of a set of intentions can be constructed by interleaving the execution traces of each intention. However, randomly interleaving intentions may cause negative interactions to arise. To model the successful interleaving which achieves all intentions, each element in the potential execution trace of a set of intentions cannot be blocked right before its execution. Such an execution trace is called the conflict-free execution trace. Regarding the positive interactions, we define the concept of mergeable execution trace to capture the commonality of different intentions. Furthermore, we provide a method of computing all potential overlapping programs among a set of intentions to identify such potential positive interactions in Section 6.2.2. As a consequence, we are essentially transforming the problem of intention interleaving in BDI agents to be a task of searching for a conflict-free and maximally merged execution trace if there exists one. Therefore, it is natural for us to employ the planning to search such a conflict-free and maximal-merged execution trace for the agent. To this end, we provide a formal framework of applying planning to solve the problem of intention interleaving in Section 6.2.3. In addition, we also provide the detailed instruction of implementations along with necessary knowledge transformation between BDI agents and First-principles Planning (FPP) in Section 6.3. Finally, to demonstrate the feasibility and effectiveness of our approach, we present evaluation in manufacturing testbeds of increasing sizes in Section 6.4.

6.2 Intention Interleaving Planning Framework

In this section, we formally define the goal-plan trees to model the intentions of a BDI agent, and we use these goal-plan trees in Section 6.2.1 to define the *conflict-free* and *maximal-merged* execution traces of intentions. In Section 6.2.2 and Section 6.2.3, we outline a theoretical approach where planning is used to manage the intention interleaving in a way that maximises the intention merging while guaranteeing the achievability of all intentions.

6.2.1 Intention Formalisation

In BDI agent systems, the so-called goal-plan trees have been the canonical representation of intentions [TP11]. The root of a goal-plan tree is a top-level goal, and its children are plans that can be used to achieve such a top-level goal. Plans may also contain sub-goals, giving rise to a tree structure representing all possible ways of achieving the goal. In this thesis, we also use it to represent the underlying hierarchy in the plan library. Before we proceed with our framework, it is stressed that the BDI language which we use in this thesis allows the various form of triggering events, e.g. achievement goal and belief addition as discussed in Section 2.3.1. However, for the reason of legibility, we unify all potential form of triggering events and symbolically represent

them as goals, denoted $\mathcal{G}$ as a set of (sub)goals We now give the definition of a goal-plan tree [TP11] which we formalise and simplify in this thesis. Let us recall that Π is a set of plans and Λ is a set of actions. We can have the following definition of goal-plan tree for an intention:

Definition 12. A goal-plan tree for an intention in a BDI agent to achieve a top-level goal $G \in \mathcal{G}$ is an AND/OR graph $T = (N_\vee \cup N_\wedge, L_\vee \cup L_\wedge, E_\vee \cup E_\wedge, \bar{n})$ where:

1. $\bar{n} = G$ (i.e. the top-level goal);
2. $N_\vee \subseteq \mathcal{G} \cup \Lambda$ (i.e. sub-goals or individual actions);
3. $N_\wedge \subseteq \Pi$ (i.e. plans to deal with goals);
4. $L_\vee = \mathcal{L}$ (i.e. the logical language);
5. $L_\wedge \subseteq \mathbb{N}^+$ where $\mathbb{N}^+$ is the set of positive integers;
6. $(G, \varphi, P) \in E_\vee$ if $P \in \Pi$ such that $head(P) = G$ and $context(P) = \varphi$;
7. $(G', \varphi', P') \in E_\vee$ if there exists a path from G to G' with $G' \in \mathcal{G}$ such that $P' \in \Pi$ with $head(P') = G'$ and $context(P') = \varphi'$;
8. $(P, j, h) \in E_\wedge$ if there exists a path from G to P with $P \in \Pi$ such that $O(h, P) = j$;

where $head(P)$, $context(P)$, and $body(P)$ refers to the head, context condition, and plan-body of a plan P, respectively, and $O(h, P)$ denotes the plan-body order of a given component h in the body of the plan P.

The root node of a goal-plan tree[1] is the top-level goal specified by the criterion (1). Criterion (2) and (3) assign the BDI components to the nodes. In detail, the set of (sub)goals and individual actions are assigned to be OR-nodes, whereas the set of plans the AND-nodes. Criterion (6) and (7) link a goal with its relevant plans using OR-edges which are labelled with the context condition of the corresponding plan according to the criterion (4). Meanwhile, the criterion (8) links a plan with its plan-body using AND-edges which are labelled with a natural number which indicates the execution order shown in the criterion (5). For convenience, we refer to the component of goal-plan tree T, e.g. $T[\bar{n}]$ for the top-level goal of T, and $T[N]$ for both OR-nodes and AND-nodes of T. To explain the concepts in Definition 12 above, we now present an example and graphical illustration as follows:

Example 4. Let G_1 and G_2 be the goals of our Mars Rover to transmitting the soil experiment results and transmitting the image collection results, respectively. We have plan P_1 and P_2 to achieve G_1 and G_2, respectively, and they are given as follows:

[1] Note that despite still being called a goal-plan tree here, it does not satisfy the definition of a tree as there can exist two nodes connected by more than one path due to multiple relevant plans for a (sub)goal.

$$P_1 = G_1 : \varphi_1 \leftarrow a_1;a_2;a_4; \quad P_2 = G_2 : \varphi_2 \leftarrow a_1;a_3;a_4;$$

where φ_1 and φ_2 are the context condition of P_1 and P_2, respectively. The action a_1 (resp. a_4) stands for establishing (resp. breaking) the connection. Meanwhile, the action a_2 (resp.a_3) denotes transmitting the soil experiment (resp. image collection) results. The corresponding goal-plan trees are presented in Figure 6.1, which are constructed according to Definition 12 above.

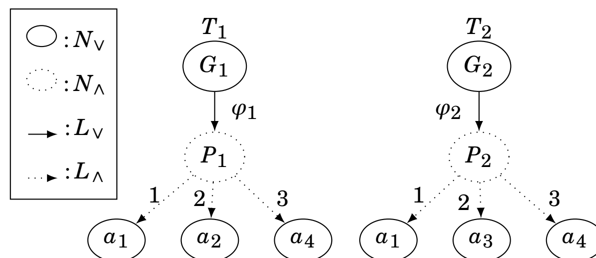

Figure 6.1: AND/OR Graphs for Goal-plan Trees.

We have modelled the intention in a BDI agent via the structure of the goal-plan tree. Recall that BDI agents typically pursue multiple goals in parallel through the interleaving of steps in different intentions. We now look at the problem of intention interleaving in the context of goal-plan trees. To begin with, we introduce the definition of the *execution trace* of a single intention, which identifies every unique way in which a given intention can be achieved. Therefore, we can have the following definition of an execution trace of a goal-plan tree for an intention:

Definition 13. Let T be a goal-plan tree for an intention. An execution trace of T is defined to be $\tau(T) = \tau(T(\bar{n}))$ such that the following condition hold:

1. $\tau(G) = G; \tau(P)$ s.t. $head(P) = G$;

2. $\tau(P) = P; \tau(h_1);\ldots;\tau(h_n)$ such that $body(P) = h_1;...;h_n$;

3. $\tau(a) = a$;

where $T(\bar{n})$ denotes the top-level goal of T, $a, P, G \in T(N)$ (i.e. the set of AND/OR nodes of T). We also denote the set of all execution traces of a goal G by $\omega(G)$, i.e. $\tau(G) \in \omega(G)$.

Definition 13 says that an execution trace of an intention is an execution trace of its top-level goal. The condition (1) says that an execution trace of a goal is the sequence beginning with the adoption of such goal followed by the execution trace of *one* of its relevant plans. Normally, a goal in a BDI agent has more than one relevant plans, thus amounting to more than one execution trace for a goal. The condition (2) says the execution trace of a plan consists of the plan identifier followed by the trace of the individual elements of its body. The plan identifier in the execution trace stands for the selection of the plan. We note that the plan selection and the execution of the

first body element in a plan are distinct steps. In addition, the execution trace of the plan-body of a plan preserves the order of its elements shown in (2). Finally, we can see that the execution trace of action is trivially the action itself given in the condition (3). The following is an example of execution traces of a goal-plan tree for an intention:

Example 5. Consider the goal-plan tree T_3 given in Figure 6.2. We can see that it has two execution traces, namely $\tau_1(T_3)$ and $\tau_3(T_3)$ as follows:

$$\tau_1(T_3) = G_3;P_3;a_4;a_5; \quad \tau_2(T_3) = G_3;P_4;b_4;b_5;b_6;$$

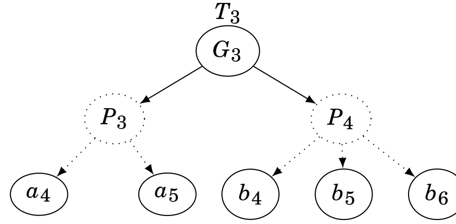

Figure 6.2: A Goal-plan Tree with Two Relevant Plans.

So far we have defined the execution trace for a single intention. We are ready now to define an execution trace of a set of intentions $\{T_1, \cdots, T_m\}$, as a BDI agent is typically pursuing multiple goals in parallel. Let us recall $T_j(\bar{n})$ is the top-level goal of T_j and $\omega(T_j(\bar{n}))$ is the set of all execution traces of $T_j(\bar{n})$. We have the following definition:

Definition 14. An execution trace of a set of intentions $\{T_1, \ldots, T_m\}$ is any sequence σ obtained by interleaving a finite number of execution traces from the set of $\bigcup_{j=1}^{m} \omega(T_j(\bar{n}))$ such that $|\{i \mid \sigma[i] = T_j(\bar{n})\}| = 1$ where $\sigma[i]$ denotes the ith element of σ and $1 \leq j \leq m$.

Definition 14 says that the construction of an execution trace of a set of intentions is to interleave elements in the execution traces of different intentions. Intuitively, the requirement on the cardinality of the top-level goal of each intention says that each intention only needs to be achieved once. Therefore, there is one and only one execution trace of each intention being interleaved with the execution traces of other intentions. The following is an example of an execution trace for a set of intentions:

Example 6. In Figure 6.1, we can have that the intention T_1 has only one trace, namely $\tau(T_1) = G_1;P_1;a_1;a_2,a_4$. Meanwhile, there are two execution trace for intention T_3, namely $\tau_1(T_3)$ and $\tau_2(T_3)$ given in Example 5. Therefore, one possible execution trace of intentions $\{T_1, T_3\}$ can be $\sigma = \mathbf{G_1};\mathbf{P_1};G_3;P_3;\mathbf{a_1};a_4;\mathbf{a_2};\mathbf{a_4};a_5$ by interleaving $\tau(T_1)$ and $\tau_1(T_3)$, where the subsequence in bold is $\tau(T_1)$ and non-bold is $\tau_1(T_3)$, i.e. one of execution traces of T_3 given in Example 5.

However, randomly interleaving intentions can cause negative interactions to arise. For example, a previously achieved effect may be undone before an action that relies on it begins executing, thus preventing that action from being able to execute. Therefore, to model the successful interleaving which achieves all intentions (i.e. intention resolution), we now define the concept of a *conflict-free* execution trace of a set of intentions as follows:

Definition 15. Let $\mathcal{B}_j$ be the belief base before the execution of the jth element of an execution trace (i.e. $\sigma[j]$). An execution trace σ is conflict-free if and only if the followings hold:

(i) if $\sigma[j] = P \in \Pi$, then $\mathcal{B}_j \models context(P)$ (i.e. the context of plan P must be met before selection);

(ii) if $\sigma[j] = a \in \Lambda$, then $\mathcal{B}_j \models \psi(a)$ (i.e. the precondition of action 'a' must be met before execution).

where $j \in \{1, \ldots, |\sigma|\}$ and $|\sigma|$ is the length of σ.

Definition 15 says that a conflict-free execution trace is an execution trace which can be fully executed to completion without failure once it starts executing, thus avoiding all possible negative interactions between intentions. In order for an execution trace to be completed without failure, every element of such an execution trace cannot be blocked. For instance, the context condition of a plan in the execution trace has to hold when the agent selects this plan. Similarly, the agent cannot execute an action in the execution trace if its precondition does not hold before execution.

Besides the negative interactions between intentions, there may also exist potential positive interactions between them. For example, there may be a common sub-intention of two intentions that need only be executed once (i.e. merging such two identical sub-intentions into one) in order to progress both these two intentions. Therefore, to capture the commonality of intentions in the execution trace, we start with providing the following definition of the *mergeable* execution trace:

Definition 16. An execution trace σ of $\{T_1, \ldots, T_m\}$ is a mergeable execution trace if and only if the followings hold:

(i) $\exists j \in \{1, \ldots, |\sigma|\}$ such that $\sigma[j] = \ldots = \sigma[j+k]$ where $|\sigma|$ is the length of σ and $2 \leq k \leq |\sigma| - j$;

(ii) $\forall l \in \{1, \ldots, m\}, \nexists s, t \in \{j, \ldots, j+k\}$ where $s \neq t$ such that $\sigma[s] \subseteq \tau(T_l) \subseteq \sigma$ and $\sigma[t] \subseteq \tau(T_l) \subseteq \sigma$.

(iii) σ^m is a conflict-free execution trace where σ^m is the merged execution trace of σ by reducing each subsequence consisting of consecutive identical elements characterised by (i) and (ii) in σ to only one element.

In Definition 16, criterion (i) and (ii) first capture the synchronisation stage, which requires different intentions ready to be executed the same actions at the same time. In the criterion (iii), it formalises the intention merging stage such that the subsequent merged execution trace σ^m is still a conflict-free execution trace, thus ensuring a correct execution.

Example 7. In Figure 6.1, we can have that one of the execution traces of T_1 and T_2 is $\sigma_1 = G_1;P_1;G_2;P_2;\mathbf{a_1};\mathbf{a_1};a_2;a_3;\mathbf{a_4};\mathbf{a_4}$. We can conclude that σ_1 is mergeable according to Definition 16 and its merged execution trace $\sigma_1^m = G_1;P_1;G_2;P_2;\mathbf{a_1};a_2;a_3;\mathbf{a_4}$ is indeed a conflict-free execution trace (see in the example discussed in Section 6.1).

Finally, we can define the following concept of *maximal-merged* execution trace to seeking the maximal amount of intention merging if possible.

Definition 17. The merged execution trace σ^m of a mergeable execution trace σ of $\{T_1,\ldots,T_m\}$ is maximal-merged if there is no another mergeable execution trace σ' of $\{T_1,\ldots,T_m\}$ such that $|\sigma'^m| < |\sigma^m|$ where $|\sigma|$ stands for the length of σ.

At this stage, we have fully defined the execution trace of a given set of intentions. With each set of intentions, we can now associate a (potentially large) set of execution traces, we are interested in finding one *maximal-merged* trace for a set of intentions if one exists. To this end, in the next section, we leverage the power of FPP to help us find such a *maximal-merged* trace.

6.2.2 Intention Interleaving Planning Preparation

In this section, we show that the off-the-shelf FPP planners can be applied to identify a maximal-merged trace if one exists. Before we formally present our FPP approach to solving the problem of intention interleaving, we start with some technical preparation.

Indexing nodes: We introduce some additional notations, i.e. indexes, to the nodes of goal-plan trees. If a node n is a top-level goal of intention T, it is already uniquely identified by the notation $T(\bar{n})$. For nodes of action and sub-goals, i.e. $n \in \Lambda \cup \mathcal{G} \setminus \{T(\bar{n})\}$ of T, we use $n^{P,j,T}$ to denote the jth member of *body*(P) in T. This ensures that, e.g. the same action in distinct plans is seen as different. Similarly, we use n^T to denote a plan node $n \in \Pi$ in an intention T. For ease of reference, we denote $J(idx)$ to retrieve the actual node of the index idx. From now on, we assume that whenever we talk about the nodes, we refer to the indexes of these nodes.

Terminal and initial node set: We introduce the *terminal node set* for a goal node $G \in \mathcal{G}$. This set encodes the completion condition of the goal node, which is the last element of an execution trace of a goal. To be precise, the *terminal node set* of a goal node G is defined to be $\nu(G) = \{\tau(G)^\infty \mid \tau(G) \in \omega(G)\}$ where $\tau(n)^\infty$ stands for the last element of execution trace $\tau(n)$. Therefore, we can have $z_g = \{tn_1,\ldots,tn_m\}$ to be a *terminal node set* of a set of intentions $I = \{T_1,\ldots,T_m\}$, denoted $z_g \rhd_{tn} I$, where $tn_j \in \nu(T_j[\bar{n}])$ and $j \in \{1,\ldots,m\}$. When every element in a terminal node set is reached for a set of intentions, we know that this set of intentions is completed successfully. Similarly, the top-level goal of each intention in $I = \{T_1,\ldots,T_m\}$, denoted as $z_0 = \{T_1(\bar{n}),\ldots,T_m(\bar{n})\}$, is called an *initial node set* of I. This set announces the starting point of each intention. To illustrate the concepts of indexes and terminal and initial node set, we now provide the following example with visualisation.

Example 8. In Figure 6.1, we can have the indexes and terminal and initial nodes of execution traces of intention T_1 and T_2 as follows:

$$\tau(T_1): \begin{pmatrix} node & G_1 & P_1 & a_1 & a_2 & a_4 \\ index & T_1(\bar{n}) & P_1^{T_1} & a_1^{P_1,1,T_1} & a_2^{P_1,2,T_1} & a_4^{P_1,3,T_1} \end{pmatrix}$$

initial node (pointing to $T_1(\bar{n})$ and G_2) *terminal node* (pointing to $a_4^{P_1,3,T_1}$ and a_4)

$$\tau(T_2): \begin{pmatrix} node & G_2 & P_2 & a_1 & a_3 & a_4 \\ index & T_2(\bar{n}) & P_2^{T_2} & a_1^{P_2,1,T_2} & a_3^{P_2,2,T_2} & a_4^{P_2,3,T_2} \end{pmatrix}$$

Progression links: We introduce the concept of *progression links* to encode the progression order information of the execution traces. To begin with, we first define the primitive progression links to visualise the progression order of execution trace elements in the context of indexes.

Definition 18. Let σ be an execution trace. For every two adjacent elements with indexes n, n' in σ (i.e. $n;n' \subseteq \sigma$), we say that an item in the form of $(n \to n')$ is a primitive progression link in σ, denoted as $(n \to n') \in \sigma$.

Example 9. (Example 8 continued). We can have the progression links of execution trace $\tau_1(T_1)$ and $\tau_2(T_2)$ shown as follows:
$\tau(T_1)$: $(T_1(\bar{n}) \to P_1^{T_1}), (P_1^{T_1} \to a_1^{P_1,1,T_1}), (a_1^{P_1,1,T_1} \to a_2^{P_1,2,T_1}), (a_2^{P_1,2,T_1} \to a_4^{P_1,3,T_1})$;
$\tau(T_2)$: $(T_2(\bar{n}) \to P_2^{T_2}), (P_2^{T_2} \to a_1^{P_2,1,T_2}), (a_1^{P_2,1,T_2} \to a_3^{P_2,2,T_2}), (a_3^{P_2,2,T_2} \to a_4^{P_2,3,T_2})$;

Computing overlaps: We have mentioned the potential common sub-intentions among many different intentions. We now discuss how to compute all potential overlapping programs among a set of intentions.

Definition 19. The overlap set of a set of intentions $\{T_1, \ldots, T_m\}$ $(m \geq 2)$ is a set of tuples of the form $\langle (idx_b^1 \to idx_e^1), \ldots, (idx_b^k \to idx_e^k) \rangle$ $(2 \leq k \leq m)$ such that the followings hold:

(1) $J(idx_e^1) = \ldots = J(idx_e^k)$ where $J(idx_e^i)$ stands for the actual node of the ending index idx_e^i;

(2) $\forall l \in \{1, \ldots, m\}, \nexists s, t \in \{1 \ldots, k\}$ and $s \neq t$ s.t. $(idx_b^s \to idx_e^s) \in \tau(T_l)$ and $(idx_b^t \to idx_e^t) \in \tau(T_l)$.

Definition 19 defines the concept of the overlap set which groups progression links from different intentions that reach the same agent program. In detail, the criterion (1) requires that there are same actual programs to be accomplished, whereas the criterion (2) ensures that these same actual agent programs need to be achieved in the different intentions. The following is an example to show what an overlap set looks like:

Example 10. (Example 9 continued). The overlap set of intentions $\{T_1, T_2\}$ has two elements (a) and (b) as follows:

(a) $\langle\langle(P_1^{T1} \to a_1^{P_1,1,T_1}), (P_2^{T_2} \to a_1^{P_2,1,T_2})\rangle\rangle$ where $J(a_1^{P_1,1,T_1}) = J(a_1^{P_2,1,T_2}) = a_1$.

(b) $\langle\langle(a_2^{P_1,2,T_1} \to a_4^{P_1,3,T_1}), (a_3^{P_2,2,T_2} \to a_4^{P_2,3,T_2})\rangle\rangle$ where $J(a_4^{P_1,3,T_1}) = J(a_4^{P_2,3,T_2}) = a_4$.

Proposition 3. Computing the overlap set of intentions can be done using at most an $O(n!) \times O(n^3 \times log(n))$ algorithm.

Proof. (sketch) Iterating all elements in a single execution trace of an intention is an $O(n)$ operation. Looking up the overlap between two execution traces is an $O(log(n))$ operation. Since each intention may have more than one execution traces, looking up the overlap between two intentions is an $O(n \cdot n \cdot nlog(n)) = O(n^3 log(n))$ operation. The permutation of any two intention is an $O(n!)$ operation. Finally, the overall complexity is an $O(n!) \cdot O(n^3 log(n))$ operation. ■

We note that the most naive implementation of the algorithm to compute the overlap set of intentions can have factorial time complexity. However, the discussion of optimisation for such an algorithm is out of the scope of this thesis. In Section 7.3 in Chapter 7, we will discuss the potential existing optimisation techniques for handling the same level of complexity as the future work.

To enable the BDI agent to merge the overlapping programs of a set of intentions, we now define the concept of the *overlap progression link* to formalise the action of intention merging:

Definition 20. Let an element of overlap set of a set of intentions $\{T_1, \ldots, T_m\}$ $(2 \le m)$ be $\langle\langle(idx_b^1 \to idx_e^1), \ldots, (idx_b^k \to idx_e^k)\rangle\rangle$ $(2 \le k \le m)$. For such an element of overlap set, we can have a corresponding overlap progression link $(\{idx_b^1, \ldots, idx_b^k\} \to \{idx_e^1, \ldots, idx_e^k\}) \in \{T_1, \ldots, T_m\}$.

Definition 20 says that each element of the overlap set amounts to an overlap progression link. Given an overlap progression link $(\{idx_b^1, \cdots, idx_b^k\} \to \{idx_e^1, \cdots, idx_e^k\})$, it essentially merges all primitive progression links $(idx_b^i \to idx_e^i)$ and can progress from the (b)eginning indexes $idx_b^1, \cdots, idx_b^k$ all the way to its (e)nding indexes $idx_e^1, \cdots, idx_e^k$ $(2 \le k \le m)$.

Example 11. (Example 10 continued). The overlap progression links of $\{T_1, T_2\}$ are (a') and (b') for the two element of the overlap set (a) and (b), respectively, as follows:

(a') $(\{P_1^{T_1}, P_2^{T_2}\} \to \{a_1^{P_1,1,T_1}, a_1^{P_2,1,T_2}\})$ for (a) $\langle\langle(P_1^{T1} \to a_1^{P_1,1,T_1}), (P_2^{T_2} \to a_1^{P_2,1,T_2})\rangle\rangle$;

(b') $(\{a_2^{P_1,2,T_1}, a_3^{P_2,2,T_2}\} \to \{a_4^{P_1,3,T_1}, a_4^{P_2,3,T_2}\}$ for (b) $\langle\langle(a_2^{P_1,2,T_1} \to a_4^{P_1,3,T_1}), (a_3^{P_2,2,T_2} \to a_4^{P_2,3,T_2})\rangle\rangle$.

Finally, we introduce the size of an overlap progression link as the number of the primitive progression links it merges. Given this quantitative measure of progression links, we can ensure the agent to merge as many intentions as possible.

Definition 21. Let an overlap progression link be $\alpha^o = (\{idx_b^1, \ldots, idx_b^k\} \to \{idx_e^1, \ldots, idx_e^k\})$. We say that the size of α^o is $size(\alpha^o) = k-1$ (i.e. merging $k-1$ extra primitive progression links). By default, the size of a primitive progression link α^p is $size(\alpha^p) = 0$ (i.e. no merging at all).

We close this section by noting that what we have done so far is to make the preparations for transforming the intention interleaving problem into an FPP problem. In particular, we introduce the overlap progression links of a given set of intentions to formalise the intention merging. In the following section, we will formally represent the intention interleaving problem as an FPP problem and incorporate the overlap progression links in FPP to facilitate maximal intention merging if possible.

6.2.3 Intention Interleaving Planning Formalism

In this section, we incorporate the overlap information in Section 6.2.2 in FPP to facilitate intention merging. We now represent the problem of intention interleaving as an FPP problem.

Definition 22. An FPP problem of interleaving intentions $I = \{T_1, \ldots, T_m\}$ is $\Omega = \langle \Sigma, N_{id}, O, s_0, S_G \rangle$ where:

- Σ is a finite set of (propositional) atoms;
- $N_{id} = \bigcup_{j=1}^{m} T_j(N_\vee \cup N_\wedge)$ is a set of node indexes of I;
- $O = O^p \cup O^o$ is a set of progression links.
- $s_0 = \mathcal{B}_0 \cup z_o \in 2^\Sigma \cup 2^{N_{id}}$ is the initial state;
- $S_G = \{z_g \mid z_g \rhd_{tn} I\} \subseteq 2^{N_{id}}$ is the goal state;

where O^p (reps. O^o) denotes the collection of primitive (resp. overlap) progression links of a set of intentions I while z_0 (reps. z_g) stands for the initial (reps. terminal) node set of I.

Definition 22 says an initial state s_0 is a finite set of (propositional) atoms encoding an initial belief base $\mathcal{B}_0$ and the initial node set z_0 of intentions I, whereas the goal state S_G encodes the terminal node set z_g of intentions I. The set of progression links O captures the state transitions, e.g. the indexes in the execution traces. The progression link $\alpha \in O$ is of the form $\langle pre(\alpha), del(\alpha), add(\alpha) \rangle$ where $pre(\alpha)$, $del(\alpha)$, and $add(\alpha)$ are called the precondition, delete-list, and add-list, respectively. The precondition, delete-list, and add-list are sets of atoms and node indexes in which the delete-list (resp. add-list) specifies which atoms and node indexes are removed from (resp. added to) the state of the specification. Table 6.1 gives the Stanford Research Institute Problem Solver (STRIPS) representation of primitive progression links in O^p where idx_b is the beginning node index. For example, the progression link $(idx_b \to P^T)$ in Table 6.1 captures the transition from idx_b to a plan P^T. The precondition of applying progression link

$(idx_b \rightarrow P^T)$ says that the context condition of P^T is being met and the agent currently is at the node idx_b (i.e. $idx \cup \varphi \in pre(\alpha^p)$). The progression link of $(idx_b \rightarrow a^{P,j,T})$ and $(idx_b \rightarrow G^{P,j,T})$ can be similarly explained.

Table 6.1: STRIPS Progression Links

link α^p	$pre(\alpha^p)$	$del(\alpha^p)$	$add(\alpha^p)$
$(idx_b \rightarrow P^T)$	$idx_b \cup \varphi$	$\{idx_b\}$	$\{P^T\}$
$(idx_b \rightarrow a^{P,j,T})$	$idx_b \cup \psi(a^{P,j,T})$	$\phi^- \cup \{idx\}$	$\phi^+ \cup \{a^{P,j,T}\}$
$(idx_b \rightarrow G^{P,j,T})$	idx_b	$\{idx\}$	$\{G^{P,j,T}\}$

Definition 23. Let an overlap progression link be $\alpha^o = (\{idx_b^1, \ldots, idx_b^k\} \rightarrow \{idx_e^1, \ldots, idx_e^k\}) \in O^o$ where $\alpha_i^p = (idx_b^i \rightarrow idx_e^i) \in O^p$ $(1 \le i \le k)$. We can have the precondition, delete-list, and add-list of α^o, namely $\langle pre(\alpha^o), del(\alpha^o), add(\alpha^o)\rangle$ such that the followings hold:

- $pre(\alpha^o) = pre(\alpha_1^p) \cup \ldots \cup pre(\alpha_k^p)$;
- $del(\alpha^o) = del(\alpha_1^p) \cup \ldots \cup del(\alpha_k^p)$;
- $add(\alpha^o) = add(\alpha_1^p) \cup \ldots \cup add(\alpha_k^p)$.

Definition 23 confirms that the overlap progression link α^o essentially merges related primitive progression links $\alpha_i^p = (idx_b^i \rightarrow idx_e^i)$ $(1 \le i \le k)$ into one. Therefore, the precondition, delete-list, and add-list of α^o are the conjunction of precondition, delete-list, and add-list of α_i^p, respectively.

Definition 24. The result of applying a progression link $\alpha \in O$ to a state $s = \mathcal{B} \cup z$ is described by the transition function $f : 2^\Sigma \cup 2^{N_{id}} \times O \rightarrow 2^\Sigma \cup 2^{N_{id}}$ defined as follows:

$$f(s,\alpha) = \begin{cases} (s \setminus del(\alpha)) \cup add(\alpha) & if\, s \models pre(\alpha) \\ undefined & otherwise \end{cases}$$

Hence we have the result of applying a sequence of progression links to a state specification s defined inductively:

$$Res(s, \langle\rangle) = s$$
$$Res(s, \langle \alpha_0; \ldots; \alpha_n \rangle) = Res(f(s,\alpha_0), \langle \alpha_1; \ldots; \alpha_n \rangle)$$

We now formally define the solution to our planning problem of intention interleaving as follows:

Definition 25. A sequence of progression links $\rho = \langle \alpha_0; \alpha_1; \ldots; \alpha_n \rangle$ is a solution to a planning problem $\Omega = \langle \Sigma, N_{id}, O, s_0, S_G \rangle$, denoted as $\rho = sol(\Omega)$, iff $Res(s_0, \rho) \models S_G$. We also say that ρ is optimal if the sum of the size of the progression link $size(\alpha_i)$ is maximum where $i = 0, \ldots, n$.

Definition 25 says the solution to a planning problem in Definition 22 is a sequence of progression links ρ which, when applied to the initial state specification using the *Res* function, reach a state that supports the terminal specification. The optimal solution is the solution which not only accomplishes all intentions but also merges the highest number of primitive progression links (see in Definition 21). To ensure that the optimal solution of intention interleaving planning problem is indeed corresponding to maximal-merged execution of the same set of intentions, We now formally establish the equivalence of these two.

Theorem 4. *Let $I = \{T_1,\ldots,T_m\}$ be a set of intentions and $\Omega = \langle\Sigma, N_{id}, O, s_0, S_G\rangle$ be its corresponding intention interleaving planning problem. We have a maximal-merged trace σ^m of intentions $I = \{T_1,\ldots,T_m\}$ if and only if there exists an optimal solution ρ to Ω.*

Proof. Suppose that there exists a maximal-merged trace σ^m of intentions $I = \{T_1,\ldots,T_m\}$. Hence, σ^m is also a conflict-free trace according to Definition 16 and Definition 17. Therefore, the terminal nodes of intentions I can be achieved. By the construction of the planning problem Ω, we can infer that the goal state S_G can be reached (i.e. there exists a solution). From Definition 21, we can see that by definition the number of merged primitive progression links is the size of a progression link, i.e. *size*(α_i). Hence, there also exists an optimal solution according to Definition 25. For the other side, let the optimal solution ρ be $\alpha_0;\ldots;\alpha_n$ such that $\alpha_i = (\{idx_b^{i_1},\ldots,idx_b^{i_k}\},\{idx_e^{i_1},\ldots,idx_e^{i_k}\}$ where $i = 0,\ldots,n$ and $k = 1,\ldots,m$. We can construct an execution trace σ in the following steps: (1) sequentialise α_i into $\alpha'_i = idx_b^{i_1};\ldots;idx_b^{i_k};idx_e^{i_1};\ldots;idx_e^{i_k}$; (2) remove any duplicate beginning indexes in $\sigma = \alpha'_0;\ldots;\alpha'_n$; (3) reduce subsequence $idx_e^{i_1};\ldots;idx_e^{i_k}$ in σ into $idx_e^{i_1}$; (4) retrieve the actual node of indexes in σ (see in Section 6.2.2). Finally, we can say σ is maximal-merged by contradiction. To be precise, if σ were not maximal-merged, then we would have ρ were not the optimal solution (which contradicts the assumption). ■

So far, what we have discussed is known as *offline planning*, i.e. a complete plan is generated and then executed in full. However, the environment is dynamic and pervaded by uncertainty. It may imply that the change of the environment (e.g. exogenous events can occur) would block the execution of the complete plan generated from FPP. For example, in a smart home environment, there is an intelligent domestic robot which finished chores in the lounge and needs to move to the hall doing chores. The robot chooses a plan which needs to pass through the hallway door to reach the hall. However, the pet dog accidentally slammed the door shut before the robot reaches the hallway door. As a consequence, this plan would be undesirably blocked. In BDI agents, when an execution failure occurs, the agent will backtrack to the related motivating goal and tries another applicable plan to achieve such a goal. Therefore, different from the classical replanning which replanning takes place right from the current state where the execution failure happens, the BDI agent propagates the failure to its higher-level goal first. Therefore, for intention interleaving replanning, we need the prefix steps which backtracks to the higher-level goal and modifies the

initial node. The steps of replanning are given in Algorithm 2 in which, e.g. **line 5-7** instruct the procedures for failure backtracking and initial node state modification.

Example 12. In Figure 6.2, if the agent is currently at the node a_4 and is no longer able to progress to a_5 (e.g. the environment changed unexpectedly). Then the agent should go back to its motivating goal G_3 and start replanning from there. Correspondingly, for its planning problem Ω the initial state $s_0 = \mathcal{B}_0 \cup \{a_4\}$ updates to $s_0 = \mathcal{B}_0 \cup \{G_3\}$ for replanning.

Algorithm 2: Intention Interleaving Replanning

Input: Planning problem $\Omega = \langle \Sigma, N_{id}, O, s_0, S_G \rangle$

1. $\alpha_0;\ldots;\alpha_n \leftarrow sol(\Omega)$ /* FPP solution */
2. $i \leftarrow 0, \alpha \leftarrow \alpha_0, s \leftarrow s_0$ /* initialisation */
3. **while** $s \notin \Upsilon$ **do**
4.   **if** $f(s,\alpha) = undefined$ **then**
5. $idx_b \leftarrow$ BEGINNING-INDEX(α)
6. $G \leftarrow$ BACKTRACK(idx_b) /* backtrack */
7. $s_0 \leftarrow \mathcal{B} \cup z \setminus \{idx_b\} \cup \{G\}$ /* modify state */
8. $sol'(\Omega) \leftarrow$ FPP($\langle \Sigma, X, O, s_0, S_G \rangle$) /* replan */
9. $\alpha_0;\ldots;\alpha_n \leftarrow sol'(\Omega)$
10. $\alpha \leftarrow \alpha_0, i \leftarrow 0$ /* re-initialisation */
11.   EXECUTE α
12. $s \leftarrow f(s,\alpha)$
13. $i \leftarrow i+1$
14. $\alpha \leftarrow \alpha_{i+1}$

6.3 Intention Interleaving Planning Implementation

In this section, we provide the practical implementation of our FPP approach in Planning Domain Definition Language (PDDL) representation [MGH+98] which consists of two parts: (i) an *operator file* which contains the STRIPS-like progression links; (ii) a *fact file* which encodes the initial/goal state description.

Operator File: We start with encoding the primitive progression link in PDDL in an *operator file*, namely ($idx_b \rightarrow P^T$), ($idx_b \rightarrow a^{P,j,T}$), and ($idx_b \rightarrow G^{P,j,T}$) according to Table 6.1 in Section 6.2.3. Note PDDL definitions require predicates. For legibility of presentation, however, we simply use the relevant mathematical symbols as syntactic sugar. Therefore, we can have the following list of actions in PDDL.

(:action ($idx_b \rightarrow P^T$)
: precondition (and idx_b $context(P)$)
: effect (and (not idx_b) P^T))
(:action ($idx_b \rightarrow a^{P,j,T}$)
: precondition (and idx_b $\psi(a^{P,j,T})$)
: effect (and (not ϕ^-) ϕ^+ (not idx_b) $a^{P,j,T}$))

(`:action` $(idx_b \rightarrow G^{P,j,T})$
: `precondition` idx_b
: `effect` (and (not idx_b) $G^{P,j,T}$))

We now encode the overlap progression links in PDDL in an *operator file*. Let an overlap progression link be $\alpha^o = (\{idx_b^1,\dots,idx_b^k\} \rightarrow \{idx_e^1,\dots,idx_e^k\})$ where the primitive progression link $\alpha_i^p = (idx_b^i \rightarrow idx_e^i)$ $(1 \leq i \leq k)$. Therefore, we have the following:
(`:action` $(\{idx_b^1,\dots,idx_b^k\} \rightarrow \{idx_e^1,\dots,idx_e^k\})$)
: `precondition` (and $pre(\alpha_1^p)\dots,pre(\alpha_k^p)$)
: `effect` (and $add(\alpha_1^p)\dots add(\alpha_k^p)$
(not $del(\alpha_1^p)$) ... (not $del(\alpha_k^p)$)
(`increase (efficiency-utility)` $size(\alpha^o)$))))

where the syntax (`increase (efficiency-utility)` $size(\alpha^o)$) specifies the reward of the progression link to be its size (i.e. increase $size(\alpha_o)$ so-called total-efficiency[2]). We also note that despite that PDDL language supports both of minimisation and maximisation, most planners only support the minimisation while in our work, we use maximisation. Therefore, in practice, we overcome this issue by assigning the primitive progression link with the highest value. Depending on how many extra primitive progression links an overlap progression link merges, the value of the overlap progression link decreases correspondingly.

Fact File: The *fact file* includes the initial state description and the goal state description. We start by declaring the objects present in the planning problem instance. The objects consist of all indexes of elements of all execution traces besides other ground belief atoms.
(`:objects` $\forall x \in X$, $\forall$ BELIEF_ATOMS $\in \Sigma$)

The initial condition consists of initial belief base $\mathcal{B}_0$ and the top-level goals of intentions.
(`:init` $\mathcal{B}_0, \forall T \in I, T(\bar{n})$)

The goal for the planning problem is to reach any terminal node of each intention in $\{T_1,\dots,T_m\}$.
(`:goal` (and (or tn_1^1 ... $tn_{k_1}^1$) ... (or $tn_1^m \dots tn_{k_m}^m$))
where $\{tn_1^j,\dots,tn_{k_j}^j\}$ $((1 \leq j \leq m))$ is the terminal node set of the intention T_j and the syntax 'or' means that reaching any of the terminal nodes $\{tn_1^j,\dots,tn_{k_j}^j\}$ would achieve the intention T_j (i.e. `:disjunctive-preconditions` requirement in PDDL).

Finally, we show how to obtain a maximal-merged execution trace through the optimisation in PDDL. To do so, we add a fluent function `(:function(efficiency-utility))` to keep track of the efficiency utility with an initial efficiency utility specification `(=(efficiency-utility)0)`. Then we add a `:metric` section to the *fact file* with `(:metric maximise(efficiency-utility))` to specify that maximising the sum of efficiency-utility is the objective.

[2]It is noted that the vast majority of state-of-the-art planners only support syntax `total-cost`. For readability, we still use the syntactic sugar `efficiency-utility` for self-explanatory purpose.

Table 6.2: Effectiveness Analysis of Approach

	2.1	2.2	3.1	3.2	3.3	4.1	4.2	4.3	4.4
2	17%	33%	11%	22%	33%	8%	17%	25%	33%
3	22%	44%	15%	30%	44%	11%	22%	33%	44%
4	25%	50%	17%	33%	50%	13%	25%	38%	50%
5	27%	53%	18%	36%	53%	13%	27%	40%	53%
6	28%	56%	19%	37%	56%	14%	28%	42%	56%
7	29%	57%	19%	38%	57%	14%	29%	43%	57%
8	29%	58%	19%	39%	58%	15%	29%	44%	58%

6.4 Intention Interleaving Planning Evaluation

In this section, we present some preliminary effectiveness results to show the feasibility of our approach. Consider a manufacturing scenario of using machining operations to make holes in a metal block. There are several different kinds of hold-creation operations (e.g. twisting-drilling and spade-drilling) available, as well as several different kinds of hole-improvement operations (e.g. reaming and boring). Each time the robotic arm switches to a different kind of operation or to a hole of different diameter, it must mount a different cutting tool on its arm. If the same cutting operation is to be performed on two (or more) holes of the same diameter, then these same operations can be merged by omitting the repetitive task of changing the cutting tools.

We generate such manufacturing scenarios in which the detailed design were varied by: (i) the number of blocks (n from 2 to 8); (ii) operations per blocks (m from 2 to 4), and (iii) the maximal number of overlap operations among all metal blocks (k from 1 to 4), resulting in 63 test cases in total. We assume that each operation has three actions, e.g. twisting-drilling task needs (i) action of taking on a twisting-drill, (ii) actual twisting-drilling action, (iii) action of taking off this twisting-drilling. For simplicity, the shared operations among a set of blocks are in the same order in each metal block. For example, if block 1 and block 2 share both twisting-drilling and reaming operation, we would expect the twisting-drilling operation before reaming operation in both blocks in practice. The dataset and instructions for reproduction are available online[3]. These cases were then solved via our FPP approach where a planner called **Metric-FF**[4] is employed.

Table 6.2 shows the effectiveness results of our approach where rows are the number of metal blocks n from 2 to 8 and columns $m.k$ reads as there are m operations among which there are k overlapping operations. Compared to the default approach without capitalising on overlapping operations, our FPP approach not only successfully achieves all the intentions, but also reduces

[3]https://github.com/Mengwei-Xu/manufacturing-evaluation
[4]https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html

the amount of repetitive task of changing the cutting tools. The value in the table is the improved efficiency defined as the reduced number of actions divided by the total number of actions if without merging identical operations. For example, if there are 4 metal blocks, 3 operations for each metal, and 2 overlapping operations over these 3 operations, our approach can improve the efficiency by 33%, i.e. reducing 12 repetitive changing tool actions out of 36 actions in total if without intention merging. We also observe the efficiency to increase with the number of blocks (see in each column). When all operations for all blocks are the same, the efficiency is the same regardless of the number of blocks (see the same efficiency values in column 2.2, 3.3, and 4.4).

6.5 Conclusion

In this chapter, we have developed mechanisms that manage the interactions between the intentions of an agent in a rational manner. Specifically, we employ an off-the-shelf FPP planner to address the problem of the concurrent intention executions in BDI agents. In spite of at least the PSPACE complexity of FPP, the past several decades have witnessed the tremendous progress in the planning community in which large planning problems can be solved in the real time [GB13]. Furthermore, our approach of treating the planner as a black box allows us to immediately harvest from any ongoing performance improvement made to these planners. Since these intentions can interact with each other both positively and negatively, our planning-centric approach to BDI agents can also avoid negative interactions while facilitating positive interaction. To do so, it guarantees the accomplishment of intentions by finding a conflict-free execution trace among a set of intentions modelled as a goal-plan tree. Regarding the positive interaction, we focus on a specific type of positive interaction, namely identical sub-intentions among different intentions. To identify and facilitate such positive interaction, we introduce the concept of overlaps and incorporate it into FPP to minimise the cost of execution via merging identical sub-intentions, thus improving the overall execution efficiency of the BDI agents. Furthermore, the mechanism we developed to manage the interactions between intentions is in a domain-independent way, which can be integrated into the infrastructure of agent development systems. Our manufacturing experiment results indicate the effectiveness of our approach when compared to BDI agents that do not harness the advantages of commonality between intentions. Therefore, integrating our approach into practical agent systems allows building agents that are more sensible and productive in the way they pursue concurrent intentions.

CHAPTER 7

CONCLUSION

In this thesis, we have addressed the dissatisfaction related to features of the execution robustness, environment adaptivity, and intention progression efficiency in the current approaches to Belief-Desire-Intention (BDI) agents. Firstly, the deficiency in execution robustness can hamper the ability of a BDI agent to cope with failure during executing, thus limiting its applicability when, e.g. no pre-defined plan either worked or exists. Secondly, the absence of adaptivity further prevents the long-term employment of a BDI agent in an environment which changes over time. Thirdly, the deficit in execution efficiency of BDI agents renders themselves less-received by manufacturing sectors which operate in a resource-critical domain. To address the lack of these features in BDI agents, this thesis has been devoted to building towards BDI agents within a classical BDI agent programming language, namely Conceptual Agent Notation (CAN), which can (i) create new plans when either no pre-defined plan worked or existed to achieve goals; (ii) adapt to a fast-changing environment with a plan library evolution architecture with a mechanism to incorporate new plans and drop old or unsuitable plans; (iii) think ahead to not only guarantees the achievability of intentions, but also reduce the overall cost of intention execution by exploiting potential common sub-intentions.

7.1 Planning in BDI

In Chapter 4, we embedded First-principles Planning (FPP) in a popular BDI language, namely CAN agents, in a way that reuses and respects the procedural domain knowledge in the plan library. To do so, we partition the original intention set of a BDI agent into the procedural intention and declarative intention set. Such a partition retains the standard procedural intentions for the normal BDI reasoning while enabling the new declarative intention set to be achieved by FPP. Also, a novel concept of pure declarative goal for FPP, namely $goal(\varphi_s, \varphi_f)$, is inspired

by the existing normal declarative goal, i.e. $goal(\varphi_s, P, \varphi_f)$. This pure declarative goal for FPP succinctly articulates what FPP should achieve and when it should halt. We then incorporate FPP into the CAN languages. Unlike any other previous attempts which integrate FPP with BDI agents in a rigid or ad-hoc style, our approach defined a comprehensive operational semantics that specifies the behaviours of a BDI agent along with an FPP on-demand. This operational semantics intuitively specifies when and how the FPP can be utilised for the benefits of BDI agents in both offline and online settings. Furthermore, we have theoretically demonstrated the intuitive expectation of integration of BDI and FPP. Our feasibility case also showed that the combined architecture nicely obtains the key advantages of both BDI and FPP in a well-balanced manner, i.e. the improvement of the scalability of the existing BDI agents and insurance of maximum reactiveness for most of the standard procedural intentions.

In Chapter 5, we looked at the long-term employment of planning-extended BDI agents which have the ability to reuse plans when similar goals need to be achieved, and to improve domain knowledge using past experience. To achieve so, we presented our preliminary theoretical exploration of adaptive BDI agents which can be well-suited in a fast-changing and uncertain environment. In order to be more adaptive, we enabled a BDI agent to reason about the plan library to add and remove plans. In this way, a BDI agent can incorporate new plans and remove old ones based on their performance and the structure of plans. Our proposal also defined these performance and structural properties of plans that can be used to formalise plan library modification. The plan library modification consists of extension (i.e. adopt new plans) and contraction (i.e. delete old plans) of the plan library, and plan library expansion and extraction are performed when necessary. Finally, we presented and instantiated a specific contract operator which we proved satisfying our postulates.

In conclusion, the challenges of embedding in BDI agents ultimately lies in the richness of the interactions between BDI agents and the environment in which they are situated. If the environment is always cooperative, the agent, in theory, should be able to complete its intentions without any problems. However, the environment is often dynamic and uncertain in real-life applications. Meanwhile, the technical difficulties of embedding in BDI agents first come from identifying the appropriate triggers of planning, and then the proper management of the execution of plans generated by the planner(s) in the context of existing agent programs. In our approach here, the triggers of planning are focused on two types of events, namely plan failure and new opportunities. In the case of plan failure detection, planning mostly does "plan repair" to ensure the execution applicability to be restored for the blocked plans. For opportunity recognition, what our approach enables is essential to allow the agent to pursue a goal which is scheduled to pursue, but can only be pursued when some condition holds. The contribution of evolving the knowledge of the BDI agents comes right naturally to utilise the pre-cached plans generated by the planner(s) throughout the agent employment phase. Once the rationale of expansion of new knowledge is established, the knowledge contraction follows naturally. Our

approach of plan library evolution, to some extent, is rather focused on the principles of such a plan library modification process rather than, e.g. the actually methods of transforming the plans from planning to BDI plans. Of course, it is no doubt that the methods of converting the plans from planning to most suitable BDI plans remain a vital – albeit challenging – research problem. We will elaborate it in slightly more details in Section 7.3.

7.2 Managing Multiple Intentions

In Chapter 6, we looked at a different scenario when the agent is pursuing multiple intentions. As we have shown, these intentions can interact either negatively when the interactions cause one or more goals to fail, or positively when there are the overlaps between intentions. Managing these interactions is difficult in BDI paradigm as it is generally not possible to decide which plans to use in advance as they depend on (dynamic) environment conditions. For example, a non-conflict plan selected at the current step may cause the problem later on. However, it is desirable for an intelligent agent to act on various intentions in an interweaving manner. Therefore, an agent should consider these interactions and be rational in the way it pursues its intentions. To do so, we have shown that the task of intention interleaving can be managed by FPP in an automated style. We started with formalising the underlying hierarchy of a plan library into an AND/OR tree. In the context of AND/OR tree, we transformed the problem of desirable intention interleaving into a path-finding problem. In detail, the requirement of avoiding negative interactions is equivalent to obtaining a conflict-free trace of a set of intentions. Meanwhile, merging the overlapping intentions becomes the task of searching for a maximal-merged trace. After this transformation, off-the-shelf FPP tools can be used to identify, e.g. a conflict-free trace. Finally, our evaluation in a manufacturing scenario demonstrates the effectiveness of our approach when compared to BDI agents that do not harness the advantages of commonality between intentions.

In conclusion, it is expected and indeed, a key feature of any reactive agent to pursue multiple intentions. To be considered intelligent, an agent should be sensible and smart in the way it pursues its multiple intentions. Of course, the least which the agent should do is to ensure that all of its multiple intentions should be achieved in the end. In our approach, on the top of securing the achievability of multiple intentions by default, we advance it on the exploitation of synergy among multiple intentions. Such a focus can be metaphorically captured by the saying of “killing two birds with one stone". Indeed, our approach is beneficial, in particular, in the domain where the resource is limited, and there are similar but yet slightly different tasks going on. To achieve so, a large part of work is devoted to identifying the potential opportunities exploitation of synergy. Therefore, some level of computational effort needs to be asserted beforehand to secure some level of execution efficiency (through facilitating synergy) as a fair exchange. Also, since the new intentions will still be committed in an online fashion, it implies that the identification of potential synergy exploitation should be made online too. Some insights into future work are

given in Section 7.3.

7.3 Discussion and Future Work

In Chapter 4, although the functionalities and behaviours of planning can generic in BDI agents, the semantics that we have developed only applies to a specific type of BDI agents, namely AgentSpeak and CAN. It implies that a different type of BDI agent (e.g. Artificial Autonomous Agents Programming Language (3APL)) may need a new set of semantic rules. Therefore, firstly it seems both natural and plausible to investigate the new semantics of planning for other popular BDI agent frameworks as the future work. Secondly, despite the existence of a prototype-like feasibility study of the combined FPP and BDI system, the development of full implementation and its thorough evaluation are not available. Indeed, such a full implementation would amount to a considerable amount of (pure) software engineering work. The current potential software design would be as follows. On the BDI side, the current BDI reasoning cycle itself needs to be minimally modified when the execution failures occur according to the new semantic rules. Outside of BDI agents, when a pure declarative goal is generated, it can be bound to an API which first transforms the current beliefs and plan library into Planning Domain Definition Language (PDDL) files which then are passed to a planner to solve. The consistency management of declarative intention itself can be all defined similarly to the belief base. Furthermore, a full evaluation of any implementation would require a problem setting considerably larger than our domestic robot scenario. Therefore, before any evaluation, a large set of problem cases should be collected. The current existing International Planning Competition (IPC) planning problem set can be a good starting point for the planning problem to which a planner can solve to recover the related failure recoveries of BDI agents.

In Chapter 5, the framework arguably serves as a purely theoretical study of desirable properties of plan library expansion and contraction for BDI architecture. Therefore, the next step for this work is to check the presented ideas are practically realisable. For example, there is no concept of the time points in most existing BDI platforms. However, a quick solution to this time point problem can be the natural number of reasoning cycle of a BDI agent. Also, not only do time and space complexity of various measures in this work need to be further investigated, but also their corresponding complete and tractable algorithms need to be presented. Furthermore, the current framework remains agnostic regarding when the plan library evolution should start. Ideally, an autonomous agent should be able to maintain its knowledge base on its own. Therefore, a comprehensive triggering mechanism for plan library evolution is a promising line of future work. Finally, provided the integration of FPP in BDI agents in Chapter 4 agents, it is worthwhile investigating how to transform the plan generated by a planner into a suitable BDI plans for recovering the similar failure if it occurs again in the future. A promising venture would be the case-based planning [Spa01] which specialises in the reuse of past successful plans in order to

solve new planning problems.

In Chapter 6, although generic, our approach of employing planning to manage the intention progression does come with some restrictions to the type of BDI agents for applications. First, we do not allow that there is a loop in the plan library. Such a loop would cause an infinitely long path. Secondly, despite supporting the concurrent pursue of multiple intentions, the current approach does not allow the parallel plan program in each intention, thus no nested-concurrence. Thirdly, the mechanism also does not consider other forms of subgoals such as maintenance goals (e.g. maintaining a belief true for a duration of time). For the future work, the work of [YTS17] sheds light upon the loops in the plan library. Regarding the maintenance goal, they effectively put constraints on the belief base of the agent for a certain amount of time. Therefore, the concept of state-trajectory constraints in planning community would be a good start point to realise the maintenance goal. Intuitively, the state-trajectory constraints are hard-constraints in the form of logic expressions, which should be true for the state-trajectory produced during the execution of a plan, which is a solution of the given planning problem. There is another limitation in this work regarding the computation of overlap sub-intentions among a set of intentions. A naive implementation of the algorithm to compute the overlap set of intentions has factorial time complexity. For future work, we believe the algorithm can be improved to incorporate hashing ideas, such as in [Ert17], to make the algorithm viable for large scale problems. Finally, a thorough evaluation of the costs and benefits of our approach also needs to be done empirically in a wider range of applications.

BIBLIOGRAPHY

[AGM85] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The journal of symbolic logic*, 50(2):510–530, 1985.

[AP06] Leila Amgoud and Henri Prade. Explaining qualitative decision under uncertainty by argumentation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 21–9, 2006.

[BBJ+02] R.H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. Agentspeak (XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In *Proceedings of the 1st International Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1302, 2002.

[BF97] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[BFM06] Meghyn Bienvenu, Christian Fritz, and Sheila A. McIlraith. Planning with qualitative temporal preferences. *Knowledge Representation*, 6:134–144, 2006.

[BHG06] Steve S. Benfield, Jim Hendrickson, and Daniel Galanti. Making a strong business case for multiagent technology. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent systems*, pages 10–15, 2006.

[BHW07] R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[BIP88] Michael Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(4):349–355, 1988.

[BLG97] Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference of American Association for Artificial Intelligence*, pages 714–719, 1997.

[BLH+14] Kim Bauters, Weiru Liu, Jun Hong, Carles Sierra, and Lluís Godo. CAN(PLAN)+: Extending the operational semantics of the BDI architecture to deal with uncertain

information. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence*, pages 52–61, 2014.

[BM07] Jorge A. Baier and Sheila A. McIlraith. On domain-independent heuristics for planning with qualitative preferences. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, pages 7–12, 2007.

[BMH08] Max Bajracharya, Mark W Maimone, and Daniel Helmick. Autonomy for mars rovers: Past, present, and future. *IEEE Computer*, 41(12):44–50, 2008.

[BMH+16] Kim Bauters, Kevin McAreavey, Jun Hong, Yingke Chen, Weiru Liu, Llu Godoís, and Carles Sierra. Probabilistic planning in AgentSpeak using the POMDP framework. In *Combinations of Intelligent Methods and Applications*, pages 19–37. Springer, 2016.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[BPML04] Lars Braubach, Alexander Pokahr, Daniel Moldt, and Winfried Lamersdorf. Goal representation for bdi agent systems. In *Proceedings of the 2nd Workshop on Programming Multiagent Systems: Languages, Frameworks, Techniques, and Tools*, pages 44–65. Springer, 2004.

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[Bra87] Michael Bratman. *Intention, Plans, and Practical reason*. 1987.

[Bra90] Michael Bratman. What is intention. *Intentions in communication*, pages 15–32, 1990.

[Bra01] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[BS01] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1(8):445–532, 2001.

[CDB07] Bradley J. Clement, Edmund H. Durfee, and Anthony C. Barrett. Abstract reasoning for planning and coordination. In *Journal of Artificial Intelligence Research*, volume 28, pages 453–515, 2007.

[CL90] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial intelligence*, 42(3):213–261, 1990.

[CWDD17] Stephen Cranefield, Michael Winikoff, Virginia Dignum, and Frank Dignum. No pizza for you: value-based plan selection in BDI agents. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 178–184, 2017.

[Das08] Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248, 2008.

[DDBDM03] Mehdi Dastani, Frank De Boer, Frank Dignum, and John-Jules Meyer. Programming agent deliberation: an approach illustrated using the 3apl language. In *Proceedings of the 2nd international joint conference on Autonomous Agents and Multiagent systems*, pages 97–104. ACM, 2003.

[DEGL17] Ameneh Deljoo, Tom van Engers, Leon Gommans, and Cees de Laat. What is going on: Utility-based plan selection in BDI agents. In *Proceedings of Workshop on Knowledge-Based Techniques for Problem Solving and Reasoning*, pages 711–718, 2017.

[DI99] Olivier Despouys and François Félix Ingrand. Propice-plan: Toward a unified framework for planning and execution. In *Proceedings of the 5th European Conference on Planning*, pages 278–293, 1999.

[DW08] Khanh Hoa Dam and Michael Winikoff. Cost-based BDI plan selection for change propagation. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 217–224, 2008.

[DWP06] Khanh Hoa Dam, Michael Winikoff, and Lin Padgham. An agent-oriented approach to change propagation in software evolution. In *Proceedings of Australian Software Engineering Conference*, pages 309–318. IEEE Computer Society, 2006.

[DYAL14] Thu Trang Doan, Yuan Yao, Natasha Alechina, and Brian Logan. Verifying heterogeneous multi-agent programs. In *Proceedings of the 13th international conference on Autonomous Agents and Multiagent Systems*, pages 149–156, 2014.

[EHN94] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of Association for the Advancement of Artificial Intelligence*, pages 1123–1128, 1994.

[Ert17] Otmar Ertl. SuperMinHash- a new minwise hashing algorithm for jaccard similarity estimation. *arXiv preprint arXiv:1706.05698*, 2017.

[FEGS14] Edgardo Ferretti, Marcelo L. Errecalde, Alejandro J. García, and Guillermo R. Simari. A possibilistic defeasible logic programming approach to argumentation-based decision-making. In *Journal of Experimental & Theoretical Artificial Intelligence*, volume 26, pages 519–550, 2014.

[FN15] João Faccin and Ingrid Nunes. BDI-agent plan selection based on prediction of plan outcomes. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 2, pages 166–173. IEEE, 2015.

[GB13] Hector Geffner and Blai Bonet. *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[GFFOC13] Arturo González-Ferrer, Juan Fernández-Olivares, and Luis Castillo. From business process models to hierarchical task network planning domains. *The Knowledge Engineering Review*, 28(2):175–193, 2013.

[GL86] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. *Proceedings of the IEEE*, 74(10):1383–1398, 1986.

[GL87] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of Association for the Advancement of Artificial Intelligence*, volume 87, pages 677–682, 1987.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[GR13] Frédérick Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.

[GS04] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming: An argumentative approach. *Theory and practice of logic programming*, 4:95–138, 2004.

[GSS09] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Frontiers in Artifical Intelligence and Applications*, pages 633–654, 2009.

[HBHM98] Koen V. Hindriks, Frank S. de Boer, Wiebe Van Der Hoek, and John-Jules Ch Meyer. A formal embedding of AgentSpeak (L) in 3APL. In *Proceedings of Australian Joint Conference on Artificial Intelligence*, pages 155–166. Springer, 1998.

[HBHM99] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[HLPX17] Andreas Herzig, Emiliano Lorini, Laurent Perrussel, and Zhanhao Xiao. BDI logics for BDI architectures: old problems, new perspectives. *KI-Künstliche Intelligenz*, 31(1):73–83, 2017.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. In *Journal of Artificial Intelligence Research*, pages 253–302, 2001.

[IGR92] François F. Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for real-time reasoning and system control. In *IEEE expert*, volume 7, pages 34–44, 1992.

[JB03] Nicholas R. Jennings and Stefan Bussmann. Agent-based control systems. In *IEEE Control Systems*, volume 23, pages 61–73. IEEE, 2003.

[KBM+16] Ronan Killough, Kim Bauters, Kevin McAreavey, Weiru Liu, and Jun Hong. Risk-aware planning in BDI agents. In *Proceedings of the 8th International Conference on Agents and Artificial Intelligence*, pages 322–329, 2016.

[KE12] Thomas Keller and Patrick Eyerich. PROST: Probabilistic planning based on uct. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 2012.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. pages 99–134, 1998.

[KMS98] Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. Hybrid planning for partially hierarchical domains. In *Proceeding of Association for the Advancement of Artificial Intelligence*, pages 882–888, 1998.

[KNHD97] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of European Conference on Planning*, pages 273–285. Springer, 1997.

[Kow83] Robert Kowalski. Logic programming. In *Proceedings of International Federation for Information Processing Congress*, pages 133–145, 1983.

[KR76] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley & Sons, 1976.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of European Conference on Machine learning*, pages 282–293. Springer, 2006.

[LL15] Sam Leask and Brian Logan. Programming deliberation strategies in meta-APL. In *Proceedings of International Conference on Principles and Practice of Multi-Agent Systems*, pages 433–448. Springer, 2015.

[LVL02] Emmanuel Letier and Axel Van Lamsweerde. Deriving operational software specifications from system goals. *ACM SIGSOFT Software Engineering Notes*, 27(6):119–128, 2002.

[MDC+07] Stephen D. McArthur, Euan M. Davidson, Victoria M. Catterson, Aris L. Dimeas, Nikos D. Hatziargyriou, Ferdinanda Ponci, and Toshihisa Funabashi. Multi-agent systems for power engineering applications – Part I: Concepts, approaches, and technical challenges. In *IEEE Transactions on Power systems*, volume 22, pages 1743–1752, 2007.

[MGH+98] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. S. Weld, and D.E. Wilkins. PDDL-the planning domain definition language. 1998.

[ML07] Felipe Meneguzzi and Michael Luck. Composing high-level plans for declarative agent programming. In *Declarative Agent Languages and Technologies*, volume 4897, pages 69–85. Springer, 2007.

[ML08] Felipe Meneguzzi and Michael Luck. Leveraging new plans in AgentSpeak (PL). In *Proceedings of International Workshop on Declarative Agent Languages and Technologies*, pages 111–127, 2008.

[ML11] Jianbing Ma and Weiru Liu. A framework for managing uncertain inputs: An axiomization of rewarding. *International Journal of Approximate Reasoning*, 52(7):917–934, 2011.

[MLH+14] Jianbing Ma, Weiru Liu, Jun Hong, Lluís Godo, and Carles Sierra. Plan selection for probabilistic BDI agents. In *Proceedings of 26th International Conference of Tools with Artificial Intelligence*, pages 83–90. IEEE, 2014.

[MLVC98] Michael C. Móra, José G. Lopes, Rosa M. Viccariz, and Helder Coelho. BDI models and systems: Reducing the gap. In *Proceedings of International Workshop on Agent Theories, Architectures, and Languages*, pages 11–27. Springer, 1998.

[Mon82] George E. Monahan. State of the art – a survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[MS15] Felipe Meneguzzi and Lavindra de Silva. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. In *The Knowledge Engineering Review*, volume 30, pages 1–44. Cambridge University Press, 2015.

[MZM04] Felipe Rech Meneguzzi, Avelino Francisco Zorzo, and Michael da Costa Móra. Propositional planning in BDI agents. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 58–63. ACM, 2004.

[NF71] Nils J. Nilsson and Richard E. Fikes. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[NL14] Ingrid Nunes and Michael Luck. Softgoal-based plan selection in model-driven BDI agents. In *Proceedings of the 2014 international conference on Autonomous Agents and Multi-agent systems*, pages 749–756, 2014.

[OMT07] Wassila Ouerdane, Nicolas Maudet, and Alexis Tsoukias. Arguing over actions that involve multiple criteria: A critical review. In *Proceedings of European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 308–319, 2007.

[PBJ13] Alexander Pokahr, Lars Braubach, and Kai Jander. The Jadex project: Programming model. In *Multiagent Systems and Applications*, pages 21–53. Springer, 2013.

[PBL05] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. A goal deliberation strategy for BDI agent systems. In *German Conference on Multiagent System Technologies*, pages 82–93. Springer, 2005.

[Ped89] Edwin P. D. Pednault. Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[PG09] Hector Palacios and Hector Geffner. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, 35:623–675, 2009.

[Poe13] Ibo van de Poel. Translating values into design requirements. In *Philosophy and engineering: Reflections on practice, principles and process*, pages 253–266. Springer, 2013.

[PS13] Lin Padgham and Dhirendra Singh. Situational preferences for BDI plans. In *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems*, pages 1013–1020, 2013.

[Rao96] Anand S. Rao. AgentSpeak (L): BDI agents speak out in a logical computable language. In *Proceedings of European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer, 1996.

[RDM05] M. Birna van Riemsdijk, Mehdi Dastani, and John-Jules Ch. Meyer. Semantics of declarative goals in agent programming. In *Proceedings of the 4th international joint conference on Autonomous Agents and Multiagent Systems*, pages 133–140. ACM, 2005.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of Knowledge Representation*, pages 473–484, 1991.

[RM17] Gavin Rens and Deshendran Moodley. A hybrid POMDP-BDI agent architecture with online stochastic planning and plan caching. *Cognitive Systems Research*, 43:1–20, 2017.

[Rob92] John Alan Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, 1992.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[Sil17] Lavindra de Silva. BDI agent reasoning with guidance from HTN recipes. In *Proceedings of the 16th Conference on Autonomous Agents and Multiagent Systems*, pages 759–767, 2017.

[Sim72] Herbert A. Simon. Theories of bounded rationality. *Decision and organization*, 1(1):161–176, 1972.

[SP04] Lavindra de Silva and Lin Padgham. A comparison of BDI based real-time reasoning and HTN based planning. In *Proceedings of Australasian Joint Conference on Artificial Intelligence*, pages 1167–1173. Springer, 2004.

[SP05a] Lavindra de Silva and Lin Padgham. Planning as needed in BDI systems. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2005.

[SP05b] Lavindra de Silva and Lin Padgham. Planning on demand in BDI systems. In *International Conference on Automated Planning and Scheduling*. University of Southern California, 2005.

[SP06] Gerardo I. Simari and Simon Parsons. On the relationship between MDPs and the BDI architecture. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1041–1048. ACM, 2006.

[SP07] Sebastian Sardina and Lin Padgham. Goals in the context of BDI plan failure and planning. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 16–23, 2007.

[SP11] Sebastian Sardina and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. In *Autonomous Agents and Multi-Agent Systems*, volume 23, pages 18–70. Springer, 2011.

[Spa01] Luca Spalzzi. A survey on case-based planning. *Artificial Intelligence Review*, 16(1):3–36, 2001.

[SSP06] Sebastian Sardina, Lavindra de Silva, and Lin Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1001–1008, 2006.

[SSP09] Lavindra de Silva, Sebastian Sardina, and Lin Padgham. First principles planning in BDI systems. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*, volume 2, pages 1105–1112, 2009.

[SSP10] Dhirendra Singh, Sebastian Sardina, and Lin Padgham. Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems*, 58(9):1067–1075, 2010.

[SSPA10] Dhirendra Singh, Sebastian Sardina, Lin Padgham, and Stéphane Airiau. Learning context conditions for BDI plan selection. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 325–332, 2010.

[SV10] David Silver and Joel Veness. Monte-carlo planning in large POMDPs. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[SWH06] Weiming Shen, Lihui Wang, and Qi Hao. Agent-based distributed manufacturing process planning and scheduling: a state-of-the-art survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(4):563–577, 2006.

[SWP02] Martijn Schut, Michael Wooldridge, and Simon Parsons. On partially observable MDPs and BDI models. In *Proceedings of Foundations and Applications of Multi-Agent Systems*, pages 243–259. Springer, 2002.

[TP11] John Thangarajah and Lin Padgham. Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning*, 47(1):17–56, 2011.

[TPW03a] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 2003.

[TPW03b] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *Proceedings of the 2nd International Conference on Autonomous Agents and Multiagent Systems*, pages 401–408. ACM, 2003.

[TSP12] John Thangarajah, Sebastian Sardina, and Lin Padgham. Measuring plan coverage and overlap for agent reasoning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 1049–1056, 2012.

[TWPF02] John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer. Avoiding resource conflicts in intelligent agents. In *Proceedings of European Conference on Artificial Intelligence*, volume 2, pages 18–22, 2002.

[VTH11] Simeon Visser, John Thangarajah, and James Harland. Reasoning about preferences in intelligent agent systems. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011.

[WBPL06] Andrzej Walczak, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Augmenting BDI agents with deliberative planning techniques. In *Proceedings of International Workshop on Programming Multi-Agent Systems*, pages 113–127. Springer, 2006.

[Wil90] David E. Wilkins. Can AI planners solve practical problems? *Computational intelligence*, 6(4):232–246, 1990.

[Wil13] David Willetts. Eight great technologies. *Policy Exchange*, 2013.

[Win05] Michael Winikoff. An AgentSpeak meta-interpreter and its applications. In *Proceedings of International Workshop on Programming Multi-Agent Systems*, pages 123–138. Springer, 2005.

[WMLW95] David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):121–152, 1995.

[WPHT02] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufman, 2002.

[WPS14] Max Waters, Lin Padgham, and Sebastian Sardina. Evaluating coverage based intention selection. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 957–964, 2014.

[WR11] Jason Wolfe and Stuart Russell. Bounded intention planning. In *Proc. of IJCAI'11*, pages 2039 – 2045, 2011.

[XBML18a] Mengwei Xu, Kim Bauters, Kevin McAreavey, and Weiru Liu. A formal approach to embedding first-principles planning in BDI agent systems. In *Proceedings of the 12th International Conference on Scalable Uncertainty Management (SUM'18)*, pages 333–347, 2018.

[XBML18b] Mengwei Xu, Kim Bauters, Kevin McAreavey, and Weiru Liu. A framework for plan library evolution in BDI agent systems. In *Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'18)*, pages 414–421, 2018.

[XMBL19] Mengwei Xu, Kevin McAreavey, Kim Bauters, and Weiru Liu. Intention interleaving via classical replanning. In *Proceedings of the 31st IEEE International Conference on Tools with Artificial Intelligence (ICTAI'19)*, pages 85–92, 2019.

[YL16] Yuan Yao and Brian Logan. Action-level intention selection for BDI agents. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems*, pages 1227–1236, 2016.

[YLT14] Yuan Yao, Brian Logan, and John Thangarajah. SP-MCTS-based intention scheduling for BDI agents. In *Proceedings of European Conference on Artificial Intelligence*, pages 1133–1134, 2014.

[YLT16a] Yuan Yao, Brian Logan, and John Thangarajah. Intention selection with deadlines. In *Proceedings of the 22nd European Conference on Artificial Intelligence*, pages 1700 – 1701, 2016.

[YLT16b] Yuan Yao, Brian Logan, and John Thangarajah. Robust execution of BDI agent programs by exploiting synergies between intentions. In *Proceedings of Association for the Advancement of Artificial Intelligence*, pages 2558–2565, 2016.

[YSL16] Yuan Yao, Lavindra de Silva, and Brian Logan. Reasoning about the executability of goal-plan trees. In *Proceedings of International Workshop on Engineering Multi-Agent Systems*, pages 176–191, 2016.

[YTS17] Nitin Yadav, John Thangarajah, and Sebastian Sardina. Agent design consistency checking via planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 458–464, 2017.

[ZRB16] Maicon Rafael Zatelli, Alessandro Ricci, and Rafael H. Bordini. Conflicting goals in agent-oriented programming. In *Proceedings of the 6th International Workshop*

on Programming Based on Actors, Agents, and Decentralized Control, pages 21–30. ACM, 2016.