A computational framework for behaviour adaptation: the case for agents and business processes

Yingzhi Gou
University of Wollongong
A COMPUTATIONAL FRAMEWORK FOR BEHAVIOUR ADAPTATION: THE CASE FOR AGENTS AND BUSINESS PROCESSES

A Thesis Submitted in Partial Fulfilment of the Requirements for the Award of the Degree of Doctor of Philosophy from UNIVERSITY OF WOLLONGONG

by

Yingzhi Gou
BCompSc, BCompSc Honours

School of Computing and Information Technology
Faculty of Engineering and Information Sciences
2018
© Copyright 2018
by
Yingzhi Gou
ALL RIGHTS RESERVED
CERTIFICATION

I, Yingzhi Gou, declare that this thesis, submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Computing and Information Technology, Faculty of Engineering and Information Sciences, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Yingzhi Gou
31 03 2018
Dedicated to

my wife
and
my parents
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures/Illustrations</td>
<td>v</td>
</tr>
<tr>
<td>List of Symbols/Abbreviations</td>
<td>vii</td>
</tr>
<tr>
<td>List of Publications</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>x</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Agent programming: Gaps in the literature</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Business process execution: Gaps in the literature</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Research questions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Structure of this dissertation</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Intelligent Agents</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Rational Agents</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 BDI-Agent Architecture</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3 Abstract BDI-Agent Interpreter</td>
<td>12</td>
</tr>
<tr>
<td>2.1.4 Plan and Plan Library</td>
<td>14</td>
</tr>
<tr>
<td>2.1.5 Planning</td>
<td>15</td>
</tr>
<tr>
<td>2.1.6 Current Implementations of BDI Systems</td>
<td>15</td>
</tr>
<tr>
<td>2.1.7 Intention Conflicts</td>
<td>16</td>
</tr>
<tr>
<td>2.1.8 Verification and Validation of BDI-Agent Systems</td>
<td>17</td>
</tr>
<tr>
<td>2.1.9 Model Checking Agent Systems</td>
<td>18</td>
</tr>
<tr>
<td>2.1.10 Conclusion</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Business Process Management</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 Business Process Modelling Notation</td>
<td>23</td>
</tr>
<tr>
<td>2.2.2 Process Semantics</td>
<td>24</td>
</tr>
<tr>
<td>2.2.3 Verification and Validation of Processes</td>
<td>25</td>
</tr>
<tr>
<td>2.2.4 Process Flexibility</td>
<td>26</td>
</tr>
<tr>
<td>2.3 Game Tree Search</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 Minimax Tree Search</td>
<td>28</td>
</tr>
<tr>
<td>2.3.2 Heuristic Function in Minimax Tree Search</td>
<td>29</td>
</tr>
</tbody>
</table>
# Table of Contents

## 1 BDI Agent Semantics and Its Application

### 3 Semantic Annotation of BDI Agent Programs

#### 3.1 Introduction

#### 3.2 Semantics of BDI-Agent Plans

##### 3.2.1 Semantics of Agent Execution

##### 3.2.2 Semantics of Plan Selection

#### 3.3 Soundness and Completeness of Plan Library

##### 3.3.1 Plan Internal Analysis

##### 3.3.2 Inter-plan Analysis

##### 3.3.3 Soundness and Completeness

#### 3.4 Compliance Analysis

#### 3.5 Experimental Evaluation

#### 3.6 Related Works

#### 3.7 Conclusion

## 4 Enhancing Agent Execution with Semantic Annotations

### 4.1 Introduction

### 4.2 Robust Agent Execution

### 4.3 Experimental Evaluation

#### 4.3.1 Agent Programs

#### 4.3.2 Environment Behavior Model

#### 4.3.3 Minimax Tree Search

#### 4.3.4 Monte Carlo Tree Search

#### 4.3.5 Evaluation Design

#### 4.3.6 Results

### 4.4 Related Works

### 4.5 Conclusion

## 5 Semantic Merging of BDI Agent Programs

### 5.1 Introduction

### 5.2 Illustrative example

### 5.3 Semantic effects

### 5.4 Semantic merging

### 5.5 Conclusions
## TABLE OF CONTENTS

### II Robust Process Execution in Adversarial Environments

103

6 Semantic Conformance and Compensation Computation 104

6.1 Introduction ............................................. 105
6.2 Semantic Process Monitoring ............................ 107
6.3 Semantic Compensation ................................. 111
6.4 Implementation and Evaluation ......................... 113
6.5 Related Works ............................................ 118
6.6 Conclusion ............................................... 119

7 Robust Process Adaptation using Game Tree Search 120

7.1 Introduction ............................................. 120
7.2 Processes Annotated with Post-conditions .............. 124
7.3 The Robust Process Enactment Problem ................. 127
7.4 Evaluation ............................................... 135
7.5 Related Work ............................................. 140
7.6 Conclusion ............................................... 141

8 Computing process compensation in complex, dynamic and potentially adversarial domains 142

8.1 Introduction ............................................. 143
8.2 Preliminaries ............................................. 144
8.3 A Game Against the World .............................. 145
8.4 Process Engine with Adversarial Compensation ....... 147
8.5 Experimental Evaluation ............................... 150
8.5.1 Environmental Behavior Model ....................... 151
8.5.2 Predicting Process Outcome ......................... 153
8.5.3 Compensation (Runtime Redesign) .................... 157
8.6 Related Works ............................................. 160
8.7 Conclusion ............................................... 162

9 Conclusion 164

9.1 Overview of the results presented .................... 164
9.1.1 Contributions to the body of research results ....... 164
9.1.2 Contributions to the practitioner community ........ 167
9.2 Limitations of this work and directions for future research .................... 167

A Process Models 170

References 202
List of Tables

6.1 Evaluated Process Models .................................................. 117
6.2 Best Evaluation Result ..................................................... 118
7.1 Summary of Result — Simple Process Set ............................. 138
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Agents interact with environments through sensors and actuators</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Illustration of a BDI agent</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>One Iteration of Generic MCTS</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Simple Plan Library</td>
<td>62</td>
</tr>
<tr>
<td>3.2</td>
<td>PROMELA model of pt</td>
<td>65</td>
</tr>
<tr>
<td>3.3</td>
<td>Time Spend in Constructing Traces for each Plan Library</td>
<td>69</td>
</tr>
<tr>
<td>3.4</td>
<td>Average Time Spend for a Constructing New State Description</td>
<td>70</td>
</tr>
<tr>
<td>4.1</td>
<td>Success Rate (achieving goal) in Simple Plan Libraries</td>
<td>84</td>
</tr>
<tr>
<td>4.2</td>
<td>Simulation Time in Simple Plan Libraries</td>
<td>85</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulation Time by Plan Libraries in Simple Plan Libraries</td>
<td>86</td>
</tr>
<tr>
<td>4.4</td>
<td>Success Rate (achieving goal) in Complex Plan Libraries</td>
<td>87</td>
</tr>
<tr>
<td>4.5</td>
<td>Simulation Time in Complex Plan Libraries</td>
<td>88</td>
</tr>
<tr>
<td>4.6</td>
<td>Simulation Time by Agent Systems in Complex Plan Libraries</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of classical, text-based merging (unnecessary conflicts)</td>
<td>93</td>
</tr>
<tr>
<td>5.2</td>
<td>Goal plan trees annotated with semantic effects for programs in Figure 5.1</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>A merged goal-plan tree and program for the example in Figure 5.1</td>
<td>100</td>
</tr>
<tr>
<td>7.1</td>
<td>Example: Game Tree Search in Car Service</td>
<td>129</td>
</tr>
<tr>
<td>7.2</td>
<td>Success Rate — Complex Processes</td>
<td>140</td>
</tr>
<tr>
<td>7.3</td>
<td>Average Decision Time (Seconds) — Complex Processes</td>
<td>140</td>
</tr>
<tr>
<td>8.1</td>
<td>changes of MSE over 2000 continuously simulated process instances</td>
<td>153</td>
</tr>
<tr>
<td>8.2</td>
<td>Predictions Errors Grouped by Processes</td>
<td>154</td>
</tr>
<tr>
<td>8.3</td>
<td>Predicted Process Outcome and the Average Success Rate</td>
<td>155</td>
</tr>
<tr>
<td>8.4</td>
<td>Predictions Errors Averaged by Instances</td>
<td>156</td>
</tr>
<tr>
<td>8.5</td>
<td>Percentage of Success without and with Compensations</td>
<td>158</td>
</tr>
<tr>
<td>8.6</td>
<td>Percentage of Success without and with Compensations (MCTS)</td>
<td>160</td>
</tr>
<tr>
<td>8.7</td>
<td>Percentage of Success without and with Compensations (Minimax Tree Search)</td>
<td>161</td>
</tr>
</tbody>
</table>
List of Symbols/Abbreviations

α-β pruning  Alpha-beta pruning (page 30)

Cn() a function that produces the deductive closure of an input logic formula.

⊖ a selection operator (binary operation) that returns a state description that is the union of the state description (left) and the context (right) if the union is consistent (see Definition 3.2).

α an action that the agent is able to perform

b BDI agent’s belief base

A capability library

c the context of a plan in the BDI agent’s plan library

e an effect of an action if the action is preformed successfully

E a set of (alternative) effects

μ an event of agent/business-process execution

g a goal (of a BDI agent, process model, or process instance) that is commonly represented as set of sentences of a given language \( L \)

\( G \) a set of goals

\( I \) a set of impediments

KB the background knowledge base which contains a set of sentences in some formula language \( \mathcal{L} \)

\( \mathcal{L} \) is some formal language

p a plan in BDI agent’s plan library.

L the plan library of a BDI agent

P a set of plans
List of Symbols/Abbreviations

- **P**: a process model
- **S**: a set of sentences from a given language \( \mathcal{L} \)
- **s**: a set of sentences/assertions in a formal that describes the state of affairs of the environment/world
- **S**: denotes a set of state descriptions.
- **\( \tau \)**: a task (of a process model or an enterprise capability library)
- **pt**: represents a predicted trace. Each predicted contains a sequence of state descriptions, each of which describes a possible state of the execution environment in the agent run time. A predicted trace indicates a possible execution instance of the agent system
- **PT**: a set of traces \( pt \)
- **\( \oplus \)**: an effect update operator (binary operation) used to produce all the possible resulting states after performing some action (see Definition 3.1)
- **ps**: a predicted state

Abbreviations:
- **ASP**: Answer Set Programming (page 64)
- **ATL**: Alternating-time Temporal Logic (page 71)
- **BDI**: Belief-Desire-Intention (page 9)
- **BPM**: Business Process Management (page 21)
- **BPMN**: Business Process Modelling Notation (page 23)
- **CNF**: Conjunctive Normal Form (page 81)
- **CTL**: Computation Tree Logic (page 69)
- **JPF**: Java Path Finder (page 70)
- **LTL**: Linear Temporal Logic (page 61)
- **MCTS**: Monte Carlo Tree Search (page 30)
- **RMSE**: Root Mean Squared Error (page 152)
- **UCB**: Upper Confidence Bounds (page 36)
- **UCT**: Upper Confidence Bounds for Trees (page 37)
List of Publications


ABSTRACT

Behaviour adaptation is a critical desideratum in almost all kinds of computational machinery. The challenge is to address the limitations of pre-programmed behaviour, which cannot anticipate the potentially vast range of possible situations in which that behaviour would need to be deployed or the potential adversarial behaviour of other entities in the operating context. This dissertation seeks to address this challenge in the context of two currently popular classes of computational machinery: intelligent agents and business processes.

The fact that the focus of this thesis is on these two kinds of machinery should come as no surprise. Besides that fact that considerable investments have been made (and continue to be made) on these technologies, it is also generally recognized that intelligent agent systems and business process execution frameworks have deep connections/similarities. In both cases, the solutions presented leverage two important observations: (1) The annotation of agent programs and business process models with post-conditions enables us to define triggers for behaviour adaptation and compute modified behaviours. (2) Viewing the interaction between the computational machinery and the environment as adversarial game-playing enables us to leverage game tree search as a means of computing robust adaptations to (a worst-case assumption of) maximally adversarial behaviour on the part of the environment.

This thesis makes contributions to agent technology by providing a scheme for annotating BDI agent programs in AgentSpeak-like agent programming languages. It then shows how this scheme can be used to achieve a framework for behaviour adaptation using game tree search. It also shows how these post-condition annotations can be used to merge agent programs.

With business process execution frameworks, this dissertation offers a novel notion of semantic conformance (as distinct from the current conception of conformance, which is structural conformance) which allows us to answer the following question: Has the process produced the expected post-conditions after each step? This conception leads to triggers for process adaptation. Specifically, the design of a process instance needs to be re-considered if a process is found to be semantically non-conformant. The thesis then offers a mechanism for computing process adaptations (called compensations). It then offers a game-tree search formulation of the problem computing process compensations.

KEYWORDS: BDI Agents, Agent System Evaluation, BDI-Agent Robustness, BDI-Agent Enactment, Business Processes Monitoring, Business Process Robust Enactment
Acknowledgements

I would like to express my deepest appreciation to my PhD supervisor Professor Aditya Ghose for the support and guidance he has given me throughout my candidature. I am extremely lucky to have met Aditya in 2012 when I was considering whether to take the Honours degree. It is him who encouraged and inspired me in challenging my self to reach a higher achievement, and introduced me to the area of AI and BDI agents.

It is a pleasure to thank those who made my degree possible. I owe my gratitude to Aditya, Associate Professor Andrew Miller and Illawarra Shoalhaven Local Health District for arranging and providing the scholarship that has supported me for the last few years.

A special thanks to my parents who have supported my education all my life, for the enormous amount of sacrifices they have made to allow me to achieve highly in my education.

Thank you, Jing, for being by my side.

It is an honour for me to be a part of the Decision Systems Lab. To who has been, is being and will be part of the lab, thank you, for all the advice on my study and life in the past years, for all the laughs and tears we shared, for all the game-nights, late-night writings, and paper-reading parties. It has been fun working with you.

Thanks to you all!
Chapter 1

Introduction

Behaviour adaptation is a critical desideratum in almost all kinds of computational machinery. The challenge is to address the limitations of pre-programmed behaviour, which cannot anticipate the potentially vast range of possible situations in which that behaviour would need to be deployed or the potential adversarial behaviour of other entities in the operating context. In early 2018, a fatal accident caused Uber to permanently shut down its self-driving car program in Arizona, when its self-driving car struck and killed a pedestrian. The car failed to recognize the pedestrian in time (even though it had operated normally on public road before the accident) due to many reasons, including the visibility of its camera at night, the speed at which it is able to process sensory information, and the ability of its object recognition system to correctly classify the objects on the road. More importantly, it was the failure of its pre-programmed behaviour. According to the report from the National Transportation Safety Board (NTSB) in the U.S.\(^1\), Uber disabled the car’s built-in emergency braking system to prevent conflict with the self-driving system, and a human operator in the vehicle is expected to brake in case of emergency. However, the scenario of the accident is unexpected, and was not anticipated. The car was traveling at 43 mph (about 70 km/h), 6 seconds before the impact when the self-driving system first registered the
pedestrian as an unknown object, then as a vehicle and then as a bicycle with varying expectations of its future travel path. Only at 1.3 seconds before the impact, the system determined that an emergency braking manoeuvre was necessary. The system log showed the operator intervened less than 1 second before the impact by engaging the steering wheel. The speed of the impact was 39 mph (over 60 km/h) and the operator began braking less than a second after the impact. This is one compelling example of how pre-programmed behaviours might not easily handle all contingencies.

Another example of a failure of pre-programmed behaviour in unanticipated scenarios and adversarial environments is when Microsoft unveiled its Twitter bot, Tay, as an experiment in conversational understanding in 2016. Tay was programmed to learn to engage people through “casual and playful conversation”. However, the learning module was not capable of correctly distinguishing between acceptable and unacceptable conversational behaviour. It was programmed to learn from all conversations with Twitter users whenever the conversation activation signal (“repeat after me”) was identified. In about 15 hours, Tay began to post offensive tweets and Microsoft had to shut it down only 16 hours after launch.

Examples such as these abound in any application of pre-programmed behaviour, and there are more detailed examples in Chapter 4, Chapter 6, Chapter 7 and Chapter 8 which are more relevant to the topics and the application domains of these chapters. This dissertation seeks to address this challenge in the context of two currently popular classes of computational machinery: intelligent agents and business processes.

The fact that the focus of this thesis is on these two kinds of machinery should come as no surprise. Besides that fact that considerable investments have been made (and continue to be made) on these technologies, it is also generally recognized
that intelligent agent systems and business process execution frameworks have deep connections/similarities. In both cases, the solutions presented leverage two important observations: (1) The annotation of agent programs and business process models with post-conditions enables us to define triggers for behaviour adaptation and compute modified behaviours. (2) Viewing the interaction between the computational machinery and the environment as adversarial game-playing enables us to leverage game tree search as a means of computing robust adaptations to (a worst-case assumption of) maximally adversarial behaviour on the part of the environment.

1.1 Agent programming: Gaps in the literature

Agent technology, and agent programming, is in the midst of a comeback of sorts, with a spike in interest driven by the growth in autonomous unmanned vehicles, drones, autonomous submersibles etc. There is also greater recognition of the role of agent technology in providing decision support for command-and-control applications.

The dominant BDI framework for intelligent agents [101] and the associated AgentSpeak [100] family of agent programming languages, while providing a sophisticated framework for agent systems development, continue to suffer from some drawbacks. One of these is the absence of post-conditions. To understand the behaviour of an agent program, or to understand its effects or the way it changes the state of its environment, the best that current solutions offer is a sequence of actions. This has several undesirable consequences. It rules out the prospect of using planning-like techniques in conjunction with the reactive reasoning that BDI agent systems routinely support. It makes goal-oriented reasoning difficult (a goal is typically specified as a desired post-condition, but in the absence of any reference to post-conditions in the agent program, it is difficult to determine whether the execution of the agent will indeed lead to the achievement of the relevant goals). It makes the analysis of whether agent execution leads to the desired
intermediate conditions (i.e., not necessarily just the final goals) difficult. This latter form of analysis is crucial in a number of important domains, such as medicine. It also makes compliance checking difficult (i.e., answering the question: will any compliance rules be violated in the course of agent execution?). Ultimately, we will also show that post-conditions are critical in generating robust behaviour adaptation in adversarial environments.

The adaptation of agents to changes in the environment has been a key underlying theme in the design of agent programming languages and environments. However, a critical form of reasoning, which assumes adversarial behaviour on the part of other entities/agents in the environment, and which involves look-ahead search (of the kind popularly referred to as game-tree search) has been conspicuously absent in the literature on agent programming.

The ability to annotate agent programs with post-conditions comes with other positive consequences, one of which is an elegant approach to merging agent programs. This too is reported in this thesis.

### 1.2 Business process execution: Gaps in the literature

Business process management frameworks and systems have enjoyed significant industry uptake in the past few decades. Concomitantly, there has been a significant amount of research activities in this space. There is a general consensus that the design, execution and management of flexible and adaptive business processes is one of the major challenges facing the research community in this space.

Business process execution monitoring is critical in managing process execution. Current conceptions of process monitoring are somewhat limited. The dominant approach to process monitoring is process conformance checking. Conformance checking helps us ensure that the sequence of tasks being executed is in fact a valid execution
sequence mandated by the process model/design. It does not, however, ensure that
the process *does the right thing*. Current conceptions conflate correct execution with
executing a correct task sequence. As shown in Chapter 6, execution errors within
tasks are often not visible to the conformance checking machinery, leading to situations
where the process fails to do the right thing even though the conformance checking
machinery suggests that it has. There has been almost no work done on the question
of *semantic conformance*, where the intent is to check whether a process has delivered
on the expected (or intended) effects after every task. There is a substantial literature
on ECA (Event-Condition-Action) rules (e.g., [52]) and GTA (guard-activity-triggers)
approach (e.g., [92]) but nothing in this literature addresses the problem of semantic
conformance.

When a process instance is found to be semantically non-conformant, one needs
to deploy a “fix” or a *compensation*. A compensation can be viewed as an alternative
completion of the process instance (relative to what was mandated by the process
design). There is little by way of proposals in the literature on the computation of
process compensations, something that this dissertation seeks to redress.

Business processes often execute in dynamic uncertain environments (in many
business settings, process instances rarely end in the manner anticipated by the process
design). A conservative approach, involving worst-case analysis, would assume that
the environment is populated by actors/agents that are *maximally adversarial* to the
current process instance (i.e., they seek to place as many *impediments* as possible to the
successful execution of the process instance). Robust adaptation would require worst-
case analysis of this kind, but there are no proposals in the literature that support
this form of reasoning. This dissertation addresses this gap by formulating the robust
process adaptation problem as an adversarial, two-person, turn-taking game of perfect
information, consisting of the process execution actor pitted against the (potentially
maximally adversarial) environment actor.

1.3 Research questions

The key research questions addressed in this dissertation are as follows:

- **RQ-1:** Can agent programs be annotated in a manner that permits the user to compute the post-conditions achieved by an agent at any point in its execution via design-time analysis?

- **RQ-2:** Can agent programs use adversarial game-tree search to compute optimal behaviour choices in dynamic uncertain environments?

- **RQ-3:** Can a scheme for annotating agent programs with post-conditions provide the basis for a principled approach to merging agent programs?

- **RQ-4:** Can business process execution be monitored by leveraging intended intermediate effects?

- **RQ-5:** Can compensations for processes that deviate by failing to deliver the intended intermediate effects be computed efficiently?

- **RQ-6:** Can adversarial game-tree search be used to compute robust process adaptations in dynamic uncertain environments?

1.4 Contributions

The key contributions of this thesis are as follows:

- This thesis presents a novel scheme for annotating BDI agent programs with post-conditions in a manner that permits the user to analyze the effects achieved by
1.4. Contributions

agent execution purely through design-time analysis. This enables compliance analysis, goal analysis and the analysis of intermediate effects, amongst others. Ultimately, post-conditions play a vital role in generating robust adaptations, both in the case of agents and business processes.

- This thesis presents an approach to robust behaviour adaptation in BDI agents by leveraging post-condition annotations and adversarial game-tree search.

- The thesis also shows how post-condition annotation of BDI agent programs can lead to a principled scheme for merging such programs.

- This thesis offers important advances in the way in which business process execution is monitored by defining a notion of semantic conformance (as opposed to the traditional notion of conformance — which we refer to as structural conformance).

- The dissertation shows that it is possible to compute compensations (i.e., alternative process completions) for process instances that are found to be semantically non-conformant. The guiding principle is to compute compensations that deviate minimally from the structure mandated by the process design, that restore semantic conformance as early as possible while still ensuring that process goals are satisfied.

- This dissertation offers a scheme that used simple models of environment behaviour, coupled with game-tree search techniques to compute robust process compensations in dynamic, uncertain environments.
1.5 Structure of this dissertation

This dissertation is structured as follows. Chapter 2 provides the necessary background material the reader will need to appreciate the original research contributions in the following chapters. Chapter 3 presents the scheme for annotating BDI agent programs with post-conditions. Chapter 4 shows how game-tree search can be used to compute robust behaviour adaptations in BDI agent programs. Chapter 5 shows how the post-condition annotation scheme can be used to support a method for merging BDI agent programs. Chapter 6 defines the notion of semantic conformance and shows that process compensations can be efficiently computed. Chapter 7 provides a preliminary approach to using game-tree search for process adaptation while Chapter 8 elucidates these techniques in greater detail.
Chapter 2

Background

This chapter reviews existing literature, introduces the concept of agents, Belief-Desire-Intention model, business process management and game-tree search, all of which are necessary in understanding this thesis. Despite the diversity of the topics this thesis covers, the general concepts and precipices are very similar. Section 2.1 provides preliminary knowledge on intelligent agents and Belief-Desire-Intention model. Section 2.2 introduces the basics of business process management. The last section, Section 2.3, covers the fundamentals of game-tree search and two of its most popular family of algorithms.

2.1 Intelligent Agents

Intelligent agents are treated as a potential technology that is able to replace the current methods for design and implementation of complex software systems [94]. The Belief-Desire-Intention (BDI) agent technology is one of the most promising intelligent agent architectures, which has been implemented and applied in wide range of domains. In the following sections, we first introduce rational agents and agent-oriented programming, then discuss a specific rational agent model, BDI-agent architecture.
This section reviews the current state-of-the-art in developing BDI-agent systems. The review uncovers the immaturity of verification and validation techniques.

### 2.1.1 Rational Agents

The concept of modelling systems around agents is not only in computer science. A rational agent is an agent that acts rationally with respect to its preferences towards the optimal expected outcome. It is used in economics, game theory, decision theory and artificial intelligence. In these disciplines, rational agents could be people, firms, machines, software systems, etc. In artificial intelligence, the term intelligent agent normally shares the same meaning with rational agents. From now on, we will only use rational agents and intelligent agents to refer to software agents in software systems.

Agent-oriented programming is the idea of modelling software system around the concept of agents [119], in which each component, or agent, perceives environments, where it is through sensors and acts on the environment with actuators on behalf of another entity [108].

In [108], Russell and Norvig defined the rational agent as the agent that always selects actions that are expected to optimize its utility measure based on its perceptions and the built-in knowledge it has [108]. This definition focuses on the result of each
2.1. Intelligent Agents

agent’s decision making. On the other hand, Rao and Georgeff [103] have defined a rational agent based on the decision-making process, that is a rational agent “has certain mental attitudes” that affect its decision-making behaviours [103].

The characteristics of software systems that can be recognized as agents include [140]:

**Autonomous** agents should be able to control their actions and decision-making process without the direct intervention of humans and other agents.

**Social** agents should interact with other entities such as other agents, other software systems, humans, etc. via some communication languages [58].

**Reactive** agents should be able to respond to events occurring in the environment.

**Pro-active** agents should act to realize long-term goals.

**Situated** agents should interface with the environment where it is located.

These characteristics are commonly agreed on, however, there are other descriptions of characteristics of being agents in [50, 51, 80], such as learning, mobile, flexible, character, context-sensitive, procedural, parallel, interactive, dynamic, uncertainty, etc. In general, the notion of an agent is a point of view used to analyse and model complex systems. There are no absolute characteristics that classify agent and non-agent [105]. The characteristics of agents are highly dependent on the systems where the agent is.

2.1.2 BDI-Agent Architecture

The *Belief-Desire-Intention* (BDI) model is an agent architecture of modelling rational agents, which is inspired by the Belief-Desire-Intention model of human practical reasoning by Michael Bratman (but they are not the same model). The concept of modelling rational agents under the BDI architecture was first raised by Rao and
2.1. Intelligent Agents

Georgeff [101, 102]. In their work, a BDI agent is described as having the mental attitudes of belief, desires (goals) and intentions.

**Belief** is an agent’s information about itself and the external environment. We use the term “belief”, rather than “knowledge”, because the agents’ information of itself and the environment may or may not be true all the time (e.g. synchronized update of information) but it is believed by the agent at this present point of time.

**Desire** (or goal) is what an agent wants to achieve.

**Intention** is the mean of achieving a desire/goal after the agent has committed to achieving it. In the classic BDI-agent system, such as AgentSpeak(L) [100] programming language family, an intention normally is an instantiated plan, or plan and its backup plans such as in CANA [111].

2.1.3 Abstract BDI-Agent Interpreter

The abstract BDI-Agent Interpreter, described by Rao and Georgeff [102], works in reasoning cycles shown in Algorithm 2.1 (also illustrated in Figure 2.2), where $B$, $G$, $I$ denotes a set of beliefs, goals and intentions respectively. The input to BDI-agent

**Algorithm 2.1** Abstract BDI Agent Interpreter

```
Initialize_state()
loop
    options ← option_generator(event_queue, B, G, I)
    selected_options ← deliberate(options, B, G, I)
    update_intentions(selected_options, I)
    execute(I)
    get_new_external_events()
    drop_successful_attitudes(B, G, I)
    drop_impossible_attitudes(B, G, I)
end loop
```
systems are events, which will be put into an event queue. The BDI-agent interpreter will generate all possible options based on what is currently in the event queue, then select the most feasible options as its new intentions. After the agent executes its intentions (one or many), it will check if there are any updates from the environment. At the end of the cycle, the agent will drop goals that are successfully achieved, and goals and their correlated intentions that looks impossible to achieve.

This abstract interpreter is the basis of the modern BDI-agent systems, but they may treat these mental attributes differently. For example, in some BDI-agent languages and systems, the event also includes internal events that are created by the agent itself, and the options are represented as *plans*, such as in AgentSpeak(L), Jason [17, 18].
In CAN\textsuperscript{A} \cite{111}, JACK \cite{25}, etc. In CAN\textsuperscript{A} \cite{111}, the agent, instead of dropping impossible attitudes, first searches in the backup plans to see if it is possible to realize the same goal by different means. There are systems that use events to represent goals, like in AgentSpeak(L), meanwhile, others use declarative goals such as GOAL \cite{19}.

In the most popular programming language of BDI-agent system, such as 3APL \cite{37,36} and AgentSpeak(L) \cite{100}, the core of an agent program is a set of initial beliefs, a knowledge/rule base, and a plan library. The initial beliefs represent the default beliefs of the agent about the environment where it is, which could be updated at runtime. The knowledge base (sometimes called rules) describes some facts and rules that the agent should follow when reasoning about updating its beliefs, reasoning about how to achieve goals, and other internal reasoning tasks, which can be assumed to be static during the execution. The plan library gives the agent a set of options (plans) to fulfil a desire (goal), from which the agent chooses a plan to achieve a given goal.

### 2.1.4 Plan and Plan Library

In a BDI-agent system, a Plan is introduced as a procedure to satisfy a certain goal. In AgentSpeak(L) and most of its extensions, a plan consists of head and body, where the head is a triggering event and the context describing when the plan is feasible, and body is a sequence of actions and/or sub-goal to achieve. The example from the first formalization of AgentSpeak(L) below \cite{100} illustrates a plan in a cleaning robot.

```latex
+\text{location}(\text{waste}, X): \text{location}(\text{robot}, X) & \text{location}(\text{bin}, Y)
\leftarrow \text{pick}(\text{waste});
\quad !\text{location}(\text{robot}, Y);
\quad \text{drop}(\text{waste}).
```

When cleaning robot received the event \textit{location}(\textit{waste}, \textit{X}), which means there are wastes at location \textit{X}, the plan above will be selected only if the robot believes that
it is currently at location $X$ and the bin at location $Y$. The plan body, starting after "$< -$" instructs the robot to pick up the wastes, then move itself to location $Y$, ("!" means "to achieve", it will raise an internal event $\text{location}(\text{robot}, Y)$, then the agent will find a suitable plan for the new event), and drop the waste at the end.

A goal may have more than one plan to handle it, which are called the relevant plans for the goal. The plan library is a set of predefined plans as described above.

### 2.1.5 Planning

On a different note, there are also planning systems that are capable of generating plans in design time or runtime of BDI-agent systems, such as STRIPS [49], value-iteration [7], policy-iteration [66], and hierarchical task networks (HTN) [44, 43]. We are not going to review the detail of planning system as this work is focusing on the static verification and validation of BDI-agent system based on a pre-defined plan library, although some of the methodology developed in this thesis may be able to be extended to support planning systems, which is also beyond the scope of this thesis.

### 2.1.6 Current Implementations of BDI Systems

There are many programming languages and platform developed over last two decades for implementing BDI-agent systems. Some of these languages include PRS (Procedural Reasoning System) [67], AgentSpeak(L) [100], Jack [25], dMARS (Distributed Multi-agent Reasoning System) [138, 11], 3APL [113], AgentSpeak(XL) [11], MABLE [139], SPARK (SRI Procedural Agent Realization Kit) [85], GOAL [19], AgentSpeak(I) [134], Jason [17, 18], 2APL [35], CASO [32, 33], CAN [111], BAOP [34].

PRS is the first system that was implemented based on the BDI model. In some implementations of agent development systems, explicitly represented goals are absent, instead, the goals are represented as events, such as in PRS, dMARS, AgentSpeak(L),
and Jason. In languages featuring declarative goals, such as GOAL, a goal specifies a state of the environment for an agent to achieve. Take the previous plan for a cleaning robot as an example, its corresponding go could be described as follow,

```prolog
goals{
    location(waste, Y).
}
```

which simply means to move waste to \( Y \) where the bin is. Some languages, like MABLE, are conventional imperative programming languages extended with additional structures to support BDI-agent architecture.

### 2.1.7 Intention Conflicts

When developing BDI agents, one plan is designed for a specific goal, with the assumption that the agent will follow the “step by step” plan to achieve a goal. In most BDI-agent implementations, such as 3APL and AgentSpeak(L) [100], the agent is normally allowed to have multiple desires, so that the agent is able to achieve more than one goal concurrently. Therefore, the agent could have more than one intentions at a random point in time. The execution of each step in an individual intention is always in the same sequence as planned (programmed). Therefore, executing an intention among a set of concurrent intentions should not be different from executing the intention alone. However, since an agent is allowed to concurrently execute intentions, a new issue occurs, that is, actions from different intentions may interfere each other. In imperative programming, every step of the program in execution is well defined, but in BDI-agent implementations, there are no specifications whatsoever in the program itself how any two plans should be concurrently executed, which has been acknowledged as one cause of agents’ failing to realize an objective [125].

There are discussions in recovering when an intention fails [111]. However, there
is only a few addressing the conflict among intentions with concurrent execution [2, 46, 47, 115]. Thangarajah [125] discussed this issue in two catalogues, the resource conflicts and interference. The resource conflict is the situation where, for example, two concurrent intentions require 100 units of energy each to realise their goals, and there are only 150 units available at the time. There is interference, for example, when an agent uses a plan to cook dinner by going to store, buying some items, coming home and preparing the meal. A rational agent should not allow a plan that took the agent elsewhere after it arrives the store and before it finishes buying items [127].

2.1.8 Verification and Validation of BDI-Agent Systems

In recent years, BDI-agent systems have been studied and applied in a wide variety of domains, such as simulation [116], business process management [22], transportation [23], logistics [97], terrorist detection, crowd modelling [116], and even safety-critical applications such as autonomous spacecraft control [59, 88]. To prove that these software systems are able to provide the expected result, verification and validation is the pivotal procedure during the development process. However, there has not been much research addressing it. The current techniques and tools for verifying agent-based system are still in their infancy, and in most of the practical verification approaches, the verifications are done only on code [12].

Compared with procedural and object-oriented programming, establishing test cases are more difficult in testing agent-based systems [93] as the actual steps of execution is opaque, which could depend on a complex combination of architectures, plans, and the changes in the environment during execution [83]. Therefore, the focus of agent system verification and validation shifted from testing to formal methods, especially model checking [1] as model checking is considerably easier than theorem proving, another formal method, and it can provide counterexamples that theorem
proving cannot [12].

Furthermore, there are other approaches that aim to provide semi-automatic bug detection, such as in [96, 99, 98], the techniques detect bugs by comparing execution traces with pre-specified interaction protocol, which are beyond our scope as it is bug detection rather than formal verification.

2.1.9 Model Checking Agent Systems

Rao and Georgeff discussed the very first model-based theoretic approach to verify BDI-agent system [103]. In the paper, they defined three propositional BDI logic languages, $\text{CTL}_{\text{BDI}}$, CCTL$\text{BDI}$ (Committed $\text{CTL}_{\text{BDI}}$), and $\text{CTL}^*_{\text{BDI}}$, based on the branching temporal logics $\text{CTL}^2$, Fair $\text{CTL}$ and $\text{CTL}^*$, respectively, to increase their expressive power under BDI system context. There are two types of well-formed formulas in the three languages. A state formula is a description of a particular world at a particular time. A path formula is true in a particular world along with a particular path. After discussing the expressive power and model checking using the three different languages, Rao and Georgeff summarized that the complexity of model checking for $\text{CTL}_{\text{BDI}}$ has the same or greater complexity than linear temporal logic (LTL) model checking, and has the potential of verifying practical agent-oriented systems using expressive multi-modal, branching-time logics.

In recent years, research has focused on extending existing model checking systems and technologies to agent and multi-agent systems, such as [8, 13, 16, 14, 139]. In 2003, Bordini, Fisher, Pardavila and Wooldridge [14] introduced AgentSpeak(F), a variation of AgentSpeak(L), that can be translated directly to Promela\(^3\), a model specification

\(^1\)Model checking refers to automatically and exhaustively checking if a given technique hasn’t been applied to industrial scale cases [12].

\(^2\)CTL: Computation Tree Logic

\(^3\)Promela: **Process Meta Language**
language for SPIN\footnote{SPIN: Simple Promela Interpreter} model-checking system. AgentSpeak(F) is still immature, and there is more work to be done to improve the efficiency and scalability. Similarly, the multi-agent programming language MABLE, proposed in 2002, can also be translated to Promela and verified by SPIN\cite{139}. One year later, Bordini, Fisher, Visser, and Wooldridge\cite{15} stated an alternative approach, in which AgentSpeak(L) is translated to Java program and then is verified using JPS (Java Path Finder), an existing model checking tools developed by NASA.

Meanwhile, in 2003, Benerecetti and Cimatti\cite{8} presented another approach of validating multi-agent systems, which is based on describing the system using Multi-Language FSM, the expected properties of a system specified in modal temporal logic, and a decision procedure based on model checking techniques. The algorithm described has been implemented in NuMAS system (NuSMV for Multi-Agent Systems) that is built on top of NuSMV symbolic model checker\cite{8}.

Based on the common semantic basis for BDI languages\cite{38}, and their previous works on verifying BDI-agent system using JPF, Bordini, Dennis, Farwer, and Fisher discussed the automated verification using this common semantic basis\cite{13}. The previous works on model checking agent-based system are mostly bounded to one particular programming language. Bordini et al.\cite{13} introduced the Agent Infrastructure Layer (AIL) that encloses the main concepts of a wide range of agent programming languages. The idea is that all that the agent-based systems developed using languages that are supported by AIL can be translated, and run in an AIL-based interpreter using AIL data structures and Java classes provided by the AIL toolkit. In addition to AIL, there is an MCAPL interface that provides the support for model checking multi-agent systems against specifications written in Property Specification Language (PSL). The reason for such a translation is to provide a unified framework to verify agent-based systems developed in different languages. The verification of translated
source code is done using AJPF, an extended version of JPF. AJPF has AIL classes embedded in and extended to support temporal logic model checking. Some simple case studies on simple agent systems show AJPF is more than 150 times faster than JPF [13]. However, agent systems studied were too simple to show that the approach is applicable to large-scale practical systems.

On the other hand, there are new model-checking methodologies and formal languages designed exclusively for agent and multi-agent systems, such as MCMAS [79] that supports CTL, ATL (Alternating-time Temporal logic [3]), and mv μK-calculus. ATL is designed to model multi-player games. The complexity of model checking with ATL has been shown to be the same complexity as the theorem-proving problem, which is EXPTIME-complete for “full” ATL, and PSPACE-complete for a fragment of ATL. mv μK-calculus [74] is an expressive logic used to specify knowledge and time in a multi-agent system. Hoek and Wooldridge [118] showed CKLₙ (a temporal logic of knowledge) based model checking on distributed agent systems, and showed that it is possible to reduce CKLₙ model checking to linear temporal logic model checking.

The current technologies of model checking agent and multi-agent systems are immature [12]. In some BDI-agent programming languages, such as AgentSpeak(L) and most of its extensions, the agent does not understand the meaning of its actions. Agents in such systems will select actions based on its belief and knowledge. Therefore, when statically verifying a system (without executing), the verification can only be done at the level of checking the accepted order of actions. Moreover, all current model checking methods are not able to prove a foundational propriety in BDI-agent systems, that is the consistency within the plan library. The possible inconsistency that could occur within a plan library may include,

1. effect of a plan does not realize its designed goal;

2. not all possible effects of a plan realize its designed goal;
3. there are goals that are not realized by any plan;

4. under a certain context, there is no plan that is feasible to realize a goal.

Last but not least, efficiency and scalability are limitations of current model checking methods.

2.1.10 Conclusion

We have introduced the BDI-agent architecture, the current implementations of BDI-agent systems and their applications. Then we have briefly discussed the needs of proving a BDI-agent system can provide the expected outcome and the difficulty of doing such verification. We reviewed the current state-of-the-art in BDI-agent system verification and validation, the uncovered the missing links in current state-of-the-art.

2.2 Business Process Management

A business process is commonly referred as a description of how the things are done in the specific business domain normally described as a chain of event, activities, and decisions that have to be performed [42]. The common notations that are used to describe and visualize a business process including plain text document, formal language, flowchart, petri-net [129] [128], and BPMN [95].

The meaning of business process management (BPM) can be different in research and industry and sometimes even differ in application domains. In the Cambridge Dictionary, BPM is defined as “the development and control of processes used in a company, department, project, etc. to make sure they are effective”, which only describe the early developments of BPM. Later, Smith and Fingar define shifted the focus of the BPM to the coordination and collaborations of activities of processes and value and outcomes that the processes deliver [123]. IBM sees the BPM as a
2.2. Business Process Management

software and services in their definition that highlights the discovery, documentation, automation and continuous improvement that increase the efficiency and reduce the cost[1]. van der Aalst, et al. sees the BPM as “supporting business processes using methods, techniques and software to design enact control, and analyse operational processes involving humans, organizations, applications documents and other sources of information” [132, 130]. The most common definition that covers the most activities that are possibly a part of BPM is that BPM is the term used to describe activities involved in modelling, executing, controlling, measuring, improving, optimizing, and automating business processes [124].

The idea of modelling processes is not new, but has long been used to incorporate views and knowledge. In BPM, the process modelling exercise is normally done in the way that filters out any irrelevant details of the business environment so that the core of the process can be exposed without the complication of everything within the organizations. The process model, then, is an abstraction of the collection of business instances that aims to deliver the same business outcome, with a limited number of entities that are directly related, affect and contribute to the process. Some language or notations are designed to describe the process at this level. van der Aalst has proposed to use petri-net for modelling business processes [129, 128] to use the formal semantics and the graph nature of the petri-net to expressively describe process ultimately leading to formally analyse, verify, and validate a process or a collection of processes. On the other hand, declarative languages such as in [86, 75] have been used to describe the process model too, where, instead of using graph to visualize the process model, a set of predefined notations are used to describe the temporal relation of tasks, decisions and events of the process model as well as some constraints that future limits the possible process instances. Commonly, with such declarative approach

2.2. Business Process Management

are more general and fixable, where a process description can result in multiple process models visualized in petri-net or BPMN, and with extra information such as costs of tasks and extra constraints, the optimal models can then be generated from the declarative description for the given process execution domain. In this thesis, the more general, and standardised notation, business process modelling notation (BPMN), that is preferred by the industry is used as the default notation of process models.

2.2.1 Business Process Modelling Notation

The business process modelling notation creates graphical representations of business processes using types of standard graphical elements including activity objects, flow objects, connectivity objects, grouping objects, event objects, annotations and artifacts [95].

Some event objects trigger either the start (start events) or the end (end events) of a process instance while some other events trigger or interrupt intermediate point of the process instance (timers, calendar events, messages received events, etc.) [95].

An activity or a task is a single atomic step of the process in executing the process [95]. The granularity of the meaning of the task can differ according to the business domain. For example, the task “prepare order” in a process handling online order may be further decomposed the task of finding the ordered item from the warehouse and retrieve the item from the warehouse. However, this level of details may or may not be useful and relevant to the BPM activities in the process modeller’s mind. In BPMN, such case can be handled by sub-processes notation where there are some activities that can be realized via another process [95], which effectively provide the tools for the modeller to consider and model more details into the process model while having the option to present and analyse the process on the more general level by hiding the sub-processes thus sees them as activities instead. This notation is more expressive and more flexible in the modelling compared against petri-net.
Gateway objects direct the process pathways according to the process instance and its execution context. For example, the XOR gate marks a decision that has to be made to choose one and only one path out-going from the gate. In the insurance claim handling process, such decision could be either accepting or rejecting the claim. In the visa handling process, such decision could decide the type of the visa that is available to the applicant. The OR gate is more general, where at least one out-going path can be selected to execute. The AND gate is commonly known as parallel execution where all the paths have to be completed but the order of the tasks across the different paths does not matter as long as the tasks are completed in the order of its own path [95].

Other notations used such as the artifacts that are created, and/or required by the process such as documents, forms, record, etc., can be added to the process model to provide more information. The notation of annotations can be used to provide more detailed description such as a description of the task. Last but not least, pools are used to represents the major participants of the process and lanes can be used to organise and categorise tasks within a pool according to function or role.

### 2.2.2 Process Semantics

Although BPMN provides comprehensive selection of notations to model business processes, its lack of task semantics has been noted by Hinge et al. [64], where they argue that BPMN only provide structural notation of the task flow but the meanings (the effects or the semantics, in the author’s words) of the executing these flows are not known. In [64], a formal definition of business process models has been provided to BPMN, where each task are described and annotated using their expected outcomes (effects) using a given formal language. Then along with the defined semantics of flow objects and gateways, the expected outcomes of every step of the process execution can be then calculated automatically to provide the semantic description of the process.
2.2.3 Verification and Validation of Processes

When the business process is modelled and documented, the next question to ask is whether the executed processes are confirmed with the process model, and detect if any part of the model has been violated, which could include the syntactical (or structural) violation and semantic violation.

Syntactical (or structural) violations are the non-conformances that lie in the order of the tasks, where the activities are carried out in the orders that differ from the process model. Cook et al. propose a process validation framework, in which the event stream from the process model are compared against the event stream from the execution (either in real-time or from the execution logs of the past process instances), then a string distance metrics are used to decide “how much” of the differences are there from the design (the process model) to the execution (the execution logs), where each event is treated as an element of the string [28]. A conformance checker developed by Rozinat and van der Aalst [107] as a part of the ProM framework checks a collection of event logs (consisting of events collected during process execution) of a given process model to determine whether the process execution behaviour reflects the designed behaviour.

On the other hand, a semantic violation is a case when the result of the process or the part of the process does not realize the expected outcome according to the process design. Semantic violations are defined differently in literature, in Chapter 6 of this thesis, we see the semantics of the processes as the expected outcomes (described in some formal language) of every step of the process execution. Meanwhile, a number of proposals for goal-oriented process management exist [53, 73] where only the outcomes of complete processes (goals) are taken into account as the process semantic, which focus on the ends (goals) and the means to the end (the process) are not important in the investigation of [53, 73].
2.2.4 Process Flexibility

Process flexibilities have long been recognized as an issue in the real world business process management [131, 21, 60, 69, 105]. Some of the existing literature on process flexibilities are addressed in design [62], or exception handling by design [69]. Whenever there is a violation, no matter if it is syntactical and/or semantic, it means things have gone wrong in the execution. Some recovery options have to be considered to prevent the process fails to deliver the expected business outcome and values. The act of looking for recovery solutions are commonly known as exception handling. Klein and Dellarocas [69] present a knowledge-based approach to detect such exceptions as well as handling exceptions in work-flow systems. The exceptions of the process and the types of the exceptions are pre-defined by the process designer, and the participant of an enacted process will be notified whenever there is an exception, so that the participants are able to modify the enacted process instances to resolve the exception according to their expert domain knowledge and allow the process to continue without any errors or failures [69].

Others address the flexibilities at the process execution, such as by taking into account of risks [27], by generating optimized enactment plans according to multiple optimization objectives [68], by following a checklist if the process is human-driven [6], or by allowing minimal deviation from a design during execution as proposed in Chapter 6. Generally speaking, to change or augment the process instances according to the execution context, we can either follow some predefined guidelines [62, 69], or we can automatically or manually create a new process model for the current context in runtime with some form of predefined objectives [68, 6].

Additionally, there are proposals that utilize the agent technologies to create more flexible process models, as the intelligent agent architectures are designed to deal with a flexible environment [21]. Automatic planning from agent-related research also
adapted for process planning [112], which argues that the manual planning (process modelling) can be replaced with planning. As a result, a more optimized and instance-based plan (process model) can be created on-the-fly during process execution.

2.3 Game Tree Search

A game tree is a directed graph where every node represents a state of the game and an edge represents the move can be made to change the game from a state (source of the directed edge) to the next state (the target of the directed edge). Game states are commonly classified into 3 types, initial states, terminal states, and intermediate states. The initial states are the possible starting points of the game where no move has been made by players yet. The termination or terminal states are the states when the game ends and no more legal moves available to players. In an adversarial game, where players compete against each other, the terminal states can be further classified to winning/losing states for any given player. The intermediate states are the states neither initial nor terminal states. In a deterministic, turn-taking game of perfect information, all the incoming edges of the level are associated to the legal moves of one player who makes moves on the corresponding turn, so that the levels of the tree describes the turn of the game of the players.

A complete game tree is a game tree that starts from an initial state, contains every possible move from every intermediate state, to all the possible termination states. Calculating a complete game tree that mapping all possible game state is expensive, where the number of nodes of the tree normally grows exponentially with the depth of the tree [90]. For example, with the simple game of tic-tac-toe, the complete tree has 255,168 nodes. The game of chess has an average branching factor of 35 (for every game state, a player could make 35 different moves on average), an average game would take about 50 moves each player, thus an average game tree of chess would have $35^{100}$
2.3. Game Tree Search

nodes [108]. It is usually infeasible to perform a complete search of a large game tree due to the limited resources and computing power of the current hardware even with pruning technique such as the alpha-beta algorithm [5, 70, 90]. Some form of heuristic function can be used to limit the search to a limited depth of the game tree, where the heuristic function evaluates and estimates the utilities of nodes at the given depth. It used to be agreed that increasing the depth limit of the tree search with heuristics would improve the search quality. However, research finds that in some game trees, increasing search depth will not improve the probability of the decision being correct but cause the decisions become increasingly random [89, 90, 91]. The arguments that in some games (known as pathological games), the deeper of the search, the heuristics of each decision will eventually become equal in their utility thus the decision will appear to be more random. This is not applicable to the game such as chess, but it makes the point that increasing depth of the search may not be beneficial for all games. Pearl argues that instead of using utilities as the measurement of how good of a state to the player, the probability of the game outcome at any given state can be instead used to improve the tree search in pathological games [136].

In this chapter, the two major algorithms, Minimax Tree Search and Monte-Carlo Tree Search, and some of their variants are reviewed to provide the foundations of the algorithms used in the later chapters.

2.3.1 Minimax Tree Search

Given two players with conflicting goals, and a utility function, we can call one player MAX (maximizing player) and another MIN (minimizing player) as one aims to maximize the utility and another to minimize. In another word, what is good for MAX will be bad for MIN, and vice versa. Given a game tree, the optimal strategy of any player at any given state of the game (node in the game-tree) is determined by
the minimax value, which is defined as [108],

\[
\text{minimax}(s) = \begin{cases} 
\text{utility}(s) & \text{if } s \text{ is terminal state} \\
\max_{\alpha \in \text{actions}(s)} \text{minimax}(\text{result}(s, \alpha)) & \text{if MAX makes move} \\
\min_{\alpha \in \text{actions}(s)} \text{minimax}(\text{result}(s, \alpha)) & \text{if MIN makes move}
\end{cases}
\]

where \( \text{utility} : S \rightarrow \mathcal{R} \) is the utility function that assigns the utility value to a given state \( s \), \( \text{result} : S \times A \rightarrow S \) denotes the next state if the given action \( \alpha \) is taken by the player, and \( \text{action} : S \rightarrow A^* \) returns a set of legal actions that available to the player at the given game state \( s \). Using the minimax value, the optimal choice for, for example, a maximizing player can be determined by simply selecting the action that leads to a state with the highest minimax value.

The minimax algorithm performs a complete depth-first exploration by recursively expands the game-tree and calculates the minimax value. Assuming the depth of the tree is some positive integer \( m \) and the branching factor is fixed at some positive integer \( n \), the time complexity of the minimax algorithm is \( \mathcal{O}(n^m) \), and the space complexity is \( \mathcal{O}(nm) \). Thus, the minimax tree search itself does not reduce the search space as well as eliminate the need to explore the complete game tree, which makes the algorithm impractical for large game-trees.

### 2.3.2 Heuristic Function in Minimax Tree Search

The minimax tree search relies on the calculation of the utility value of the terminal state, and propagate back to its ancestors in the search tree. To reduce the search space, it is possible to limit the depth of the search tree to a pre-defined value and use a static function to estimate the minimax value of the node in the given depth instead. Such function is normally constructed based on the game, and it is understood to be
2.3. Game Tree Search

very difficult to obtain an accurate estimation in a complex game [24].

2.3.3 Alpha-Beta Pruning

The minimax algorithm does not reduce the complexity of a complete game-tree search, while it only provides the means to help the decision making for MAX/MIN player. However, it is possible to compute the correct minimax value without exploring the complete game-tree to make exactly the same decisions. A particular technique for it is *Alpha-beta pruning*, or $\alpha - \beta$ pruning, which, when applied, will favour the same moves as the minimax algorithm on a complete game-tree would, but prunes (not search, nor explore) the branches of the tree that do not influence the decision during the search.

Let $\alpha$ be the value of the current best choice for a maximizing player in a partially explored game tree, a.k.a the current highest minimax value and $\beta$ be the value of the current best choice of the same search for the minimizing player, a.k.a the current lowest minimax value, similar to branch-and-bound algorithm [76], the pruning is taken place as soon as the current node is “worse” than the current $\alpha$ or $\beta$ respectively. That is, for the maximizing player, if the value of the current node is lower than $\alpha$, or for the minimizing player, if the value of the current node is higher than $\beta$, recursion is terminated and move on to other nodes that are not yet searched (see Algorithm 2.2).

2.3.4 Monte Carlo Tree Search

*Monte Carlo tree search* (MCTS) algorithm is well-known to be useful for game playing, and has been used in games with randomness and/or partial observability, as well as in deterministic games of perfect information [29, 20], for example, Chess, Go, and 2048 (a simple video game with randomness). MCTS does not require to fully explore the

---

*based on [108] with revised notations that are consistent with this thesis*
Algorithm 2.2 The minimax tree search with $\alpha$-$\beta$ pruning

1: function MINIMAXTREESEARCH(s)
2:     $v \leftarrow$ MAXVALUE(s, $-\infty$, $+\infty$)
3:     return action $\alpha \in$ actions(s) s.t. $\min\max(\text{result}(s, \alpha)) = v$
4: end function

5: function MAXVALUE(s, $\alpha$, $\beta$)
6:     if s is terminal state then
7:         return utility(s)
8:     end if
9:     $v \leftarrow -\infty$
10:    for each $\alpha \in$ action(s) do
11:        $v \leftarrow \max(v, \text{MINVALUE(result}(s, \alpha), \alpha, \beta))$
12:        if $v \geq \beta$ then
13:            return $v$ \hspace{1cm} $\triangleright$ $\beta$ cut-off
14:        end if
15:        $\alpha \leftarrow \max(\alpha, v)$
16:    end for
17:    return $v$
18: end function

19: function MINVALUE(s, $\alpha$, $\beta$)
20:     if s is terminal state then
21:         return utility(s)
22:     end if
23:     $v \leftarrow +\infty$
24:    for each $\alpha \in$ action(s) do
25:        $v \leftarrow \min(v, \text{MAXVALUE(result}(s, \alpha), \alpha, \beta))$
26:        if $v \leq \alpha$ then
27:            return $v$ \hspace{1cm} $\triangleright$ $\alpha$ cut-off
28:        end if
29:        $\beta \leftarrow \min(\beta, v)$
30:    end for
31:    return $v$
32: end function
2.3. Game Tree Search

search tree, instead, it utilizes Monte Carlo sampling \[1\] to evaluate the current nodes of the game tree. It expands and evaluate the game tree incrementally to balance the exploration and exploitation \[20\], until some computation budget (e.g. time, memory, or iteration constraint) has been reached, at which point then the action that leads to the best child node of the current search root (representing the current game state) is returned. The exploration is to look in the branches of the tree that have not yet been well sampled. The exploitation then evaluates the currently explored branches and decide which is the most promising (e.g. most likely to win for a given player). The sampling is to simulate the gameplay number of times using a default policy from the game state that is represented by the selected leaf node of the partially expanded game tree until reaching a terminal state, then collect the win/loss statistics of the given player. The default policy can be seen as the behaviour models of the players involved in the game. A basic behaviour is to assume all players make random and legal moves in the simulated game play, namely flat Monte Carlo, for example, which has been used and achieved champion-level play in Bridge and Scrabble \[117, 55\]. The win/loss statistics collected from the sampling is used as the odds of winning at the node. Consequently, the accuracy of the statistics approximate depends on the number of the samples (number of the simulated games) and the default playing policy \[20\]. For example, MCTS used to be the state-of-the-art algorithm for Go-playing agent, but never be able to compete with professional Go players \[120\]. Then by replacing the policy with a trained deep neural network (and some other tweaks) the Go-playing agent is able to play against professional players and win \[120\].

The basic MCTS algorithm iteratively constructs a game tree. Each iteration has 4 steps (illustrated in \[Figure 2.3\]),

1. **Selection**: recursively select the child node from the root node to descend to the most promising expandable (non-terminal or not fully expanded) leaf node
2.3. Game Tree Search

Figure 2.3: One Iteration of Generic MCTS

of the current partial game tree.

2. **Expansion**: expand the selected leaf node by adding child node or child nodes according to the legal moves available to the game state of the selected leaf node.

3. **Simulation/Sampling**: simulate the game play according to the default policy from game states represented by the new node or new nodes.

4. **Backpropagation**: propagate the simulation result ascending through the tree branch.

Browne et al. further summarize these steps to 2 distinct policies \[^{20}\] tree policy and default policy, so that the MCTS algorithm can be described as Algorithm 2.3. The tree policy, combining selection and expansion, select an expandable node and create a leaf node or leaf nodes to simulate. The default policy simulates the gameplay from the game state of the selected node from the tree policy. In \[^{29}\], Coulom proposes a different approach that, instead of simulating game play and collecting only the statistics, it combines minimax tree search with MCTS by allowing the simulation to

\[^{7}\] This illustration expands one new leaf node in every iteration.
2.3. Game Tree Search

grow the tree by adding searched states to reveal the structural information of the
game tree from sampling.

**Algorithm 2.3** Generic MCTS Algorithm

1. \textbf{function} MctsSearch\( (s_0) \)
2. create root node \( v_0 \) with State \( s_0 \)
3. while within computational budget do
4. \( v_l \leftarrow \text{TreePolicy}(v_0) \)
5. \( \Delta \leftarrow \text{DefaultPolicy(stateOf}(v_l)\) \)
6. \text{Backpropagate}(v_l, \Delta)
7. end while
8. return \text{actionOf}(\text{BestChild}(v_0))
9. end function

The backpropagation is to update the node statistics after the iteration using the
reward value \( \Delta \) produced in the simulation. When updating, the visit count of each
node visited, i.e. all the ancestor nodes of the leaf node where the simulation is run,
the iteration is incremented, and the average reward value is updated. The average
reward is commonly understood as a function of visit count and the reward values of
each visit, where the reward value can be a discrete value such as win, draw, or loss,
a real number, or a vector of reward values related to each player in a more complex
game.

Once the computation budget is reached (or manually interrupted in some implementation),
the search is terminated and an action from the root node is selected as the best
possible move according to some criteria. The selection criteria can differ from case
to case. Chaslot et al. find there are 4 common definitions of the “best” child [26],

1. \textbf{max child} select the child that has the highest (average) reward value.

2. \textbf{robust child} select the child with the highest visit count (given that the tree
   policy may prefer to select nodes with higher reward).

\footnote{based on [20] with revised notations that are consistent with this thesis}
3. **max-robust child** select the child with both highest visit count and highest reward value, and if there does not exist such a child, the search will continue until such a child is found [29].

4. **secure child** select the child that maximizes a lower confidence bound (see Section 2.3.5).

MCTS becomes the popular algorithm in different domains with success because of its characteristics. Browne et al. [20] summarize there are three major characteristics, *aheuristic*, *anytime* and *asymmetric*, that make MCTS popular among other tree search algorithms.

Being aheuristic means MCTS does not require (but such information can be useful) for domain-specific knowledges. It does not require to use any static heuristic function that defines the quality of limited-depth minimax tree search. Additionally, it does not require to design a heuristic function that can accurately estimate the utilities of the game state at all. If the domain knowledge is available, it is possible to utilize it in the search such as in tree policy and default policy (i.e. instead of uniformly random simulating the gameplay, use domain knowledge to make more informed moves).

Being anytime allows the algorithm to return the best action (so far) from the root at any moment. This is because of the iterative nature of the algorithm. In each iteration of the search, all the values of the partial tree are always up-to-date with the latest sampling result.

Asymmetric search allows the algorithm to search the more promising nodes more often but without eliminating the possibility of exploring previously unvisited nodes. As a result, it commonly leads to an asymmetric partial search tree where the high reward nodes are exploited (visited) more times and the part of the tree that has high-value nodes are expanded deeper.

Research argues that the classic MCTS relies on the randomness of the sampling
thus the decisions made by MCTS may not correspond to the game theoretic optimum like from the most alpha-beta based algorithm \[71\]. This is to say that it is possible to obtain a sub-optimum decision from MCTS when, in a small probability, the random sampling reaches to the conclusion that a node has high value when the actual winning probability is low because of the low sampling rate or the search is interrupted prematurely. Since the value of a given node is dependent on the value of its child nodes, the more accurate in estimating the value of its child nodes, the more accurate the estimation of the node will be. In \[29\], this is overcome by progressively averaging the minimax values as the number of simulation grows to allow more effective selection of nodes to explore.

Another issue that MCTS faces is to balance the exploitation of the currently most promising action and the exploration of the alternative options when expanding the game-tree. With the limited computing budget, too much exploitation done would lead to suboptimal decisions due to only a small amount of alternative actions are considered. On the other hand, too much of exploration may also lead to suboptimal decisions because there may not be a statistical significance to differentiate possible options due to the lack of samplings. One of the popular solutions is to treat the selection problem as a multi-armed bandit problem, in which *Upper Confidence Bounds* (UCB) is commonly used to determine any given arm will be optimal. Auer et al. \[4\] proposed a simple UCB policy, named UCB1, that defined as

\[
UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}}
\]

(2.1)

where \( \bar{X}_j \) is the average reward of the arm \( j \), \( n_j \) is the number of times arm \( j \) was pulled, and \( n \) is the overall times of plays. The exploitation and exploration are then balanced by the two terms of the equation, where the first term, \( \bar{X}_j \), encourages to play the arm with higher average rewards so far and the second term, \( \sqrt{\frac{2 \ln n}{n_j}} \), will
decrease if the same arm $j$ was played many times, thus encourages plays on less played arms/choices. The development of MCTS combined with UCB1 is discussed later in Section 2.3.5.

### 2.3.5 Upper Confidence Bounds for Trees (UCT)

As the most popular algorithm in the MCTS family, the *Upper Confidence Bounds for Trees* (UCT) is inspired by UCB1 selection in multi-armed bandit problem to ensure the search converging to the game-theoretic optimum if sufficient computing budget is given and having a small error probability if the search stopped prematurely. The goal of developing UCT is to approximate the game-theoretic optimal value of actions in the search thus improves the quality of decisions with finite computing budget, by balancing the exploitation and exploration when constructing the game-tree.

MCTS iteratively build a partial game-tree instead of building the complete tree like the basic minimax algorithm and resulting partial tree is dependent on how the node is selected in the tree policy in each iteration of the search. Kocsis et al. [72, 71] propose to use UCB1 as the tree policy, by which a child node is selected to maximize

$$ UCT = \bar{X}_j + 2C_p \sqrt{\frac{2\ln n}{n_j}} $$

where $n$ is the number of times the current (parent) node has been visited, $n_j$ is the number of times the child node has been visited, and $C_p > 0$ is a constant. In the case where there is more than one node that has the same maximum UCT value, the tie is broken by randomly selecting one of them. Note that $n_j = 0$ is normally understood to have the maximum UCT value, $+\infty$, thus the previously unvisited node is assigned with the largest UCT value possible to ensure all the child nodes of the current parent are visited at least once before expanding to the next level of the game-tree. The constant $C_p$ controls the balance of exploration and exploitation. Decreasing
2.3. Game Tree Search

$C_p$ decreases the amount of exploration. The value $C_p = 1/\sqrt{2}$ is shown to satisfy the Hoeffding inequality with rewards in the range $[0, 1]$ [71].

The generic version of UCT algorithm is shown in Algorithm 2.4 summarised by [20]. The basic search routine, $UCTSearch$, is the same as in the generic MCTS algorithm (see Algorithm 2.3). The tree policy expands the node if the node is not fully expanded or selects the best child of the fully expanded node according to their UCT value. In $TreePolicy$, $stateOf(v)$ is used to represent the state of the game that is represented by the node $v$ and $actionOf(v)$ is the move taken that leads to the state of $v$. The function $result : S, A \rightarrow S$ calculates the new game state if a given action is taken on the given game state according to the game rules. The default policy is used to simulate the gameplay (at uniformly random moves, in this simplest case) to a terminal game state. Then the value $\Delta$ is backpropagated to all nodes visited during the iteration of the search.

In the search tree, each node $v$ holds two values, the number of time it been visited, denoted by $N(v)$, and the total reward $Q(v)$ of all playouts that has the node $v$ as their ancestor. Thus $Q(v)/N(v)$ can be used as an approximation of the node’s average reward in UCT value in the equivalent of $\bar{X}_j$ in Equation 2.1 and Equation 2.2. In every iteration of the search, the two values will be updated for every node visited as described in Algorithm 2.5 and Algorithm 2.6. Algorithm 2.5 shows the backpropagation without the constraints of two-player, zero-sum, and turn-order, which are typically found in other literature, while Algorithm 2.6 shows the version of two-player, zero-sum, and alternative moves.

The significance of UCT algorithm is, as in [72, 71], that given unlimited computing resources the UCT algorithm will converge to the game-theoretic optimal decision at the root of the tree. In other words, the probability of selecting suboptimal moves at the root of the tree decrease to 0 when the number of simulation grows to infinity.
Algorithm 2.4 UCT algorithm [20]

1: function UCTSearch($s_0$)
2:     create root node $v_0$ with state $s_0$
3:     while within computational budget do
4:         $v_l \leftarrow$ TreePolicy($v_0$)
5:         $\Delta \leftarrow$ DefaultPolicy(stateOf($v_l$))
6:         Backpropagate($v_l$, $\Delta$)
7:     end while
8:     return actionOf(BestChild($v_0$))
9: end function

10: function TreePolicy($v$)
11:     while $v$ is nonterminal do
12:         if $v$ not fully expanded then
13:             return Expand($v$)
14:         else
15:             $v \leftarrow$ BestChild($v$, $C_p$)
16:         end if
17:     end while
18:     return $v$
19: end function

20: function Expand($v$)
21:     choose $\alpha$ in untried actions from $\mathcal{A}$ that is legal on stateOf($v$)
22:     add a new child $v'$ to $v$
23:     stateOf($v'$) $\leftarrow$ result(stateOf($v$), $\alpha$)
24:     actionOf($v'$) $\leftarrow$ $\alpha$
25:     return $v'$
26: end function

27: function BestChild($v$, $c$)
28:     return $\arg\max_{v' \in \text{children of } v} Q(v') + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$
29: end function

30: function DefaultPolicy($s$)
31:     while $s$ is non-terminal do
32:         choose $\alpha \in \mathcal{A}$ that is legal uniformly at random
33:         $s \leftarrow$ result($s$, $\alpha$)
34:     end while
35:     return reward for state $s$
36: end function
2.3. Game Tree Search

Algorithm 2.5 UCT backpropagate [20]

1: function Backpropagate(v, Δ)
2:     while v is not null do
3:         N(v) ← N(v) + 1
4:         Q(v) ← Q(v) + Δ(v, p)
5:         v ← parent of v
6:     end while
7: end function

Algorithm 2.6 UCT backpropagate for 2 players [20]

1: function Backpropagate(v, Δ)
2:     while v is not null do
3:         N(v) ← N(v) + 1
4:         Q(v) ← Q(v) + Δ
5:         Δ = −Δ
6:         v ← parent of v
7:     end while
8: end function

2.3.6 Conclusion

Game tree search is a large field of research in AI and there is a long history of advancements. This section only reviewed the classic and the most popular algorithms (without many of their variants) that are directly related and contribute to this thesis.
Part I

BDI Agent Semantics and Its Application
Chapter 3

Semantic Annotation of BDI Agent Programs

With most agent applications being situated in increasingly stringent legislative and regulatory environments, ensuring that these applications are compliant is a significant concern. This chapter offers solutions for design-time compliance analysis of BDI-Agent programs. The approach relies on agent programmers providing normative effect specifications for goals and actions, together with a machinery for accumulating and contextualizing these effects to obtain abstract (and potentially partial) descriptions of the effects that are obtained at each step in the execution of the agent. The framework offers useful guidance for compliance resolution, i.e., redesign of the agent to address non-compliance. The chapter evaluates the complexity of this analysis using MAX-SAT\(^1\) solvers and demonstrates that the approach is practical.

\(^1\)Maximum Satisfiability problem
3.1 Introduction

When developing BDI-agent systems, developers are designing plans and logic rules to address a specific set of goals the agent would achieve in a dynamic environment. The problem arises when the development is done, how the correctness of the plan library can be checked given that we don’t know what the state of the environment would be like when the agent start running. Software testing techniques may be employed for it but it is a common practice to design test cases to verify a computing system performs as expected in these cases. However, BDI-agent systems are commonly designed to deploy in a loosely controlled environment compared with more classic software systems where the state of the execution can be well-defined. The designed test cases may never cover all the situations the agent system would face \[93\] given that the environment may also change unexpectedly in the runtime.

In some case, the agent systems’ behaviour is also important. When the agent is trying to achieve a given goal, it follows the pre-defined the plans according to the current runtime situation. However, in some domain, some behaviours may not be acceptable or not allowed as these could lead to violations of some other requirements. Because the agent finds applicable plans in the run time, it is hard to find out what combination of plans the agent would actually select for any given goal in any point of time. The compliance checking is to check if it is possible for an agent to behave according to a given set of compliance rules and to find out in what situations where the agent may behave out-of-bound. Our aim is to check possible agent behaviours for compliance. We focus on design-time compliance analysis, which means that we cannot predict with any certainty what might happen in the world, in which the agent is situated, nor predict how effectively an agent is able to update its beliefs. In relation to the belief update machinery, we can make one of the following 2 assumptions, which, between them, cover all possibilities:
• Perfect belief update: This assumes that the agent has the capability to update its beliefs in a manner that adequately reflects changes in its environment, which also guarantees that at least one of the predicted effect scenarios will actually be entailed by the agent’s beliefs after an action or plan has been executed.

• Imperfect belief update: This assumes that the agent’s belief update machinery might not be capable of reflecting all the changes that occur in the world in which the agent is situated adequately or completely in its beliefs. Under this assumption, the agent’s beliefs will never be unsound (i.e., contain beliefs that should not be true) but may be incomplete. If this is the case, there is no guarantee that at least one of the predicted effect scenarios will be entailed by the agent’s beliefs after an action or plan has been executed.

Ultimately, the compliance analysis machinery we seek to develop may generate two kinds of findings:

• Possible non-compliance: In this case, we flag that there exists a possible behaviour, given the current agent program, and given some feasible evolutions of the world in which the agent is situated, such that some compliance rules are violated. If the agent’s belief update machinery is perfect (in the sense of the definition above), then we would expect this analysis to be correct. More interestingly, the correctness of this finding is not impacted if the agent’s belief update machinery is imperfect. Imperfect belief update entails that certain plans that should have been found to be applicable will now not be found to be applicable, leading to set of possible behaviours being restricted (but never expanded, given that all beliefs are still sound).

• Necessary non-compliance: In this case, we flag the possibility that all behaviours will lead to a compliance violation.
The following sections start by defining the semantics (the effect on the possible states of the execution environment) of BDI agent’s behaviour including preforming an action, and selecting a plan from the plan library (Section 3.2). Section 3.3 presents a way of mapping out the traces of possible environmental state changes if the agent were in action, then analyses the soundness and completeness of the plan library with respect to the goals of the agent to identify the possible scenarios where the agent either not able to achieve a goal (soundness) or a goal cannot be achieved by the given plan library (completeness). Section 3.4 demonstrates how the compliance checking can be done at the design time using the traces of the state changes the agent is capable of bringing about and the existing model checkers. In Section 3.5, we evaluate the cost of the state update operator that is defined in Section 3.2 which is the core of creating traces of environmental state changes that is the base of all the analysis in this chapter. We find the cost of the operator can be considered acceptable in a practical system even though the operator itself is modelled as an MAX-SAT problem that is NP-hard. In the end, we review the literature related to this chapter in Section 3.6 and remark our finds and discuss the possible future works in Section 3.7.

### 3.2 Semantics of BDI-Agent Plans

We are aiming to represent the semantics of a BDI-agent plan, set of plans and, eventually, the entire BDI-agent system, in order to offer a basis for formally analysing the system, such as functional correctness and completeness, and compliance of the system implementation.

To do this, we assume that each action is related to a set of effects that are mutually exclusive. Each of the effects can be viewed as a context-independent, epistemic input if the actions were executed successfully (given the state of the system can be described in epistemic states). In the most general understanding of the effects, there is no
3.2. Semantics of BDI-Agent Plans

limitation on the number of alternative effects that could be associated with one action
due to the context when the action was carried out, or, in this case, the state that
describes the entire system. For example, a simple action “flip the switch” could result
in either the connected light on or off depending on state of the light before the switch
was flipped, i.e the context of the system. The result of action “submit paper” to a
conference or “lodge application” could be either “accepted” or “rejected”, generally
speaking.

We assume that all actions in an agent program (and possibly other agent programs
in a multi-agent system) are drawn from a capability library, which may be viewed as
a set of pairs of the form \( (\alpha, E) \), where \( \alpha \) is an action identifier and \( E \) is a set of effects, such as

\[
(\text{submit}(\text{Paper}), \{\text{accepted}(\text{Paper})\}, \{\text{rejected}(\text{Paper})\})
\]

We assume that each effect is represented by a set of declarative sentences, while
the inter-relationships and rules are represented in a background knowledge base \( KB \).
For example,

\[
KB = \{\text{accepted}(\text{Paper}) \rightarrow \text{submitted}(\text{Paper}), \\
\text{accepted}(\text{Paper}) \rightarrow \neg \text{rejected}(\text{Paper}), \\
\text{rejected}(\text{Paper}) \rightarrow \neg \text{accepted}(\text{Paper})\}
\]

We noticed, in general, restricting the expressiveness of these rules can offer efficiency
gain in the usual fashion, but it is beyond the topic of this chapter. Moreover, any
reference in the following of this chapter to consistency or consistency of a set of
assertions \( S \) is implicitly the consistency of \( S \cup KB \).

Similarly, each goal/sub-goal is annotated with effects, or, in this case, its declarative
descriptions in the same formal language. This bases on the intuition that an (achievement)
goal must be representable as an assertion that one is intended to make true. Yet, as with actions, one might imagine a goal to assess an insurance claim being realized in two ways, one by achieving a state of affairs where the claim is accepted, and another by achieving a state of affairs where the claim is rejected. Note that this situation cannot be dealt with by requiring that every possible state where a goal is achieved entails the same goal assertion. Thus, we allow the goal to be annotated with a set of effects/assertions. For example, the goal “submit paper” is achieved by the following state of affairs:

\{
\text{submitted(Paper), submitted_to(Paper,C), conference(C)}\}

\{
\text{submitted(Paper), submitted_to(Paper,J), journal(J)}\}

where, one of the state indicates the paper is submitted to a conference, and another is to a journal.

The idea of declarative effects of actions and goals is featured in some of BDI-Agent programming language but named differently, and they are described using different declarative languages, such as 3APL [113], GOAL [63], etc. On the other hand, the concept of knowledge base exists in almost every BDI-agent programming language in various forms.

To understand the semantics of a BDI-agent plan, we first have to discuss how completing an action or achieving a goal affects the agent’s belief state and everything else within the same system (e.g. the environment and other agents’ belief states). Assuming the current state of the system is described by a set of assertions \(s\), at least one effect \(e\) (\(e \in \mathcal{E}\)) of the action or goal must hold in the possible set of assertions \(s'\) that describe the state of the system immediately after the completed action and/or achieved goal. That is, there is a transition function \(f : \mathcal{S} \rightarrow \mathcal{S}\), where \(\mathcal{S}\) are all
possible state descriptions of the system. However, it may not be as simple, as what has been argued by Ginsberg in [56]. For example, a book $A$ is placed on top of a book $B$; after an action resulting the relocation of the book $B$ to its right for 20cm, the book $A$ could be at many possible locations such as $A$ is still on top of $B$, $A$ is remaining at where $B$ used to be, somewhere in between of original location of $B$ and new location of $B$, somewhere at the right of $B$, etc. Any of the combination of statement description location of $A$ and $B$ would result in a consistent state description. Thus, the function $f$ should result in a set of consistent state descriptions, each of which describing a possible world, i.e. $f : \mathcal{S} \rightarrow 2^\mathcal{S}$. The function $f$ might be defined in a variety of ways, leveraging alternative intuitions from the literature on reasoning about action. The following definition is suitable for our current purposes, while much of our framework is general enough to admit alternative definition of $f$, and is based on the possible worlds approach [56].

From now, we shall refer a state description $s$ as a set of sentences/assertions in a formal language $\mathcal{L}$ that describes any objects and agents in the agent system, including but not limited to the environment, active agent and other agents, and $\mathcal{S}$ is a set of such descriptions. This state description may not be equivalent to the agent’s belief, as belief in the most BDI-agent implementation is complete by definition (i.e. close world assumption is applied in the most cases that use Prolog\footnote{Prolog is a general-purpose logic programming language.} syntax), and belief of BDI agents in the implementation is not necessarily up-to-date with the environment due to the agents implemented with “reasoning cycles”. However, if there is such a BDI system that is capable of updating agent’s belief $b$ in synchronised fashion to its environment, then we would expect the following relation holds with the state description $s$. language that describes the state of objects, agents, etc in the execution environment.

$$b \subseteq \text{Cn}(s \cup \mathcal{KB})$$
where $\text{Cn}(s)$ denotes the deductive closure of $s$. In modelling the semantics of more complex agent systems, it is possible that the above relationship does not hold. In other words, the agent has its own frame of reference about the world, which could cause the agent to form a contradictory theory about the world from its belief.

### 3.2.1 Semantics of Agent Execution

We view the agent’s executing actions and plans as it gradually changes the system. The change could exist in the agent’s mind (e.g. via belief updates or perceptions), affect the world (e.g. the action that result in an object changing its state) or both.

**Definition 3.1.** Given a state description $s$, and a set of effects $E$ of an action or a goal that is to be performed or achieved, the outcome state descriptions $S'$ are defined as

$$S' = \bigcup_{e \in E} s \oplus e$$

for each $s' \in s \oplus e$ and

1. $s'$ is consistent ($s' \not\equiv \bot$),
2. $s' \subseteq s \cup e$,
3. $e \subseteq s'$,
4. $\exists s''$, s.t. $s' \subset s'' \subseteq s \cup e$ and satisfies (1), (2) and (3).

Upon successfully completing an action or achieving a goal, it is expected that the effect $e$ of the action or goal are present in the new state description $s'$ ($e \subseteq s'$), as well as preserving the information about other objects from $s$ as long as there is no reason to believe it is incorrect, e.g. inconsistent with the effect of actions and goals. Moreover, we allow the possibility that an action/goal could result in multiple different
3.2. Semantics of BDI-Agent Plans

effects, i.e. they may affect the world differently under different undefined contexts. For example, given a state description

\[ s = \{ \text{power}(\text{connected}), \text{light}(\text{broken}) \} \]

and the action “flip the light switch” that has the following known effects,

\[ e_1 = \{ \text{light}(\text{on}) \} \]
\[ e_2 = \{ \text{light}(\text{off}) \} \]

By performing the action, according to the clause (2) and (3) in Definition 3.1 we would expect the following state descriptions that describes the possible worlds the action would create,

\[ s'_1 = \{ \text{power}(\text{connected}), \text{light}(\text{broken}), \text{light}(\text{on}) \} \] (3.1)
\[ s'_2 = \{ \text{power}(\text{connected}), \text{light}(\text{broken}), \text{light}(\text{off}) \} \] (3.2)

Obviously, the state description [Equation 3.1] is inconsistent as it is impossible for a broken light being on, which does not satisfy the clause (1) in Definition 3.1. Because we respect the latest information \text{light}(\text{on}) more than information of the previous state \text{light}(\text{broken}), we would revise the state to

\[ \{ \text{power}(\text{ok}), \text{light}(\text{on}) \} \] (3.1)

to keep it consistent. On the other hand, state description [Equation 3.2] is consistent as it is possible for a light being broken and off at the same time.
3.2.2 Semantics of Plan Selection

Since a state description $s$ normally contains incomplete information of the world at the design time (e.g. state of other agents or some object is only available to the agent at runtime), all the extra information about the system that exists in the agent program should also be included, such as the context of a plan. Assuming when a plan is selected for execution on the given state $s$, it must be the case that the context $c$ of the plan is met according to the BDI-agent languages. In this case, we know that the $c$ and $s$ should be describing the same state of the world. In the design time analysis, the information available is limited. For example, we may lack of information of the agent’s perceptions. If we only work with the effects of actions and goals, it is possible that the state description at the plan selection may not include all the necessary assertions to prove $c$ holds. On the other hand, $\neg c$ being concluded from a state description makes it a clear case that the plan with the context $c$ cannot be selected to achieve goal.

**Proposition 3.1.** The plan $p$ is not feasible at a given state description (potentially partial) $s$ when $s \models \neg c$.

*Proof.* Given a context $c$ which consists of a set of assertions, and a consistent state description $s$. Assume when $s \models \neg c$, the plan that has the context $c$ is feasible to realise the goal. With $s \models \neg c$, it must be the case $\neg c \subseteq \text{Cn}(s)$. We know that if the plan is selected when the contexts are met, denoted by $c \subseteq \text{Cn}(s)$, then we have $c \cup \neg c \subseteq \text{Cn}(s)$. Intuitively, $c \cup \neg c$ is inconsistent, which implies $\text{Cn}(s)$ is inconsistent, and contradicts the statement that $s$ is consistent. Thus, it must be the case that when $s \models \neg c$, the plan is not feasible to realise the goal. \qed

Assuming at the design time, after action “submit paper”, there are two possible
worlds, \(s_1\) and \(s_2\),

\[ s_1 = \{ \text{submitted}(\text{Paper, Conference}), \text{accepted}(\text{Paper, Conference}) \} \]

\[ s_2 = \{ \text{submitted}(\text{Paper, Conference}), \text{rejected}(\text{Paper, Conference}) \} \]

and we are achieving sub-goal “attending conference”, which can be achieved by the following two plans,

@p1

+! attendingConference

: accepted(Paper, Conference) & heldIn(Conference, City)

<- !bookFlight,

...

@p2

+! attendingConference

: accepted(Paper, Conference) & heldIn(Conference, City)

<- !bookTrain,

...

In this case, both plans are not suitable for \(s_2\), given the rule

\[ \text{rejected}(\text{Paper, Conference}) \rightarrow \neg \text{accepted}(\text{Paper, Conference}) \]

However, if any of the plan is applicable on a state, i.e. \(s_1\), there is extra information
in the context that has to be added into the state description. Then we would have

\[
s'_1 = \{submitted(Paper, Conference),
    accepted(Paper, Conference),
    heldIn(Conference, City),
    haveFlight(City)\}
\]

\[
s''_1 = \{submitted(Paper, Conference),
    accepted(Paper, Conference),
    heldIn(Conference, City),
    haveTrain(City)\}
\]

because, if the plan is assumed to be applicable, its contexts must assume true.

Accepting the incomplete information that is available at the design time, we have to assume a plan is feasible and will be selected to achieve a goal if the state description of the system does not show \(\neg c\) holds. With the knowledge of a plan is being feasible, the context of the plan can be then added to the state description as the context and the state description at the plan selection are essentially describing the same state of the system.

For example, a plan body that contains an action “check password” followed by a sub-goal “display message” achieved by three plans. The first plan requires the password is correct and the account is not locked, then it displays a welcome message. Another plan requires the password is incorrect, then it displays a message indicating the login failed. The last plan, when the account is locked for some reason but the user provided the correct password, displays the message telling the user to contact the
administrator. With only information above available (assuming the information of the account being locked is only available at the run time), and checking password would result in either password is being correct or incorrect, when the first plan is selected, the state descriptions is “correct password” and “account is not locked”; when the second plan is selected, the state description contains only “incorrect password”, and “correct password” and “the account is locked” describes the state when the last plan is selected.

Definition 3.2. Given a state description $s$, and a context $c$ of a plan, when $s \cup c \not\models \bot$, $s' = s \ominus c = s \cup c$ and $s'$ describes the same state of the system with $s$.

$\ominus$ operator enriches $s$ with the information in context $c$ that describes the same state if the plan is selected to execute. For example, when $s \models c$, the plan can be selected to achieve the goal. When $s \models \neg c$, the plan clearly is not applicable. When $s \not\models c$ and $s \cup c \not\models \bot$, which shows that we do not have enough information to decide if the plan is possible or impossible to realise the goal on $s$. We assume the plan could be selected because there is no information telling us otherwise. Therefore, we then use the extra information in context $c$ to expand $s$ based on the assumption that $s$ and $c$ are describing the same state of the world.

3.3 Soundness and Completeness of Plan Library

If we look at the plan library as an algorithm, the plan library is sound when the resulting states (output) by using the plan library to realize the goal are all goal states. The plan library is complete if it is able to realize all the goals, i.e. for all goals, the plan library is able to result in all possible goal states.

Soundness and completeness analysis of BDI-Agent programs involves two phases: (1) internal plan analysis, and (2) inter-plan analysis. Internal analysis aims to detect
situation where an agent plan and its associated statements of semantic intent (i.e. the effect of the goal of the plan provided by the programmer) contradicts each other. If every plan in the plan library is internally sound, then the inter-plan analysis can be performed to detect the situations where during execution, there is no plans that are feasible to achieve a sub-goal within the plan and the attempt of realizing the goal has to terminate at possibly a non-goal state, resulting in an unsound plan library, most likely because the actions and sub-goals prior to the sub-goal leads to a predicted state that contradicts to the context/preconditions of every plan achieving the sub-goal.

For the purpose of the following discussions, we assume that all the effects of actions and goals are correctly defined.

### 3.3.1 Plan Internal Analysis

Given a plan $p$ designed to achieve goal $g$, consisting of context $c$ and plan body $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ ($\alpha$ could be either an action or a sub-goal), the predicted state descriptions $s_i$ of the plan $p$ depends on the predicted state description $s_{i-1}$ (denoted by $s_{i-1} \xrightarrow{\mu} s_i$). If the set of alternative effect of an action/sub-goal $\alpha_i$ is given by $E_i$ and the set of accepted goal states of $g$ is denoted by $e_g$, the set of predicted state descriptions $S_i$ is then calculated by

$$S_{i \in [1 \ldots n]} = \bigcup_{s \in S_{i-1}} s \oplus e$$

(3.3)

$S_0$ is a special case as if we do not consider the context $c$ of the plan $p$, then

$$S_0 = S_{Init}$$

(3.4)

given $s_{Init}$ is the initial state descriptions when the plan is selected to execute. Otherwise,

$$S_0 = \{s \oplus c | s \in S_{Init}\}$$

(3.5)
Additionally, we let
\[ S_{\text{end}} = S_n \]  \hspace{1cm} (3.6)
where \( S_n \) is the set of state descriptions when the plan ends. Let us call \( S_{\text{end}} \) and \( s_n \) then effect of the plan.

### 3.3.1.1 Realization of a Goal

**Postulate 3.2.** If a plan \( p \) realizes a goal \( g \), which is described as a set of goal state description \( e_g \), upon given initial state descriptions \( S_{\text{Init}} \), it must be the case that for all \( s \in S_{\text{end}} \), there exists \( e \in E_g \) such that \( s \models e \).

In BDI agent, completing the last element of the plan body \( \alpha_n \) marks the completion of the plan, and on the completion of the plan, the goal that the plan realizing is assumed to be achieved. Therefore, the last set of states description \( S_n (S_{\text{end}}) \) realizes the goal.

In addition to the realization of the goal, this is also interesting to describe the degree of goal conformance of a plan as listed below.

**Strong Goal Identity** iff \( \forall s \in S_{\text{end}}, \exists e \in E_g, \text{ s.t. } s = e \),

**Strong Goal Entailment** if \( \forall s \in S_{\text{end}}, \exists e \in E_g, \text{ s.t. } s \models e \),

**Weak Goal Identity** if \( \exists e \in E_g, \exists s \in S_{\text{end}}, s = e \), and

**Weak Goal Entailment** if \( \exists e \in E_g, \exists s \models e \).

In the case of the strong effect consistency, we prefer the strong effect identity over all other relations. When the strong effect identity is unachievable, a correct plan should always strongly entail its goal (strong effect entailment). Moreover, it is possible to list the cases of all non-realizations and inconsistencies, including,

**Strong non-entailment** \( \forall s \in S_{\text{end}}, \forall e \in E_g, s \not\models e \).
3.3. Soundness and Completeness of Plan Library

Weak non-entailment \[ \exists s \in S_{\text{end}}, \forall e \in E_g, s \not\models e. \]

Strong inconsistency \[ \forall s \in S_{\text{end}}, \forall e \in E_g, s \cup e \models \bot. \]

Weak inconsistency \[ \exists s \in S_{\text{end}}, \forall e \in E_g, s \cup e \models \bot. \]

Note that in the identity and entailment relation, we did not consider the coverage of a plan on the effects of the goal. That is, it is allowed to have \( e \in E_g \), there is no \( s \in S_{\text{end}} \) such that \( s \models e \) or \( s = e \), as \( e \) could be realised by other plans, or not realised by any plan (redundant effect of the goal or incomplete plan library, which will be discussed later). If there exists any non-entailment, it is indicated that the plan does not realize the goal in some or all cases. Furthermore, the inconsistencies indicate that there are errors in the plan.

Internal analysis triggers, and provides guidance for programmers to resolve potential differences between intended and implemented semantics. Once these deviations have been resolved, the inter-plan analysis can start.

3.3.2 Inter-plan Analysis

Definition 3.3. A predicted trace \( pt \) is a sequence of states where every \( s_i \) (\( i > 0 \)), it satisfies \( s_{i-1} \xrightarrow{\mu_i} s_i \) and \( s_0 \) is a start/initial state of the agent system. Moreover, we shall refer the sequence \( (\mu_1, \mu_2, \ldots, \mu_n) \) as the identity of the predicted trace \( pt \), where \( \mu \) indicates the event of agent execution which includes action performed, plan selected, etc.

Each predicted trace indicates one possible execution instance. Assume an abstract plan in BDI-agent language consists of a context \( c \) and a plan body \( \langle \alpha, \alpha, \ldots, \alpha \rangle \), the predicted traces of achieving a goal \( g \) on an initial state \( s_{\text{init}} \) are generated using \[ \text{Algorithm 3.1} \]
Algorithm 3.1 Predict traces of realizing goal $g$ on the initial state $s_{init}$

**Require:** Plan library $L$, and effects $E_\alpha$ of every action $\alpha$

```plaintext
1: function PREDICTTraces($g$, $s_{init}$)
2:   $PT_g \leftarrow \emptyset$
3:   $P \leftarrow$ all plans in $L$ that realize $g$
4:   for each $p \in P$ do
5:     $c \leftarrow$ context of $p$
6:     if $s_{init} \not\models \neg c$ then
7:       $pt \leftarrow (s_{init} \ominus c)$
8:       $PT \leftarrow \{pt\}$
9:       for each $\alpha$ in the plan body do
10:          if $\alpha$ is an action then
11:             $PT' \leftarrow \emptyset$
12:             for each $pt \in PT$ do
13:                $s \leftarrow$ the last state description in $pt$
14:                for each $e \in E_\alpha$ do
15:                   $S' \leftarrow s \oplus e$
16:                   $PT' \leftarrow PT' \cup \{pt \rightleftharpoons (\alpha, s') | s' \in S'\}$
17:                end for
18:             end for
19:          else if $\alpha$ is a sub-goal then
20:             $PT' \leftarrow \emptyset$
21:             for each $pt \in PT$ do
22:                $s \leftarrow$ the last state description in $pt$
23:                $PT'' \leftarrow PREDICTTraces(\alpha, s)$
24:                if $PT'' = \emptyset$ then $\triangleright$ there is no plan can achieve the sub-goal
25:                   $PT' \leftarrow PT' \cup \{pt \rightleftharpoons pt'' | pt'' \in PT''\}$
26:                else
27:                   $PT' \leftarrow PT' \cup \{pt \rightleftharpoons pt'' | pt'' \in PT''\}$
28:                end if
29:             end for
30:          end if
31:       end if
32:   end for
33:   $PT_g \leftarrow PT_g \cup PT$
34: end for
35: return $PT_g$
36: end function
```
3.3. Soundness and Completeness of Plan Library

This algorithm requires the plans and goal to be non-cyclic or non-recursive, that is there is no case such as a plan achieving a goal contains the same goal in the plan body, so that the algorithm terminates. The cyclic and recursive issue can be easily dealt with by, for example, cyclic checking on the partially generated traces within the algorithm and simply pruning the cyclic ones as the same execution event on the same state description will always result the in same new state descriptions, and when some state is reached and there should always be another plan that terminates the recursion.

3.3.3 Soundness and Completeness

Definition 3.4. The plan library is **sound** iff for every \( pt_g \) that intends to realise \( g \), the last state \( s_n \) in \( pt_g \), there exists \( e_g \in E_g \) such that \( s_n \models e_g \).

If there exists a trace \( pt \) that intends to realise the goal \( g \), in which the last state \( s_n \not\models e_g \) for every possible state description of \( g \), the agent system is not sound by definition because there exist the situations that the “output” (the state when the agent “think” it achieved the goal) of the agent is not a goal state. Formally, when a agent program is sound, given possible outputs \( S_n \) which contains every the last state of predicted traces that intend to realise goal \( g \), and all the goal states \( E_g \) of goal \( g \), it must be the case that for every \( s_n \in S_n \), \( e_g \subseteq \text{Cn}(s_n) \) for some \( e_g \in E_g \).

If it is the case \( s_n \models e_g \) for every \( E_g \), \( e_g \not\subseteq \text{Cn}(s_n) \), alternatively, \( e_g - \text{Cn}(s_n) \neq \emptyset \), then some of the assertions that construct the goal state does not exist in \( \text{closure}(s) \). Therefore the goal is not realised, and the agent program yields a result that is undesired and the agent program is not sound in this case.

Definition 3.5. The plan library is **complete** iff for every \( e_g \in E_g \) of a goal \( g \), there is at least one trace \( pt \) such that its last state description \( s_n \), \( s_n \models e_g \).
In the previous section where the discussion mainly focuses on the termination of the predicted traces, i.e. the last/end state description in the trace. Obviously, we did not utilise as much information as it contained in the predicted trace. Intuitively, a predicted trace makes up the instance of the agent realising goal, step by step, along time. Assume the actions are atomic which cannot be divided any further (most likely is that there is no point to divide any further in practices), then predicted traces are representations of the agent’s execution instances. Then we can now look at the compliance of the agent system in run time, in which the compliance is only refers to the compliance of the state descriptions (semantics), rather than the order of actions that are conducted by the agent (of course we can but it is less interesting and we do not need any information on the state descriptions along the execution instances). We consider both state-based compliance and path-based compliance. Our approach is correct under the assumption that the effect annotations are complete and correct. Therefore, the BDI-agent-compliance requirements can be described within two settings,
3.4. Compliance Analysis

1. Atemporal compliance requirements: each compliance requirement refers to one state in an agent’s execution.

2. Temporal compliance requirements: each compliance requirement can refer to more than one state in an agent’s execution.

In the first setting, we only need to check the consistency of the atemporal compliance requirements and all possible state descriptions. Since we already contracted all possible execution instances (predicted traces) in the previous sessions, the only thing left is going through state descriptions in the traces, and if a non-compliant is detected, i.e. the state description is inconsistent with the compliance rule, the trace that is non-compliant illustrates the exact scenario of non-compliance.

For (2) with compliance requirements in linear temporal logic (LTL), compliance checking requires building a history of states (i.e. traces), then applying for example, standard LTL model checking techniques. For example, if the language $\mathcal{L}$ of the state descriptions etc. is a propositional language, the states in a trace can be translated to PROMELA\(^3\) and check against LTL compliance rules using SPIN \([65]\). In this case, the sequence of states in a trace will be represented by a unique process in SPIN, and states that preceding and succeeding a plan selection event have to be enclosed in an atomic sequence because they are describing the same state of the system. Moreover, since each trace is an execution instance, it is also possible to evaluate the concurrency and deadlock of plans when the agent is capable of concurrently achieving more than one goal. However, with the different formal language that is used in the state description could result in some differences in model checking. We shall not discuss any further with model checking in this chapter, as the topic has been discussed by much other literature.

---

\(^3\)PROMELA stands for Process or Protocol Meta Language, which is a verification modelling language introduced by Gerard J. Holzmann along with SPIN \([65]\) model checker. \(\text{http://spinroot.com/spin/Man/promela.html}\)
Figure 3.1: Simple Plan Library

@p1
+!attendConference:
  <- useTravelAgent.

@p2
+!attendConference:
  <- !bookSelf.

@p3
+!bookSelf:
  <- register,
    !bookTransport,
    bookHotel.

@p4
+!bookTransport: airline(From,To)
  <- bookFlight.

@p5
+!bookTransport: station(From) & station(To)
  <- bookTrain.

@p6
+!bookTransport:
  <- bookBus.
Here we are providing a small example of how a predicted trace and temporal compliance requirements (in LTL) can be translated to modelling language, such as PROMELA, and then checked using model checker such as SPIN. Given the following predicted traces of the plan library in Figure 3.1.

\[ pt_1 = (\emptyset, \{registered(Conference), booked(Transport), booked(Hotel)\}) \]
\[ pt_2 = (\emptyset, \{registered(Conference)\}, \{registered(Conference), airline(From, To), booked(Flight)\}, \{registered(Conference), airline(From, To), booked(Flight), booked(Hotel)\}) \]
\[ pt_3 = (\emptyset, \{registered(Conference)\}, \{registered(Conference), station(From), station(To), booked(Train)\}, \{registered(Conference), station(From), station(To), booked(Train), booked(Hotel)\}) \]
\[ pt_4 = (\emptyset, \{registered(Conference)\}, \{registered(Conference), booked(Buse)\}, \{registered(Conference), booked(Buse), booked(Hotel)\}) \]

where \( pt_1 \) corresponds to executing plan \( p_1 \). \( pt_2 \) corresponds to \( p_2 \) then \( p_3 \) and \( p_4 \), \( pt_3 \) corresponds to \( p_2 \) then \( p_3 \) and \( p_5 \), and \( pt_4 \) corresponds to \( p_2 \) then \( p_3 \) and \( p_6 \). For each trace, for example the trace \( pt_3 \) can be translated to PROMELA shown in Figure 3.2 where every atomic sequence represents a state in the trace. However, due to the limitation of the modelling language, we have to take the deductive closure of the states as well as ground all the variables, which is why \( bookedTransport \) is added as the result of \( bookedTrain \). The purpose of this example is not to show translating trace to models for model checker is easy but to demonstrate the possibility of such
approach as this is not the main topic of this chapter, and to highlight the possibilities of using traces and model checker to find deadlocks in a multi-agent environment.

3.5 Experimental Evaluation

The purpose of the experiment we designed is to evaluate the complexity of building the predicted traces. Since the $\oplus$ operator could be implemented in multiple ways (the space of alternative design decision is very large) and we do not suggest that this particular implementation is to be preferred other other possible ones (such claims can only be made after a series of substantive comparative studies, and it is most likely contextually dependent). The possible implementations could include (but not limited to)

1. modeled and solved as a MAX-SAT problem,

2. modeled and solved as a MAX-SAT problem with hard clauses,

3. implemented using a theorem prover,

4. implemented using a theorem prover and a SAT solver

5. implemented using answer set programming (ASP).

6. . . .

However, the particular implementation in this chapter only aims to provide an adequate basis for making a preliminary determination of whether this approach is practical.

According to Definition 3.1, given a consistent state description $s$ and an effect $e$, $s \oplus e$ is a set of new states, in which each of the new state $s'$ is a maximum consistent subset of $s \cup e$ (according to clause (1), (2) and (4) in Definition 3.1), and the effect $e$ must be a part of the new state $s'$ (clause (3)). Therefore, the set of new states are the
3.5. Experimental Evaluation

Figure 3.2: PROMELA model of $pt_3$

```plaintext
bool bookedTransport, bookedFlight, bookedTrain, bookedBus,
       registeredConference, stationFrom, stationTo,
       bookedHotel,

ltl g1 { eventually always registeredConference and
       bookedTransport and
       bookedHotel
   }

proctype pt3
{
    atomic {
        registeredConference = true;
    }
    atomic {
        registeredConference == true;
        stationFrom = true;
        stationTo = true;
        bookedTrain = true;
        bookedTransport = true;
    }
    atomic {
        registeredConference == true;
        stationFrom == true;
        stationTo == true;
        bookedTrain == true;
        bookedHotel = true
    }
}
```
set of all solutions of the MAX-SAT problem of $s \cup e$ where $e$ also holds. For example, given a knowledge base with a rule $p \land q \implies r$, $s = \{p, q\}$ and $e = \{\neg r\}$, there are 2 new states according to [Definition 3.1] $s'_1 = \{\neg r, p\}$ and $s'_2 = \{\neg r, q\}$.

This first design decision that have to be made is the language that is used in the state description, and the computational implementation of $\oplus$. In the following evaluation, the propositional language is used and $\oplus$ is implemented using a SAT4J SAT [9] solver with a procedure inspired by [78] used to compute all maximally satisfiable subsets in which $e$ always holds. This choice of formal language and implementation of operator is mainly a matter of convenience. Choosing first order or higher order logic language introduces issues such as semi-decidability that is not yet addressed in this chapter.

The next step is to collect a set of Agent programs with annotated effects of goals and actions. Here we use a collection of systematically generated programs with effects, controlled by the following variables,

**Language Size** from $\{20, 30, 50\}$ is the total number of primitive symbols in the language $\mathcal{L}$.

**Context Size** from $\{2, 3, 5\}$ is the number of conjunctive clauses in each context.

**Knowledge Size** from $\{20, 40\}$ is the number of conjunctive clauses in the knowledge base $KB$, given $KB$ is generated in conjunctive normal form.

**Rule Size** from $\{2, 5\}$ is the maximum number of disjunctive literals in every conjunctive clauses in $KB$.

**Effect Size** from $\{1, 3, 5\}$ is the number of conjunctive literals in effect $e$, as the disjunction can be represented in an alternative effect (i.e. the set of effects $E$ of an action can be seen as disjunctive normal form).

**Number of Goals** from $\{2, 5, 10\}$ is the total number of goals in a plan library.
3.5. Experimental Evaluation

**Number of Plans** from \( \{1, 2, 3\} \) is the number of plans that achieve one goal in the plan library.

**Plan Body Size** from \( \{5, 10\} \) is the total number of action and sub-goals within a plan.

**Number of Sub-goals** from \( \{1, 2, 3, 5\} \) is the maximum number of sub-goals that are allowed in a plan body, and there should always be plans that does not contain sub-goal.

Also, the consistency of semantics is tested during the generation, for example, (1) \( KB \) is always consistent, (2) context of plans are consistent with \( KB \) and context of plans achieving the same sub-goal are different but not necessarily mutually exclusive, and (3) the plans does not contain plan-sub-goal cycle, that is each plan library can be represented in finite goal-plan trees. With all combination of the values of these variables, the minimal number of plan library that can be generated is 7776. A subset of 1390 plan libraries is used in the evaluation, and for every state description at a plan selection, there is at least one plan that can be selected.

Every plan library and corresponding effects is then used as input to Algorithm 3.1 to construct traces. Every run of Algorithm 3.1 is timed and the resulting traces is recorded. Generation of traces for a plan library is run 10 times and timed separately to eliminate the errors that could be introduced by the hardware and software environment. The evaluation is run on Intel® Core™ i5–4440 with 16GB memory in Ubuntu 12 and Java SE 7.

Figure 3.3 demonstrates some interesting relation between the time spent of Algorithm 3.1 on each plan library and variables that represent the structures of the plan library. For every plan library, the construction of trace generation time is illustrated in Figure 3.3a. Since the complexity of plan library is different, the more complex plan library could lead to a longer time of execution. Therefore, we count the total number...
3.5. **Experimental Evaluation**

of state descriptions within traces for every plan library to compute the average time spent on computing a state description for every plan library, which is illustrated in Figure 3.3b. It can be seen that the most of the population distributed within 200 seconds for generating all traces, and within 12.5 seconds for generating a single state description. Figure 3.3c and Figure 3.3d shows the total time taken to explore traces of a plan library compared with the number of traces and state descriptions explored of the same plan library. Generally speaking, the maximum time taken for a plan library is stable when the number of traces and state descriptions increase, while the minimum time taken increases. Thus, the more complex the plan library, i.e. more possible run time instances (traces) and more state descriptions (the results of either a longer plan body, more sub-goals, etc.) possibly requires longer time to explore all possible traces. Moreover, the stable maximum time spent means that even with a simple plan library with only one trace, it is always possible to take long time to explore, which means the semantics of the plan library (i.e. knowledge base and effects) defines a difficult SAT problem that takes longer time for each $\oplus$ operator to solve.

The variable that directly connects to the average time of exploring a state description is the effect size, i.e. the number of conjunctive literals that is used in each effect as shown in Figure 3.4a because $\oplus$ operation is implemented as solving maximum satisfiability problems, the larger the effect, possibly means a more difficult problem. On the other hand, the context does not affect the average time by much (Figure 3.4b) as plan selection is only an evaluation of consistency between a state description and the context, and $\ominus$ operator is only taking the union of two sets of assertions. We cannot find any strong evidence in other variables that shows any effect on the overall time spent and the average spent.

Overall, our approach for evaluating agent program is plausible in practices, as we have explored all possible execution instances of 1390 plan libraries, each of which
3.6 Related Works

Rao and Georgeff discussed the very first model-based theoretic approach to verify BDI-agent system [103], in which three propositional BDI logic languages are defined, $\text{CTL}_{\text{BDI}}$, $\text{CCTL}_{\text{BDI}}$ (Committed $\text{CTL}_{\text{BDI}}$), and $\text{CTL}^*_{\text{BDI}}$, each of which is based on the branching temporal logic Computation Tree Logic (CTL), Fair CTL and CTL*, respectively. After discussing the expressive power and model checking using the
three different languages, Rao and Georgeff summarized that the complexity of model checking for $\text{CTL}_{\text{BDI}}$ has the same or greater complexity than linear temporal logic (LTL) model checking, and has the potential of verifying practical agent-oriented systems using expressive multi-modal, branching-time logic \cite{103}. In recent years, research has focused on extending existing model checking systems and technologies to agent and multi-agent systems. Bordini, Fisher, Pardavila and Wooldridge \cite{14} introduced AgentSpeak(F), a variation of AgentSpeak(L), that can be translated directly to PROMELA, a model specification language for SPIN model-checking system. Similarly, the multi-agent programming language MABLE, proposed in 2002, can also be translated to PROMELA and verified by SPIN \cite{139}. Bordini, Fisher, Visser, and Wooldridge \cite{15} state an alternative approach, in which AgentSpeak is translated to Java and then verified using Java Path Finder (JPF), an existing model-checking tools developed by NASA. Benerecetti and Cimatti \cite{8} presented another approach of validating multi-agent systems, which is based on describing the system using Multi-Language FSM. The expected properties of a system are specified in modal temporal logic, and a decision procedure based on model checking techniques is then introduced. The algorithm described has been implemented in NuMAS system (NuSMV for Multi-
Agent Systems) that is built on top of NuSMV symbolic model checker \(^8\).

Based on the common semantic basis for BDI languages \(^3\), and their previous works on verifying BDI-agent system using JPF, Bordini, Dennis, Farwer, and Fisher discussed the automated verification using this common semantic basis \(^3\). The previous works on model checking agent-based system are mostly bounded to one particular programming language. In \(^3\), the Agent Infrastructure Layer (AIL) that encloses the main concepts of a wide range of agent programming languages is introduced. The idea is that all the agent-based systems developed using languages that are supported by AIL can be translated, and run in an AIL-based interpreter using AIL data structures and Java classes provided by the AIL toolkit. The verification of translated source code is done using AJPF, an extended version of JPF. Some simple case studies on simple agent systems shows AJPF is considered to be more than 150 times faster than JPF \(^3\). However, agent systems studied were too simple to show that the approach is applicable to large scale practical systems.

On the other hand, there are new model checking methodologies and formal languages designed exclusively for agent and multi-agent systems, such as MCMAS \(^7\) that supports CTL, ATL (Alternating-time Temporal logic), and mv \(\mu\)-K-calculus. \(\mu\)-K-calculus \(^7\) is an expressive logic used to specify knowledge and time in a multi-agent system. Hoek and Wooldridge \(^1\) showed \(CKL_n\) (a temporal logic of knowledge) based model checking on distributed agent systems, and showed that it is possible to reduce \(CKL_n\) model checking to linear temporal logic model checking. Furthermore, there are other approaches that aim to provide semi-automatic bug detection, such as in \(^9\) \(^9\) \(^9\) \(^9\), the techniques detect bugs by comparing execution traces with pre-specified interaction protocol, which are beyond our scope as it is bug detection rather than formal verification.
3.7 Conclusion

In this chapter, we firstly discussed the execution semantics of agent programs. Then defined the soundness and completeness semantic properties of the agent plan library, which is the foundation of the correct behaviour of a BDI-agent system. Using the defined $\oplus$ and $\ominus$ operators, given action/goal semantics, it is possible to predict all potential execution instances of a given agent program such that the compliance of the agent system can be evaluated on the semantic and logic level which is more detailed than any existing approach. In the evaluation section, the cost of using $\oplus$ and $\ominus$ operators (in one of many possible implementations) are empirically evaluated. The effectiveness and scalability of our approach is very hard to define, as they depend on many factors.

The semantic views of the agent’s plan library can be a very powerful tool for understanding the behaviour of the designed system without executing the system itself, as well as preforming runtime resolution of errors and/or empowering agent’s decision-making process (see Chapter 4). It also possible to be a bridge of understanding the interactions between intentions of agents in a multi-agent system.
Chapter 4

Enhancing Agent Execution with Semantic Annotations

When an agent system is deployed in a dynamic environment, it is common understanding that some plans may fail during the execution. There is literature discussing the possible resolutions of plan failures, which involves plan revision and plan repair. This chapter provides a possible solution to this issue from a different angle, where instead of handling failures, we provide a way that enhances the agent’s plan selection capabilities so that it will prefer the plans that is less likely to fail before carrying out the plan in its execution. We see plan failures as the result of an ill-behaved execution environment, and the agent is playing a game against the environment trying to reach its goal states. The selection method involves evaluating the applicable plans using game-tree search with respect to the execution environment as an adversarial player in a two player game. We show how both minimax tree search and Monte-Carlo tree search can be used in plan selection to minimise the probability of plan failure.
4.1 Introduction

It is generally recognised that BDI-Agent systems need to be executed in a manner that it is robust and resilient to changes in the operating environment. The challenge is not only to be flexible enough to deal with immediate impediments in dynamic environments but also to anticipate future states of affairs so that the best action or a better action can be made to improve the robustness of the future execution. Impediments to the successful realisation of a goal may be many things, such as an action that is failed, an intention is blocked by another intention, the agent is blocked by other agents’ action, unexpected things occurs in the execution environment. We are not going to discuss the best practices of how a robust agent plan library can be developed, rather a solution that empowers the agent to find out the best solution to resolving runtime impediments individually or collectively given while accepting the limitations that may exist in the plan library.

BDI Agents are commonly running in reasoning cycles. In a reasoning cycle, the agent perceives, pro-act, and/or react to the changes of the environment, which makes the agent more adaptive to the dynamics of the execution environment. The agent first updates its understanding of the world by perceiving the changes of the environment. If an instruction is received (achievement goal received) from its event queue, it selects an appropriate pre-defined plan from its plan library (which then become an intention of the agent), then take the next appropriate action from one of its intentions. If the intention contains a sub-goal, then it will be posted to the event queue instead of preforming an action. The issue of this commonly agreed reasoning cycle is that the feasibility of the plan is only checked at the plan selection then carried out blindly by the agent as its intention without knowing if the plan is still feasible when acting according to the plan, which has been mentioned and addressed in [111, 2, 46, 47, 115], which all change the way of the reasoning cycle works to let the agent switch to either
4.2 Robust Agent Execution

In BDI-Agent system, the agent is only able to make decision about which predefined plan to be used to realise the given goal on the plan selection step in the reasoning cycle, the typical plan selection solely relies on evaluating the context of the plan. Then, once the plan is selected, there is no going back to a different plan (although

a different plan when the plan (intention) is no longer possible to follow, or allow the agent to carry out a different plan when the current intention fails.

In this chapter, we are addressing the same issue by empowering the plan selection function that decides which of the plans should the agent follow to maximise the possibility of realising the goal while minimising the possibility of intention failure. We hypothesise, by allowing the agent to predict the executional results of the plans with consideration of the worse cases of how the environment would react at each action of the plans, the most resilient plan can be selected at the plan selection, which could eventually improve the agent’s robustness in an adversarial and non-cooperative environment. We model the agent execution as a two-player adversarial game, where the agent is one player and the environment is another (where there may be other competitive agents that are considered the part of the environment). Then apply game tree search at during plan selection to decide which plans are most likely to win the game (i.e. successfully achieve the goal). In the end of this chapter, we simulate the agent’s execution with different plan libraries to show in this way, the agent is more likely to successfully complete its goal even when the environment are non-cooperative.

This chapter are structured as follows. Section 4.2 formalises the game problem of the plan selection, and Section 4.3 describes the evaluation of our approach and its outcome. At last, Section 4.4 and reviews the related works that address the similar problems, and Section 4.5 remarks and highlights our results and the future works.

4.2 Robust Agent Execution

In BDI-Agent system, the agent is only able to make decision about which predefined plan to be used to realise the given goal on the plan selection step in the reasoning cycle, the typical plan selection solely relies on evaluating the context of the plan. Then, once the plan is selected, there is no going back to a different plan (although
there is literature discussing on plan recovery which extends the traditional reasoning cycle to enable the agent to fall back and try a different plan \[2, 46, 47, 115\]). The aim of this section is to discuss the approach where the agent predicts the execution of each feasible plans with consideration of possible environmental impediments when selecting plans to maximise the probability of achieving the given goal.

Our proposal involves the use of game tree search to solve a 2-player adversarial game of perfect information to address the robust plan selection. The 2 players are the agent and the environment (could also be extended to include other agents). The game is one of the perfect information since we assume the state of the environment is equally accessible to both players. The simplest game tree search algorithm is \textit{minimax search}, which involves a maximising player (that seeks to maximise the payoff or utility) and a minimising player (that seeks to minimise the payoff – thus all game states are assessed by the same utility function). The key data structure is a \textit{game tree} where each node represents a state of the game and nodes at alternating levels represent states of the game that can be achieved via moves (actions) made by a given player. For a fully expanded game tree, the leaf nodes represent end-game states, which then are labeled with 1, 0 to represent a win (for the agent, the maximising player), and loss respectively. The minimax algorithm proceeds by propagating these values up the game tree, with a node corresponding to a state where the maximising player makes a move being labeled with the maximum of the utility values of its child nodes (and the converse for nodes where the minimising player makes a move). The intent is to obtain a payoff/utility value labeling all the child nodes of the root of the tree (the state at which a move must be made by one of the players). Once these labels are obtained, the maximising player selects only the moves that leads to the state with the highest utility (converse for the minimising player). For the most complex games (such as chess), the full game tree is too large to enumerate, and search proceeds by cutting
off the search at a fixed (parametric) depth and treating the nodes at that depth as a pseudo-leaf nodes, and an heuristic function is then used to give an estimation on the utility of the pseudo-leaf node, by which the estimated value is an approximate indicator of the likelihood that the game state will eventually lead to a win for the maximising player. Minimax search with $\alpha$-$\beta$ cutoffs involves bound propagation on payoff values to prune the search tree (see Section 2.3).

In our setting each node represents an expected state of the world in the agent’s execution (the predicted states we have discussed before in Chapter 3). The moves available to the agent player correspond to the actions in the capability library and is limited by the predefined plans, where there is only one action available to the agent withing the plan until the next sub-goal comes up, then all the first actions of the plans for the sub-goal are legal moves for the agent. Whereas the moves available to the environment player correspond to the potential impediments that can be made true by the environment as per the environment behavioural model. More interestingly, this environmental impediments could also include the actions in the capability library of another agent in a multi-agent setting. Both task post-conditions and impediments can be viewed as sentences in the underlying language $L$. Given such sentences in $e$ and a prior state $s$, similarly, the resulting state is denoted by $s \oplus e$ where $\oplus$ is the state update operator. State update operators generate possibly many non-deterministic outcomes in the general case. For example, the operator defined in Definition 3.1 is one instance of the state update operator. There are other operators defined according to the application domain, such as in game tree search literature, a state update operator takes the game state and a move and creates the next state of the game (such as in Chess or Go) or a set of possible states (such as in stochastic games like Backgammon and 2048). The non-deterministic states associated with a given move/action represent a point of departure from standard minimax search (where a given move/action leads
to a unique game state). This can be handled easily by extending the worst-case reasoning approach that underpins minimax search. Thus, if a maximising player contemplates a given move, it will pick the state with the highest payoff amongst the possibly many states that can result from that move (converse for the minimising player). Alternatively, the game tree is expanded by adding all the possible states resulted by a single move/action (this is a preferred solution in this chapter as the worst-case reasoning is a greedy solution, which only takes the locally optimal choice).

To avoid the explosion introduced by fully expanding the game tree, we limit the depth of the game tree to a predefined depth, use a heuristic evaluation function $h: S \rightarrow \mathbb{R}$ to evaluate the terminal and non-terminal states instead of $utility()$ that is described in Section 2.3. Designing a heuristic evaluation function $h()$ that is able to estimate the likelihood of a given state leading to a “win” for the maximising player is another challenge. The heuristic evaluation function we use in generating the experimental results presented in the later section is conceived with the following intuitions in mind.

- The heuristic function $h()$ should prefer wining states over other non-wining states.
- The heuristic function $h()$ should prefer non-losing states over losing states.
- It is possible that two different states yield the same heuristic value.
- For 2 given non-terminal states (not wining or not losing) $s_1$ an $s_2$,
  - If the next move is to be made by the maximising player, and there is a known move that leads to a wining state from $s_1$ but there is no such move from $s_2$, then $s_1$ should have a higher estimated utility than $s_2$.
  - If the next move is to be made by the minimising player (the environment), and there is a known impediment that leads to a losing state from $s_1$ but
there is no such impediment from $s_2$, then $s_2$ yields higher estimated utility than $s_1$.

- For the maximising player, if the state $s$ can be transferred into a known state $s'$ that eventually leads to a winning state, then the state $s$ yields higher utility than other states but not higher than the utility of $s'$.

- For the minimising player, if the state $s$ can be transferred into a known state $s'$ that eventually leads to a losing state, then the state $s$ yields lower utility than other states but not lower than the utility of $s'$.

A state $s$ has higher utility if there are indications that $s$ is “closer” to a winning state or to any known state that leads to a winning state eventually, e.g., a state in any of the predicted traces (Definition 3.3). On the other hand, MCTS does not require to design such a function.

Given a set of sentences $s$ and a background knowledge base $\mathcal{KB}$, we use $\text{Cn}_{\mathcal{KB}}(s)$ to denote the set of all logical consequences of $s \cup \mathcal{KB}$. Let the union of the goal states and the states in the predicted traces $PT$ discussed in Definition 3.3 be referred to as the set of desired states $\mathcal{S}_{\text{desired}}$. One plausible and intuitive means (but by no means the only one) of assessing the proximity of a state $s$ to desired states $\mathcal{S}_{\text{desired}}$ (denoted by $h(s)$) is

$$h(s) = \max_{\forall s' \in \mathcal{S}_{\text{desired}}} \left( \frac{|\text{Cn}(s) \cap \text{Cn}(s')|}{|\text{Cn}(s')|} \right)$$  \hspace{1cm} (4.1)

This function obtains a higher value when the cardinality of the intersection of the set of consequences of $s$ and $s' \in \mathcal{S}_{\text{desired}}$ and the cardinality of $s'$ get closer. In the experimental evaluation (Section 4.3), we compute the number of clauses in a CNF representation of $s'$ that are entailed by $s$, as one computational realisation of the expression above. Since we are able to work with grounded theories (universally qualified rules in the $\mathcal{KB}$ are replaced by their grounded instances — of which there is
a relatively small number), we use the SAT4J SAT solver [9] as our theorem prover.

The optimal plan of the plan selection by game tree search is defined as the plan that brings the highest utility value, which means that the plan is more robust when environment being adversarial.

4.3 Experimental Evaluation

This evaluation is to test whether the robustness of the agent execution can be improved by allowing agent to search for the optimal plan using game-tree search by assuming then environment always impedes the agent’s actions (this is the extreme case). The robustness of the execution is measured by comparing the rate of successfully achieving given goals using the standard agent execution with predefined plans, optimal plans selected by minimax tree search with $\alpha$-$\beta$ pruning, and optimal plans selected by MCTS. Due to the absence of existing results in semantics-based look ahead when selecting plans in an adversarial environment, the baseline performance is established using the default plan-selection function that selects plans solely based on the context. The evaluation is then designed to find out, (1) what would happen to an agent with pre-programmed behaviors and the default plan selection in an adversarial environment; (2) what would happen to an agent if semantics-based game tree search with anticipations of the environment being adversarial is used to select the plan that are less likely to fail in the future scenarios. The evaluation is run on Intel® Core™ i5-4440 with 16GB memory in Ubuntu 16 and Java SE 8.

4.3.1 Agent Programs

Two set of plan libraries are used in this evaluation, a set of 8 simple plan libraries, effects and knowledge rules and a set of complex plans, effects and knowledge rules. The simple plan libraries contains 1 to 8 plans and 0 to 4 goals and sub-goals. The
number of logic variables and the number of knowledge rules are all different in the
plan libraries, but the exact complexity of the semantics are hard to quantify. Case1
is the simplest, with only 4 actions in 1 plan, 1 or 2 assertions in each of the effects,
and 3 rules in the knowledge base. Case2 has the same number of tasks as Case1 but
in 3 plans but with one sub-goal, and slightly more complex effects and logical rules.
Case3 also has 1 plan, but more complex effect and more knowledge rules. For the
rest of plan libraries, we progressively increase the number of plans and sub-goals as
well as the complexity of effects and knowledge rules.

The second set of plan libraries have considerably greater structural and semantic
complexity relative to the first set. The number of unique actions in each plan libraries
varies from 10 to 100, with the number of plans and sub-goals varying from 1 to 15.
The effects are described by 20 to 50 propositional variables from library to library
and each action will change between 1 to 5 of these variables. The knowledge base
consists 20 to 50 rules.

4.3.2 Environment Behavior Model

Let $\mathcal{I}$ be the set of impediments that the environment is capable of making true.
For the purposes of experimental evaluation, we adopt a maximally adversarial model
of the environment. We generate the set of impediments $\mathcal{I}$ by creating a distinct
impediment from the negation of every clause in a conjunctive normal form (CNF)
representation of a goal state. Thus, given a set of agent’s current goal states (wining
states) $S_g$, the set of impediments of the current goal $\mathcal{I}_g$ is defined as:

$$\mathcal{I}_g = \{ \neg c | \forall c \in S_g, \forall s_g \in S_g \}$$
4.3. Experimental Evaluation

4.3.3 Minimax Tree Search

In this evaluation we use minimax tree search with \( \alpha-\beta \) pruning [Section 2.3.3]. Due to the complexity of the search, we limit the depth of the game tree to 5. The heuristic evaluation function [Equation 4.1] discussed in Section 4.2 is applied to the pseudo-leaf nodes at this depth.

4.3.4 Monte Carlo Tree Search

We compare the performance of minimax search with \( \alpha-\beta \) cutoffs against the most popular MCTS algorithm, UCT [Section 2.3.5] with a random play-out simulation. Every time the agent needs to select the next plan, the tree is sampled \( 5 \times |I| \) times. We first select the most promising leaf node that is not terminal state (i.e. goal state), expand it and perform random play-out at all the newly expanded nodes. The random play-out is a simulated game play, in which the environment randomly selects a move (or structurally according to a behavior model) and the agent follows the plan (and selects plans for sub-goals at random) every time until the agent’s plan completed or until the time runs out (the timeout is set to 3 seconds in this evaluation). The value 1 is returned when a goal state is reached in a random play-out, and 0 is returned if a goal state cannot be reached in the given time or after the agent’s plan completed.

4.3.5 Evaluation Design

For every agent plan library, we first find all the unique execution instances (i.e. predicted traces) and for each of the instances, we run simulation without considering environmental impediments, and with environmental impediments as the branch mark. We expect that the agent of the simulation without considering environmental impediments (that is the environment does not work against the agent’s goals) to always be able to achieve its goal. On the other hand, the agent in the simulation with environmental
impediments will have a very low rate of achieving goal to show the robust execution cannot be guaranteed with the classical plan selection function. Then we simulate agents using either Minimax tree search powered plan selection function or Monte-Carlo tree search based plan selection function in the adversarial environment (i.e. with environmental impediment after every action the agent is taken) to find out if the agent can achieve its goal in a higher rate of success or not. Because the impediment are randomly selected in every simulation, each simulation is then run many times.

Due to the differences in the plan library, we will simulate the classical plan selection the same number of times with the number of execution instances to cover all possible execution instances, in which each simulation is uniquely following one execution instances. To simulate the game-tree search based plan selection, the same number of times as the simulation is executed with the classical plan selection but not necessarily covering all possible execution instances, because some plans, according to the game-tree search, are less preferred by the plan selection functions.

### 4.3.6 Results

With the 8 simple plan libraries, we run the evaluation 10 times in total and collected the total of $460 \times 4$ simulated instances, and 460 instances for each setup. The result shows that the standard plan selection without considering environment impediments are always successfully achieving goal (Figure 4.1), which shows the plans libraries are “correct” from the design point of view. However, when considering the environment (randomly) acts against the agent, all the instances are failed to achieve the goal, which highlights in the adversarial environment, the classical plan selection is not sufficient for robustness. Moreover, when the agent utilises some form of game-tree search and considers the possible future impediments of the environment in the plan selection process, the rates of success are improved to 40.43% (using Minimax tree search) and
4.3. Experimental Evaluation

Figure 4.1: Success Rate (achieving goal) in Simple Plan Libraries

![Success Rate Bar Chart]

47.39% (using MCTS) respectively.

**Figure 4.2** and **Figure 4.3** illustrate the simulation time of using MCTS and Minimax tree search overall and by each of the agent systems. We are not comparing the time at each decision as the implementation re-purposes the existing game tree in the later plan selections that reduce the time of rebuilding a new tree for the subsequent plan selections (selecting plans for the sub-goals of the plan). In our simulation, all the actions take no time (symbolic simulation) so the overall simulation time is the total time of selecting plans plus a small overhead. It can be seen that MCTS takes considerably more time in this case possibly due to the number of random sampling required when expanding each pseudo-leaf node, in which a complete game play is
4.3. Experimental Evaluation

Figure 4.2: Simulation Time in Simple Plan Libraries

required every time so a score can be given. Every time when we simulate an action on a given state, the state update operator (see Definition 3.1) is used to calculate the new states, which is implemented using SAT4J as described in Section 3.5. Overall, the MCTS uses considerably more instances of this operator (in tree expansion and sampling) than Minimax tree search, and this overhead accumulates and affects the outcome. On the other hand, Minimax tree search only explores the game tree to the depth of 5 then gives a heuristic score to the pseudo-leaf node using Equation 4.1. This result also holds in the case-by-case bases (Figure 4.3).
4.3. Experimental Evaluation

Figure 4.3: Simulation Time by Plan Libraries in Simple Plan Libraries

![Diagram showing simulation time by plan libraries in simple plan libraries.]

With the more complex agent systems (plan libraries), we do not use MCTS-based plan selection as it takes longer in the simple agent systems compared with Minimax tree search (the longer the plan the more time is needed by the sampling), and considering the size of the plan libraries of this set, it would take much longer. It is possible to reduce the decision time of MCTS by changing the number of sampling required at the cost of suboptimal decisions and, possibly, reduce the success rate.

Figure 4.4 shows the success rate of the complex plan library where, as usual, without environmental impediments, the agent will always achieve its goal. However, with the assumption of the environment being adversarial, the agent cannot achieve any goal at all, but with Minimax Tree Search based plan selection function, the agent achieves 23% of its goals, at the cost of average between 2000 and 3000 seconds (Figure 4.5). What is interesting in Figure 4.5 is that it takes longer for the agent to
Figure 4.4: Success Rate (achieving goal) in Complex Plan Libraries

make a decision in the no-win situation (when the goal cannot be achieved).

At last, Figure 4.6 shows the time cost of each of the complex plan libraries, and we can see that the decision time varies dramatically from library to library. The explanation may be that there are so many properties of a plan library that could affect the time of the game-tree search. Here we are to name a few, such as, structurally, the number of plans, size of plans, number of sub-goals, numbers of sub-goals in plans, etc. and semantically, the complexity of the logic rules, size of the logic language etc.
4.4 Related Works

The current literature that is closely related to this chapter are plan repair and revision, which focus on the problem of how to recover the agent from failures. In Jason [17, 18], a plan failure will lead to a goal deletion event that allows programmers to write a “clean-up” plan to handle the failure. Alechina et al. [2] focus on a replanning mechanism that allows the failed plan to be revised to continue the execution when possible or try another applicable plan. Other literature is very similar, where there are systems in place to enable the agent keep trying other plans when a plan is failed [31].
4.4. Related Works

Figure 4.6: Simulation Time by Agent Systems in Complex Plan Libraries

[126] [111] [46] [47]. Dam et al. [31] took a cost-based approach that enables the selection of the best repair plan based on a notion of cost from existing plans from a set of repair plans. Felner et al. [46] [47] systematically search for the best alternative plan that is minimally revised based on the currently failing plan.

In contrast to these approaches, this chapter analyses plan failures from a different perspective, in which instead of reacting to the failure and trying to repair the failed plan, we take a proactive approach that tries to eliminate the possible future failures from happening. We also provide generalized model that represents the possible cause of failure, i.e. impediments and the environment model, which provides the bases of modelling the agent execution as a game against the world.
4.5 Conclusion

This chapter describes a novel approach of improving the robustness of BDI-agent system. Compared against the existing plan and intention revision approaches from the literature, in which techniques are designed to react to the plan and intention failures, our approach, instead, finds the plan that is most likely to succeed using game-tree search to reduce the possibility of plan failure. We model the failures in agent executions as the result of adversarial acts of environment, that is to say that if the agent execute an action, it always succeeds, and if there is a failure, it is due to the impediments of the environment. In this way, we are able to view the agent execution as a game against the environment, in which the agent utilises the set of pre-defined strategies (plans) and tries to reach goal states while the environment attacks these strategies. We show that the agent is able to pick the best plan to execute using game-tree search, consequently improve the probability of achieving goal states in a hostile environment.
Chapter 5

Semantic Merging of BDI Agent Programs

Modern software development environment is based on developers’ ability to work in parallel on the same code-base and perform concurrent changes, which potentially need to be merged back together. However, state-of-the-art merging systems follow text-based algorithms that focus only on modifications to text but completely ignore the semantics of the code written, for example, three-way merge (used by diff3 Unix utility), recursive three-way merge (used by Git revision control system), Weave merge (used by BitKeeper and GNU Bazaar revision control system). This limitation significantly restricts developers’ ability to perform and merge concurrent changes. In this chapter, we propose a merging technique that fully understands the programming language structure of typical BDI-agent systems. In addition, our approach effectively captures the semantics of an agent system using the notion of semantic effects of goals, plans and actions constituting the agent system.
5.1 Introduction

Engineering large, complex software systems is inherently a collaborative process since it requires the participation of teams of people who may work on the same product independently and concurrently (creating different versions of it). As a result, merging is a critical functionality in existing versioning systems which support the optimistic versioning process that enables different developers to work concurrently on the same set of software artifacts (e.g. source code) rather than pessimistically locking each artifact when it is changed by one developer. However, software merging remains a highly challenging and complicated process since merging should heavily depend on the syntax and semantics of the software artifacts [81]. State-of-the-art versioning systems (e.g. CVS, Subversion or Git) are usually based on textual merging techniques. Since any software program (including agent programs) can be seen as a piece of text, text-based merge tools have been dominantly used for merging software code. This flexibility however comes with a cost in which text-based merge tools do not take the specific syntax, structure and semantics of agent programs into account and thus the merging may often result in unnecessary conflicts or a merged version which has syntax errors and inconsistent semantic behavior.

Since the 1980s, intelligent agent technology has attracted an increasing amount of interest from the research and business communities, and the practical utility of agents has been demonstrated in a wide range of domains such as weather alerting, business process management, holonic manufacturing, e-commerce, and information management. This number continues to increase since there are compelling reasons to use intelligent agent technology. However, to the best of our knowledge, there has been no work on merging versions of an agent program. If we are to be successful in the development of large-scale agent systems which requires the participation of teams of people who may work on the same product independently and concurrently, the
research community must provide solutions and insights that will improve the practice of merging versions of agent software.

The main purpose of this chapter is to contribute towards filling that gap. We propose a merging technique specifically for Belief-Desire-Intention (BDI) agent systems. Our approach captures the essential semantics of a BDI-agent system by computing the cumulative effects of plan execution from the immediate effects of the actions constituting the plan. We then merge the semantic effects of the revisions, and use them to establish a merged version. In the remaining of the chapter, we will describe an example to illustrate the limitations of existing text-based merging approach, and present in detail our approach and how it overcomes those issues.

5.2 Illustrative example

Most of today’s version controlling systems uses text-based merging techniques which consider software programs (regardless of the programming language which they are written in) merely as text files. The most common approach is to use line-based merging where lines of text are considered as indivisible units [81]. Line-based merging however cannot handle two concurrent modifications to the same line very well, which will be shown in the following scenario.

![Figure 5.1: An example of classical, text-based merging (unnecessary conflicts)](image)

Figure 5.1 illustrates an example of two developers, Alice and Bob, concurrently work on the same agent program written in AgentSpeak(L) [100], a well-known, abstract BDI-agent programming language. BDI agents’ behavior is mostly determined
in terms of their plans to handle events or achieve goals. Each plan is typically of the form \( G : [C] \leftarrow P \), meaning that the plan is an applicable plan for achieving goal \( G \) when context condition \( C \) is believed true. Plan \( P \) typically contains a sequence of actions that are meant to be directly executed in the world (e.g. \( \text{get(umbrella)} \)) in plan P1 in Figure 5.1 or sub-goals (written as \( !G \), e.g. \( !\text{waterproof} \)) in plan P11 to be resolved by further plans. Both Alice and Bob check out the same piece of code (the common ancestor), which, in this example, is a plan (P1) to achieve goal \( \text{leave(home)} \), and make different changes to it. Alice replaces the first action of the plan with a sub-goal \( \text{waterproof} \) and creates two plans (P12 and P13) to resolve the sub-goal. In the meanwhile, being unaware of Alice’s changes, Bob replaces action \( \text{close(window)} \) with a sub-goal \( \text{safeguard(home)} \) and creates a plan (P22) to resolve it. When both developers check in their own revision, existing versioning systems (which mostly rely on text-based merging) would detect unnecessary conflicts since parallel modifications has made to the same lines of code in the common ancestor. In general, text-based merging fails in these scenarios since they are heavily dependent on the position of the texts being modified and they do not consider any syntactic or semantic information in the agent code.

5.3 Semantic effects

A BDI-agent program is built around a plan library, a collection of pre-defined hierarchical plans indexed by goals. As a result, the semantics of a BDI-agent program is mostly determined by the semantics of its plans, which can essentially be captured by the semantic effects achieved by the plans. Such effects can be expressed in terms of declarative goals that the plans are meant to achieve. However, mainly due to practical concerns, goals in BDI-agent programming languages are mostly procedural where a goal is a set of tasks or processes that are to be completely carried out.
5.3. Semantic effects

A description of effects achieved by a plan has to be explicitly established from the effects of its constituting steps in a context-sensitive manner. We note that such a description will necessarily be non-deterministic, i.e., there might be alternative effects achieved, which is due to the following reasons. First, there might be different paths in plan execution since there might be multiple ways of achieving a (sub-)goal. Second, the effects of certain plan steps might “undo” the effects of prior process steps. This is often described as the belief update or knowledge update problem — multiple alternative means of resolving the inconsistencies generated by the “undoing” of effects is another source of non-determinism.

Each action has a precondition under which an action can be successfully executed and its effect (or postcondition) on the environment. The semantic effect of action $a$, denoted as $e(a)$, is a conjunctive set of belief literals since the action’s effect on the environment may eventually be perceived by the agent. The effect of action $\text{get(umbrella)}$ in the example in the previous section is the set $\{\text{on(umbrella,hand)}\}$, and the effect of action $\text{set(alarm,true)}$ is $\{\text{alarm(on)}\}$. Many agent programming languages (e.g. 3APL [37]) require an explicit specification of actions in terms of both preconditions and effects. However, our work only focuses on leveraging action effects to establish semantical representations of agent systems and use them for merging. A BDI agent also has a belief base $B$ which encodes what the agent believes about the world. An agent’s belief base may also contain rules, which allows for new knowledge to be deduced from existing knowledge. For simplicity, in this chapter we assume that the context condition is expressed as a conjunctive set of belief literals, which is similar to a semantic effect.

The effect specification of actions allows us to determine, at design time, the (cumulative) effects of plan execution. We now define a number of basic definitions that are used in the procedure for computing the cumulative effects.
5.3. Semantic effects

Definition 5.1. For two effects $e_1$ and $e_2$, and the belief base $\mathcal{B}$, if $e_1 \not\models \bot$ and $e_2 \not\models \bot$, then the cumulative effects $\text{acc}(e_1, e_2)$ (accumulating $e_2$ onto $e_1$) is defined as:

$$\text{acc}(e_1, e_2) = \{e_2 \cup e \mid e \subseteq e_1 \land e \cup e_2 \cup \mathcal{B} \not\models \bot \land$$

if there exists $e'$ such that $e \subset e' \subseteq e_1$,

then $e' \cup e_2 \cup \mathcal{B} \models \bot\}$$

We note that the result of $\text{acc}()$ on a pair of effects is a disjunctive set of effects, each of which represents a distinct way in which potential inconsistencies between the effects to be accumulated are resolved. For example, if $e_1 = \{m, n\}$ and $e_2 = \{x, y\}$ and there is a rule $m \land n \rightarrow \neg y$ in the belief base $\mathcal{B}$, then $\text{acc}(e_1, e_2) = \{\{m, x, y\}, \{n, x, y\}\}$.

Definition 5.2. The cumulative effects of the two disjunctive sets of effects $ES_1$ and $ES_2$ are defined as:

$$ES_1 \oplus ES_2 = \bigcup_{es_i \in ES_1, es_j \in ES_2} \text{acc}(es_i, es_j)$$

The operator $\oplus$ performs the pair-wise effect accumulation $\text{acc}()$ on every pair of $(es_i, es_j) \in ES_1 \times ES_2$ to form a new disjunctive set of effects. For example, if $ES_1 = \{e_1\}$ and $ES_2 = \{e_2\}$, then $ES_1 \oplus ES_2 = \{\{m, x, y\}, \{n, x, y\}\}$. The cumulative effects of a plan are represented as a disjunctive set of effects where each effect (also called an effect scenario) corresponds to a particular path of the plan execution (i.e. a particular scenario). In order to compute a plan’s cumulative effects, we need to simulate the plan execution, particularly the context-sensitive sub-goal expansion and plan selection. For example, assume that a plan has executed a number of actions, which gives cumulative effects $ES = \{\{m, x, y\}, \{n, x, y\}\}$, and is about to expand a sub-goal $g$, which can be resolved by plan $P$ under the context condition $c = \{\neg n\}$. Since only the effect scenario $\{m, x, y\}$ is consistent with the context condition, the
cumulative effects just before plan \( P \) is executed would be \( ES \odot c = \{\{m, x, y, \neg n\}\} \).

The operator \( \odot \) which eliminates effects that are inconsistent with a plan’s context condition is defined as below.

**Definition 5.3.** The effect elimination of \( ES \) with regard to context condition \( c \) (which is a set of literals and can be considered as an effect) is defined as:

\[
ES \odot c = \bigcup_{es_i \in ES} \{es_i \cup c\}, \text{where } es_i \cup c \neq \perp
\]

**Algorithm 5.1** Computing semantic effects for a given node in the goal-plan tree (initial call is \( \text{SemanticEffect}(\text{root}, \emptyset) \))

```
1: procedure \text{SemanticEffect}(n, ES)
2: if \( n \) is a plan node then
3: \quad ES \leftarrow ES \odot \text{context}(n)
4: if \( ES \neq \emptyset \) then
5: \quad for each \( \text{child } n' \) of \( n \) from left to right do
6: \quad \quad ES \leftarrow \text{SemanticEffect}(n', ES)
7: \quad end for
8: end if
9: else if \( n \) is an action node then
10: \quad ES \leftarrow ES \oplus \{e(n)\}
11: else if \( n \) is a goal node then
12: \quad ES' \leftarrow \emptyset
13: \quad for each \( \text{child } n' \) of \( n \) do
14: \quad \quad ES' \leftarrow ES' \cup \text{SemanticEffect}(n', ES)
15: \quad end for
16: \quad ES \leftarrow ES'
17: end if
18: return \( ES \)
19: end procedure
```

A BDI-agent program can be represented as a number of goal-plan trees where each goal has as children the plans that are relevant to it, and each plan has as children its actions and/or sub-goals. The goal-plan tree is an “and-or” tree: each goal is realized by one of its relevant plans (“or”) and each plan needs all of its actions to be executed and its sub-goals to be achieved (“and”). Therefore, computing semantic effects for
5.3. Semantic effects

Algorithm 5.1 describes how we traverse a goal-plan tree to compute the cumulative effects for a particular node in the tree. The cumulative effects are stored in the set $ES$ which are accumulated as we visit each node of the tree in the depth-first search manner. If the node $n$ is a plan node (lines 2–8), we obtain the context condition of the plan (i.e. $context(n)$) and apply the effect elimination operator $\ominus$ onto the set of cumulative effects $ES$. If the outcome is not an empty set, reflecting there exists at least a scenario in which the plan is applicable, we visit each node of the plan’s children (i.e. which is either an action or sub-goal node) to accumulate its semantic effects. If the node is an action node (lines 9–10), we simply use the operator $\oplus$ to accumulate the action’s effect. Finally, if the node is a goal node (lines 11–16), we visit each node of the goal’s children (i.e. which are plan nodes), compute its semantic effects and add them into the set of cumulative effects.

Figure 5.2 shows the goal-plan trees for the agent program and its two revisions.
5.4 Semantic merging

We now outline the process of merging two BDI-agent programs using semantic effects. Our approach follows the popular three-way merging [81] which requires a common ancestor program as the base and two revisions.

1. Compute the semantic effects of the (common) ancestor agent program and its two revisions, which gives us three set of semantic effects $ES_{base}$, $ES_1$ and $ES_2$.

2. Use those semantic effects to identify the essential differences between the ancestor and the revisions. The difference of two semantic effects $ES_i$ and $ES_j$ is defined as $\delta(ES_i, ES_j) = ES_j \setminus ES_i$, which contains the semantic effects that are in $ES_j$ but not in $ES_i$. Note that $\delta$ is asymmetric, that is $\delta(ES_i, ES_j) \neq \delta(ES_j, ES_i)$.

3. Compute the merged semantic effects $ES_{merge} = (ES_{base} \cup \Delta^+) \setminus \Delta^-$ where $\Delta^+ = \delta(ES_{base}, ES_1) \cup \delta(ES_{base}, ES_2)$ and $\Delta^- = \delta(ES_1, ES_{base}) \cup \delta(ES_2, ES_{base})$. Intuitively, $\Delta^+$ is the effects newly created in the revisions, and $\Delta^-$ is the effects that are removed in the revisions. The merged version therefore has the behaviors in the base program that are preserved in both revisions and the new behaviors coded in the revisions. Note that the order of merging is not important since merging is done here in terms of set union.

4. Construct the merged program from the merged semantic effects.
We now illustrate how a merged version can be obtained in our running example by following the above steps. In the previous section, we have computed the semantic effects \( ES_{\text{base}}, ES_1 \) and \( ES_2 \). Now, we compute the semantic differences \( \delta(ES_1, ES_{\text{base}}) \) and \( \delta(ES_2, ES_{\text{base}}) \) which are the same, and equal to \( \{\{\text{on(umbrella, hand), window(closed)}\}\} \).

The set of merged semantic effects are then computed as follows.

\[
ES_{\text{merge}} = (ES_{\text{base}} \cup \Delta^+) \setminus \Delta^- = (ES_1 \cup ES_2) \setminus \Delta^-
\]

\[
= \{\{\text{raining, on(umbrella, hand), window(closed)}\},
\{\neg\text{raining, window(closed)}\},
\{\text{on(umbrella, hand), } \neg\text{safe(neighbourhood), window(closed), alarm(on)}\}\}
\]

The final step of the merging process involves reconstructing a program from the merged semantic effects. This involves reconstituting a feasible goal-plan tree such that the semantic effects of this tree, denoting \( ES_r \), satisfy the following condition:

\[
\forall es \in ES_{\text{merge}}, \exists es' \in ES_r, es' \cup \mathcal{B} \models es.
\]

Figure 5.3 shows a feasible goal-plan tree whose semantic effects

\[
ES_r = \{\{\text{raining, on(umbrella, hand), } \neg\text{safe(neighbourhood), window(closed), alarm(on)}\}\},
\]
{¬raining, ¬safe(neighbourhood), window(closed), alarm(on)}

satisfy the above condition. If we cannot reconstitute any feasible goal-plan tree from
the merged semantic effects, there must be conflicting changes made in the revisions
that need to resolved. Our future work involves identifying those conflicting changes
using the semantic effects. We also note that there may be more than one feasible goal-
plan tree and they should be presented to the software engineers for selection. Future
work would involve developing a search algorithm to find all of those feasible goal plan
trees. Computing the differences and the merged semantic effects are essentially set
operations as defined by $ES_{merge}$ (where only set union and set differences are used),
which grow linearly with the size of semantic effects of the program, as both set union
and set difference have the complexity of $\mathcal{O}(n)$. The number of ways to resolve conflicts
is constrained within the changes made in the revisions to be merged. Therefore, we
expect the approach does scale to standard programs.

5.5 Conclusions

Although there have been some recent work on providing support for the maintenance
and evolution of agent systems (e.g. [30, 31]), there is still a big gap in addressing the
versioning and merging issues of agent systems. Text-based merging is the dominant
approach used in most today’s versioning systems. Text-based merging is the dominant
approach used in most today’s versioning systems. Due to its limitations, a number
of approaches (e.g. [10] or see [81] for a comprehensive survey) have been proposed to
merge classical programs (e.g. procedural or object-oriented) in a semantical manner.
Recently, there have been some work (e.g. the recently released commercial SemanticMerge
software\footnote{http://www.semanticmerge.com/} on refactoring-aware merge (which preserves the semantics), but they are
limited to object-oriented programing languages. Such approaches are not readily
5.5. Conclusions

applied to a BDI-agent program due to its distinct syntax, structure and semantics. In addition, traditional approaches which rely on program slicing or dependency graph do not really capture the true semantics of agent programs. In particular, they cannot capture the semantic effects of agent actions. We have proposed a novel approach that enables merging versions of a BDI-agent program semantically. Since the approach is built upon an abstract BDI notation, it can generally be extended to any BDI-agent programming languages. Future work involves further refining and implementing our merge approach.
Part II

Robust Process Execution in Adversarial Environments
Chapter 6

Semantic Conformance and Compensation Computation

Socio-technical processes are becoming increasingly important, with the growing recognition of the computational limits of full automation, the growth in popularity of crowd sourcing, the complexity and openness of modern organizations etc. A key challenge in managing socio-technical processes is dealing with the flexible, and sometimes dynamic, nature of the execution of human-mediated tasks. It is well-recognized that human execution does not always conform to predetermined coordination models, and is often error-prone. This chapter addresses the problem of semantically monitoring the execution of socio-technical processes to check for non-conformance, and the problem of recovering from (or compensating for) non-conformance. We propose a semantic solution to the problem, by leveraging semantically annotated process models to detect non-conformance, and using the same semantic annotations to identify compensatory human-mediated tasks.
6.1 Introduction

Socio-technical processes, which are executed by synergistic combinations of humans and technological components, have a long history, but have assumed new significance with the current interest in issues such as crowd-sourcing, human computation and gamification. They have also become important as a consequence of the introduction of process automation in settings where human-mediated functionality is critical and indispensable (such as clinical process management, military command and control, or air traffic management). An important aspect of socio-technical processes is that the human-mediated components are fallible, while the machine-mediated components are generally not (although there are critical exceptions). One way in which such fallibility might be manifested is via structural non-conformance, where activities are overlooked or executed in the wrong order, or where the wrong activities are executed. There is a mature body of work focusing on structural non-conformance (see [107] for a representative reference). Our focus is on the harder problem of semantic non-conformance, where we are interested in managing situations where the execution of a process might be structurally correct (the right activities are executed in the right order), but the effects achieved do not conform to what is required by design, potentially due to human errors. For instance, a clinical process might require the administration of an anti-hypertensive medication. The correct execution of this task would require that a nurse should deliver the medication to the patient in question and depart only when the patient has ingested the medication. A semantically non-conformant execution might occur if the nurse delivers the medication to the patient, but does not stay around to confirm that the patient has actually taken it (and the patient happens to not take the medication). In a hospital with a process-aware information system, the nurse might then confirm to the process engine that this task has been completed, leading to a situation where no structural non-conformance
would be detected. The fact that this process instance is semantically non-conformant can only be determined by checking the effects of the process to ensure that what is expected is actually obtained. Thus, in our example, a blood pressure check later in the day might reveal elevated readings, when the expected readings are lower. This chapter addresses the problem of semantic monitoring of socio-technical processes, by leveraging process designs that have been annotated with the expected effects at each point. Semantic non-conformance is flagged in settings where the observed effects deviate from the expected ones.

The human-mediated components of socio-technical processes also offer greater flexibility in “fixing” semantic non-conformance via the introduction of human-mediated activities constructed on the fly (generating new machine-mediated functionality, such as a new web service, can often take too long to be able to correct errors in an executing process instance). Thus, in our example, the semantic non-conformance detected via the blood pressure check can be fixed by having the nurse correctly administer the medication as soon as possible. Once this is done, the clinical process instance involving this patient would be restored to a semantically conformant state. This chapter also addresses the problem of computing the best “fixes” of this kind, which we shall refer to as compensations. The problem is non-trivial. While the common-sense compensation in our running example might be to administer the anti-hypertensive medication as soon as the elevated blood pressure is detected, this might not be possible because of potential interactions between the anti-hypertensive medication and a more recently administered drug. We might thus need to search through the space of possible process re-designs to identify one where the earliest compensation is possible.

In the rest of this chapter, we show how semantic annotation of process designs can be leveraged for a machinery for monitoring process execution based on effects (Section 6.2). We then formalize the notion of compensation and discuss a class of
techniques that can be used to compute “optimal” compensations to deal with semantic non-conformance (Section 6.3). We describe the implementation and empirical evaluation of one of these techniques, with promising results (Section 6.4).

6.2 Semantic Process Monitoring

There is a large body of work that explores the use of semantic annotation of business process designs [39, 40, 114, 48, 54, 61, 64, 122, 135, 45], or mining formal semantics of processes from data [110, 109, 106]. A large body of work also addresses the problem of semantic annotation of web services in a similar fashion [104, 82, 84, 121]. Common to all of these approaches is the specification of post-conditions, which is what we primarily leverage in defining inter-process relationships. For our purposes, two aspects of the post-conditions (or effects) are important. First, post-conditions should be sensitive to process context, i.e., the post-conditions of a task at a certain point in a process design should reflect not just the effects achieved by executing that task but also the accumulated effects of the prior tasks in the process design that have been executed. Second, non-determinism must be accommodated in relation to post-conditions.

A number of the process annotation approaches referred to above achieve contextualization of post-conditions by using a device originally used in AI planning — add-lists and delete-lists of effects. Others, such as [61] and [135], use a state update operator derived from the literature on reasoning about action. We adopt this approach. The need for permitting non-determinism in effects stems from two observations. First, in any process with XOR-branching, one might arrive at a given task via multiple paths, and the contextualized post-conditions achieved must be contingent on the path taken. Since this analysis is done at design time, we need to admit non-deterministic effects since the specific path taken can only be determined at run-
time. Second, many state update operators generate non-deterministic outcomes, since inconsistencies (that commonly appear in state update) can be resolved in multiple different ways. Our approach assumes that each task/activity is annotated with post-conditions (in the implementation presented later, we shall assume them to be unique, as much of the literature does, but this can be easily generalized to admit non-deterministic post-conditions), which are contextualized via a process of effect accumulation. We shall assume that all tasks (and their post-conditions) are drawn from an enterprise capability library. In this approach, we are able to answer, for any point in a process design, the following question: what will have happened if the process executes up to this point? The answer is a mutually exclusive set of effect scenarios, any one of which might describe the actual state of affairs at that point in the execution of the process design. Additional detail on the specific effect annotation and accumulation machinery used in the implementation can be found in Section 6.4.

We note that when a process is in a state that is (partially) characterized by an effect scenario, the execution of the next task in the model, or the occurrence of the next event, can lead to a very specific set of effect scenarios, determined by the state update operator being used. In effect, the process model determines a transition system, which determines how the partial state description contained in an effect scenario evolves as a consequence of the execution/occurrence of the next task (event) specified in the model. We assign each effect scenario appearing in a semantically annotated process model a unique ID (thus if the same partial description applies to a process at different points in its design, it would be assigned a distinct ID at each distinct point). We can thus refer to the predecessors (the effect scenarios that can lead to the current scenario via a single state update determined by the next task/event) and successors (the scenarios that can be obtained from the current scenario via a single state update determined by the next task/event) of each effect scenario with
6.2. Semantic Process Monitoring

respect to the transition system implicitly defined by the process design. There are works that have been done on obtaining such effect scenario, such as in [64] and [54], which also suggest that due to different paths at gateways could be taken before a task in a process model, and/or other reasons, there could be multiple effect scenarios associated with the task.

Definition 6.1. A \textit{semantically annotated process model} $P$ is a process model in which each activity or event is associated with a set of effect scenarios. Each effect scenario $es$ is a $4$-tuple $\langle ID, S, Pre, Succ \rangle$, where $S$ is a set of sentences in the background language, $ID$ is a unique ID for each effect scenario, $Pre$ is a set of IDs of effect scenarios that can be valid predecessors in $P$ of the current effect scenario, while $Succ$ is a set of IDs of effect scenarios that can be valid successors in $P$ of the current effect scenario.

A semantically annotated process model is associated with a set of normative traces, each providing a semantic account of one possible way in which the process might be executed.

Definition 6.2. A \textit{normative trace} $nt$ is a sequence

$$\langle \tau_1, es_1, \tau_2, \ldots es_{n-1}, \tau_n, es_n \rangle$$

where

- $es_1, \ldots, es_n$ are effect scenarios, and for each $es_i = \langle ID_i, S_i, Pre_i, Succ_i \rangle$, $i \in [2..n]$, it is always the case that $ID_{i-1} \in Pre_i$ and $ID_i \in Succ_{i-1}$;

- $es_n = \langle ID_n, S_n, Pre_n, \emptyset \rangle$ is the final effect scenario, normally associated with the end event of the process;
6.2. Semantic Process Monitoring

- \( es_1 = \langle ID_1, S_1, \emptyset, Succ_1 \rangle \) is the initial effect scenario, normally associated with the start event of the process;

- Each of \( \tau_1, \ldots, \tau_n \) is either an event or an activity in the process.

We shall refer to the sequence \( \langle \tau_1, \tau_2, \ldots, \tau_n \rangle \) as the identity of the trace \( nt \).

To simplify of the presentation later on, the \( es \) in the trace, from now, refers to \( S \) in the 4-tuple \( \langle ID, S, Pre, Succ \rangle \) because \( ID, Pre, \) and \( Succ \) are meta-information used only to construct normative traces.

**Definition 6.3.** A **semantic execution trace** of a process \( P \) is a sequence

\[ \langle \tau_1, o_1, \tau_2, o_2, \ldots, \tau_m, o_m \rangle \]

where each \( \tau_i \) is either a task or an event, and each \( o_i \) is a set of sentences in the background language that we shall refer to as an observation that describes (possibly incompletely) the state of the process context after each task or event. We shall refer to the sequence \( \langle \tau_1, \tau_2, \ldots, \tau_m \rangle \) as the identity of the execution trace.

Note that we do not require each \( \tau_i \) to belong to the process design \( P \) to allow the possibility of actual executions being erroneous. We will, on occasion, refer to a semantic execution trace, simply as an execution trace.

**Definition 6.4.** An execution trace,

\[ et = \langle \tau_1, o_1, \ldots, \tau_m, o_m \rangle \]

is said to be **non-conformant** with respect to a semantically annotated process \( P \) if and only if any of the followings hold:
6.3. Semantic Compensation

1. There exists an \( o_i \) in \( et \) such that for all normative traces \( nt' = \langle \tau'_1, es_1, \ldots, \tau'_i, es_i, \ldots \rangle \) for which the identity of \( \langle \tau_1, o_1, \ldots, \tau_i, o_i \rangle \) is a prefix of its identity and \( o_j \models es_j \) for each \( j = 1, \ldots, i - 1 \), \( o_i \not\models es_i \) (we shall refer to this as weak semantic non-conformance).

2. If we replace non-entailment with inconsistency in condition (1) above, i.e., \( o_i \cup es_i \models \bot \), we obtain strong semantic non-conformance. In each case, we shall refer to \( \tau_i \) as the violation point in the process.

We only deal with semantic non-conformance in structurally compliant process instances. In other words, we assume that the identity of every semantic execution trace of interest equals the identity of some normative trace of the process.

6.3 Semantic Compensation

In this section, we formalize the notion of compensation and outline some strategies for computing these. In the following, we will view process instances as semantic execution traces. We will assume that each process is associated with a goal assertion \( g \).

Definition 6.5. A process instance \( et = \langle \tau_1, o_1, \ldots, \tau_m, o_m \rangle \) will be referred to as a semantically compensated instance of a (semantically annotated) process \( P \) if there exist \( \tau_i \) and \( \tau_j \) in \( et \), with \( i < j \), such that \( \tau_i \) is a violation point, and there exists a normative trace \( nt = \langle \tau_1, es_1, \tau_2, es_2, \ldots es_{h-1}, \tau_h, es_h, \ldots, \tau_n, es_n \rangle \) of \( P \) with an identity for which \( \langle \tau_1, \ldots, \tau_{j-1} \rangle \) serves as a prefix, such that \( o_k \models es_l \) for \( k = j, \ldots, m \) and \( l = h, \ldots, n \). As well, it must be the case that \( o_m \models g \). We shall refer to \( \tau_j \) as the compensation point. The compensation point must be a task and not an event.

Definition 6.6. Given a semantically compensated process instance \( et = \langle \tau_1, o_1, \ldots, \tau_m, o_m \rangle \) of \( P \) with a compensation point \( \tau_j \), a compensation is
6.3. Semantic Compensation

A process design $P'$ for which the completion of $\tau_{j-1}$ serves as the start event and $(\tau_j, o_j, \ldots, \tau_m, o_m)$ is a valid normative trace. Every normative trace associated with $P'$ must end in an effect scenario $es$ such that $es \models g$, where $g$ is the goal associated with the original process $P$.

This definition of compensation is fairly general. More specifically, we are interested in optimal compensations, driven by the following intuitions. We prefer earlier compensations. In other words, we aim to ensure that as few system states as possible deviate for the normative process design (noting that a later compensation will necessarily mean that there would be more states between the violation point and the compensation point). We also prefer to minimize deviation of the overall semantically compensated process instance from the semantic “intent” of the original process design. These preferences can lead to competing pulls. We might in some situations be able to introduce an earlier compensation, but the compensation, while ensuring conformance from subsequent steps (assuming no other steps deviate), might lead to greater changes in the system states than a potential later compensation.

Computing a compensation thus requires that we identify a process design which permits us to complete the currently executing process instance from the compensation point onwards in a manner that gives us a complete semantic execution trace that is as close as possible to the normative trace that would have been executed has there been no violation. The occurrence of a violation entails that we are only able to identify a prefix of this normative trace (the part that is actually executed prior to the violation). Given that multiple normative traces associated with the process design may share that prefix, we do not actually know which of these we would have actually executed had there been no violation. One way to compute the compensation is to identify that process design (or designs) which would minimize deviation from this set of normative traces (by picking one that minimizes the distance to either the closest, or the farthest
6.4 Implementation and Evaluation

In this section, we outline one specific implementation of the general framework for semantic process monitoring and compensation described above and present some preliminary empirical results. We note that the general framework could be instantiated in multiple ways (indeed the space of alternative design decisions is very large) and we do not suggest that this particular implementation is to be preferred to other possible ones (such claims can only be made after a series of substantive comparative studies). However, this particular implementation provides an adequate basis for making a preliminary determination of whether this approach is practical.

We use a machinery for semantic annotation of business process designs represented in BPMN. We omit details here for brevity but these can be found in [64]. It uses a syntactic state update operator based on the Possible Worlds Approach (PWA) [56]. The choice of this particular operator is mainly a matter of convenience (and adequate for assessing feasibility), while other operators, such as one based on the Possible Models Approach (leveraged by [135]) could also be used. We assume that a process model, semantically annotated using this machinery, is provided as input.

To measure the structural distance between a pair of sequences of activities/events, we use the Levenshtein distance \([77] lev(a, b)\) where \(a = \langle a_1, \ldots, a_n \rangle\) and \(b = \langle b_1, \ldots, b_m \rangle\).

For semantic distances, we define a simple distance function \(\phi(es, a)\) where \(es\) is
an effect scenario and $o$ is a set of observations. We note that many, potentially more sophisticated schemes for measuring semantic distance exist, but this is adequate for preliminary analysis. In the following, $V_{\text{strong}}$ computes the number of assertions in an effect scenario that contribute to strong semantic non-conformance (as in Definition 6.3), while $V_{\text{weak}}$ computes the number of assertions that contribute to weak semantic non-conformance. We leverage a background knowledge base $KB$ that contains, amongst others, domain and compliance constraints.

\[
V_{\text{strong}} = \{ e | e \in es, o \cup KB \models \neg e \}
\]
\[
V_{\text{weak}} = \{ e | e \in es, o \cup KB \not\models e, e \not\in V_{\text{strong}} \}
\]
\[
\phi(es, o) = w_{\text{strong}} \times |V_{\text{strong}}| + w_{\text{weak}} \times |V_{\text{weak}}|
\]

where, $w_{\text{strong}}$ and $w_{\text{weak}}$ are weights. If all observations reveal complete state descriptions, then weak violations do not apply. We can focus attention solely on strong or weak violations by appropriately setting the corresponding weights.

We measure the distance between a normative trace $nt = \langle a_1, es_1, \ldots, a_n, es_n \rangle$ and a semantic execution trace $et = \langle b_1, o_1, \ldots, b_m, o_m \rangle$ using the following function:

\[
J(nt, et) = \sum_{i=1}^{n} \sum_{j=1}^{m} \min (w_1 \times \phi(es_i, o_j) + w_2 \times \text{lev}(\langle a_1, \ldots, a_i \rangle, \langle b_1, \ldots, b_j \rangle)) \tag{6.1}
\]

where $w_1$ and $w_2$ are the weights for each distance. For example, given a normative trace $nt$ and an execution trace $et$,

\[
nt = \langle \tau_1, es_1, \tau_2, es_2, \tau_3, es_3 \rangle
\]
\[
et = \langle \tau_1, o_1, \tau_3, o_2 \rangle
\]
6.4. Implementation and Evaluation

where,

\[ es_1 = \{ p \} \]
\[ es_2 = \{ p, q \} \]
\[ es_3 = \{ q, \neg r \} \]
\[ o_1 = \emptyset \]
\[ o_2 = \{ p, r \} \]

and we assume there is no logic rule in the knowledge base and \( w_1 = w_2 = w_{\text{strong}} = w_{\text{weak}} = 1 \) to simplify the calculation. Therefore, the following 2 matrices can be created

\[
\Phi = \begin{pmatrix}
\phi(es_1, o_1) & \phi(es_1, o_2) \\
\phi(es_2, o_1) & \phi(es_2, o_2) \\
\phi(es_3, o_1) & \phi(es_3, o_2)
\end{pmatrix}
\]

\[
= \begin{pmatrix}
w_{\text{strong}} \times |\emptyset| + w_{\text{weak}} \times |\{ p \}| & w_{\text{strong}} \times |\emptyset| + w_{\text{weak}} \times |\emptyset| \\
w_{\text{strong}} \times |\emptyset| + w_{\text{weak}} \times |\{ p, q \}| & w_{\text{strong}} \times |\emptyset| + w_{\text{weak}} \times |\{ q \}| \\
w_{\text{strong}} \times |\emptyset| + w_{\text{weak}} \times |\{ q, \neg r \}| & w_{\text{strong}} \times |\{ \neg r \}| + w_{\text{weak}} \times |\{ q \}|
\end{pmatrix}
\]

\[
= \begin{pmatrix}
1 & 0 \\
2 & 1 \\
2 & 2
\end{pmatrix}
\]

\[
Lev = \begin{pmatrix}
\text{lev}(\langle \tau_1 \rangle, \langle \tau_1 \rangle) & \text{lev}(\langle \tau_1 \rangle, \langle \tau_1, \tau_3 \rangle) \\
\text{lev}(\langle \tau_1, \tau_2 \rangle, \langle \tau_1 \rangle) & \text{lev}(\langle \tau_1, \tau_2 \rangle, \langle \tau_1, \tau_3 \rangle) \\
\text{lev}(\langle \tau_1, \tau_2, \tau_3 \rangle, \langle \tau_1 \rangle) & \text{lev}(\langle \tau_1, \tau_2, \tau_3 \rangle, \langle \tau_1, \tau_3 \rangle)
\end{pmatrix}
\]
Let $D = w_1 \times \Phi + w_2 \times Lev$, the distance of the two traces is

$$J(nt, et) = \sum_{i=1}^{3} \min_{j=1}^{2} D_{ij} = \sum_{i=1}^{3} \min_{j=1}^{2} (w_1 \times \Phi + w_2 \times Lev)_{ij}$$

$$= \sum_{i=1}^{3} \min_{j=1}^{2} \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{pmatrix}_{ij}$$

$$= \sum_{i=1}^{3} \min_{j=1}^{2} \begin{pmatrix} 1 & 1 \\ 3 & 2 \\ 4 & 3 \end{pmatrix}_{ij} = \sum_{i=1}^{3} 2 = 6$$

Our prototype takes a semantically annotated business process and a capability library as inputs, then generates a set of all normative traces. We simulate a normative execution trace and randomly insert a violation in it. Once a violation is detected, the compensation computation machinery initiates a search for a sequence of activities from the capability library that can constitute a valid completion of the current partially complete process instance and that guarantees that it terminates in a goal-satisfying state. The prototype performs an exhaustive constructive search. Every candidate partial extension of the current process instance is evaluated for compliance with the $KB$. In the event of non-compliance, the search backtracks and evaluates an alternative extension. Our evaluation uses a propositional language for representing effects and the $KB$. Effect accumulation, goal satisfaction and compliance checking require the use of a theorem prover — in our prototype, the SAT4J SAT solver (modified to generate all maximal consistent subsets) is used for this purpose. We
apply the effect accumulation machinery to generate a semantic trace from each of the valid task sequences identified by the search procedure. This gives us a set of semantically compensated process instances which are then ranked according to the nearest distance to a valid normative trace (i.e., for each process instance, we compute the shortest distance to any valid normative trace, and the instance with shortest distance amongst all appears at the top of the ranking, and so on). We limit each task in the capability library to be used only once in a semantically compensated process instance.

In the evaluation, we manually design 5 distinct semantically annotated process models with variations in the number of activities, gateways (we only use XOR gateways), complexity of the knowledge base and effect scenarios etc (note that these cannot be randomly generated). These dimensions of the 5 process model are summarized in Table 6.1. We then identify the quality of solutions generated within a 10 minute time bound and report these results in Table 6.2. The table only shows summaries of the best compensated process instances from multiple runs of evaluation (violations are randomly generated). The evaluation is run on Intel® Core™ i5–4440 with 16GB memory in Ubuntu 12 and Java SE 7.

### Analysis of the results

The results we obtain here are only modestly encouraging. We note that none of the minimum distances for the compensated process instances are 0, but this is not a negative (any violation will lead to a non-zero distance). The location of the

---

**Table 6.1: Evaluated Process Models**

<table>
<thead>
<tr>
<th>Process Model ID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity of Process</td>
<td>6/6</td>
<td>6/6</td>
<td>7/7</td>
<td>7/7</td>
<td>7/7</td>
</tr>
<tr>
<td>Total Number of Activities and Events</td>
<td>6/12</td>
<td>9/9</td>
<td>9/9</td>
<td>9/9</td>
<td>9/9</td>
</tr>
<tr>
<td>Length of Paths in the Model (Min/Max)</td>
<td>6/6</td>
<td>12/12</td>
<td>9/9</td>
<td>7/7</td>
<td>7/7</td>
</tr>
<tr>
<td>Complexity of Semantic Annotation</td>
<td>0/0</td>
<td>0/0</td>
<td>2/2</td>
<td>6/6</td>
<td>6/6</td>
</tr>
<tr>
<td>Size of Propositional Vocabulary</td>
<td>3/3</td>
<td>13/13</td>
<td>9/9</td>
<td>15/15</td>
<td>15/15</td>
</tr>
<tr>
<td>Length of Task Post-conditions (Min/Max)</td>
<td>1/2</td>
<td>1/7</td>
<td>2/7</td>
<td>1/3</td>
<td>1/7</td>
</tr>
<tr>
<td>Complexity of Knowledge Base</td>
<td>3/3</td>
<td>3/3</td>
<td>3/3</td>
<td>3/3</td>
<td>3/3</td>
</tr>
<tr>
<td>Number of Clauses in Knowledge Base</td>
<td>4/4</td>
<td>10/10</td>
<td>7/7</td>
<td>10/10</td>
<td>10/10</td>
</tr>
<tr>
<td>Number of Activities in Capability Library</td>
<td>4/4</td>
<td>10/10</td>
<td>7/7</td>
<td>10/10</td>
<td>10/10</td>
</tr>
</tbody>
</table>
violation is clearly important. A violation at the beginning of a process presents a much larger search space than a violation later in the process. The more complex the semantic annotations are, the longer it takes to compute compensations (which is not surprising). Process models 4 and 5 are structurally identical, but 5 has semantic annotations that are significantly more complex than those of 4. As a result, we are able to compute a goal-satisfying compensation from process 4 within the time-bound, but not for process 5. In general, not all of the “closest” process instances are goal compliant. Many socio-technical processes of interest have durations far greater than 10 minutes, hence the fact that we are able to compute goal-satisfying compensations for many (if not all) of the processes is actually encouraging. This suggests that with a higher time-bound, we might find even better and more goal-satisfying compensations, while still being able to compensate quite early in these long-duration processes.

### 6.5 Related Works

Cook et al. [28] offer a process validation framework, which involves comparing the event stream from the process model against the event stream from the log using string distance metrics. Rozinat and van der Aalst [107] developed the Conformance Checker as part of the ProM framework which, given a process design and a collection
of its event log from execution, determines whether the process execution behavior reflects the designed behavior. Different from [107] and [28], our semantic conformance checking assumes that the instance of executed process is structurally correct. A number of proposals for goal-oriented process management exist [53, 73]. Klein and Dellarocas [69] present a knowledge-based approach to exception detection and handling in work-flow systems. They define an exception as “any deviation from an ‘ideal’ collaborative process that uses the available resources to achieve the task requirements in an optimal way” [69]. In their exception management approach, the participant of an enacted process will be notified when there is an exception with the exception types and associated exception handler processes proposed by the work-flow designer, so that the participants are able to modify the instance of the process to resolve the exception and allow the process to continue. Our approach does not require that exceptions handlers be written for every possible exception.

6.6 Conclusion

In this chapter we present a novel framework for semantic monitoring and compensation of business processes, leveraging semantic annotations of process designs. We identify some abstract strategies for implementing such a framework, and then present a concrete implementation. The evaluation of the implementation suggests that there is modest room for optimism that such an approach would be viable in practice.
Chapter 7

Robust Process Adaptation using Game Tree Search

A robust machinery for process enactment should ideally be able to anticipate and account for possible ways in which the execution environment might impede a process from achieving its desired effects or outcomes. At critical decision points in a process, it is useful for the enactment machinery to compute alternative flows by viewing the problem as an adversarial game pitting the process (or its enactment machinery) against the process execution environment. We show how both minimax search and Monte Carlo game tree search, coupled with a novel conception of an evaluation function, deliver useful results.

7.1 Introduction

It is generally recognized that business processes need to be executed in a manner that is robust and resilient to changes in the operating environment within which these processes are executed. The challenge is not only to be flexible enough to deal with immediate impediments to process execution, but to also anticipate future
states of affairs that might impede process execution (or the achievement of process goals). Impediments to the successful execution of a business process can appear in many forms. For instance, an outsourced search for past buying behaviour of a customer in a credit check process might return no results, or results for the wrong customer, thus preventing the successful execution of an instance of that process. An automated process for maintaining the ambient temperature inside a building might be impeded by a non-functioning air-conditioner, or by a faulty sensor that reports incorrect temperature readings. A clinical process might have to face obstacles caused by a patient who forgets (or deliberately ignores) to ingest a prescribed pill left at his/her bedside by a nurse.

Most of the examples above involve functional impediments (that prevent the achievement of functional process goals in a manner akin to the notion of obstacles \[133\]). Non-functional impediments can also occur, such as when the outsourced service for retrieving the past buying behaviour of a customer delays the delivery of its results, thus preventing a process from meeting its non-functional requirements. The framework we develop in this chapter is general enough to handle both kinds of impediments, but we mainly focus on functional impediments in the formalization and evaluation due to space restrictions.

Traditional conceptions of process designs that rely on task IDs to represent information about the effects of that task are not easily amenable to the kinds of analysis that would reveal whether a given state of affairs impedes the achievement of process goals. To perform this analysis, we would require an exhaustive enumeration of all possible states of the environment that might present obstacles to the successful execution of a process task — an often-impossible exercise. This analysis is significantly simpler (and can be performed at runtime) if processes are annotated with task post-conditions. For instance, a given state of the environment would impede the process if it negated any
of the desired post-conditions at that point. A large body of reported work leverages semantic annotation of business process designs \[114, 48, 61, 64, 40, 122, 135, 39, 54\]. A number of proposals also address the problem of semantic annotation of web services in a similar fashion \[104, 82, 84, 121\]. Our framework, therefore, leverages semantically annotated process models (i.e., process models where each task is annotated with post-conditions).

Our discussion above suggests that the relationship between a process and its operating environment can often be adversarial. The adversarial behaviour of the environment might be intentional (where entities within the environment might have an interest in preventing the successful execution of a process) or unintentional (where the natural behaviour of the environment throws up impediments). In either case, there is value in viewing the interaction between the process and the environment as an adversarial game pitting the process (which is, say, the maximizing player) against the environment (the minimizing player). The value of a game formulation stems from the following. In a manner akin to a traditional 2-player adversarial game, the process can reason about a sequence of moves it might make (tasks it might execute) that would help achieve process goals (winning states in a game formulation) in the face of counter-moves by the opposing player (impediments thrown up by the environment). The game is one of perfect information since the state of the game (in this case, the state of the process operating environment) is equally accessible to both players. The game involves turn-taking, with the process making a move, then the environment making a move and so on. While this does not necessarily exactly reflect what might happen in a real-world setting, it serves as an adequate abstraction. This game-tree search formulation of the problem relies on the process having access to some modicum of understanding of the behaviour of the environment (an environment behaviour model). In the simplest case, this might be a set of impediments (conditions in the operating
environment) that might hold. To be maximally robust, the process might perform worst-case reasoning by assuming extreme adversariality of the environment, where the environment makes those conditions true that most impede the achievement of process goals. More sophisticated models of the behaviour of the environment (via state transition models or via the generation of impediments in a context-sensitive fashion) could also be used.

It is useful to consider the manner in which adversarial game-tree search might be incorporated into the process execution machinery. Game-tree search essentially involves process re-consideration, i.e., re-designing the normative process flow (i.e., the flow mandated by the process design). Given that our understanding of possible impediments occurring in the operating environment of the process is context-sensitive in general (in many settings, it is difficult to re-compute or predict these impediments at design time), it is useful to reconsider process designs during run-time. The granularity of reconsideration can be parametric. At the one extreme, we have step-wise reconsideration, where we re-visit what is to be done next after every step. More generally, we might adopt a policy of reconsidering after every $k$ steps (or $k$-step reconsideration, the lower the $k$, the more reactive the process is). Process reconsideration can also be triggered by situations where the observed effects do not correspond to the expected effects (something referred to as semantic non-conformance in [57]). It is also useful to note that adversarial game-tree search can form the basis of offline process design, and the experimental results we present later in the paper may also be viewed as illustrating that use case. Process reconsideration can be used to decide the immediate next step, or the sequence of the following $n$ steps (closely related to $k$-step reconsideration).

We offer two formulations of the robust process design/execution problem, first in terms of the well-known minimax game tree search algorithm (with $\alpha$-$\beta$ cutoffs),
and second in terms of the Monte Carlo tree search algorithm. Our experimental evaluation suggests that this approach to achieving process robustness/resilience is practical. The experimental evaluation also offers a more nuanced understanding of the merits of minimax search with $\alpha$-$\beta$ cutoffs relative to Monte Carlo tree search. Ultimately, the intent is to compute a sequence of tasks (which might be at variance with the mandated process model) that is most likely to achieve process goals in the face of potential impediments. In the case of the clinical process example mentioned earlier in this section, a robust workaround would be to execute an additional task that involves a nurse monitoring the ingestion of the prescribed pill by the patient (similar workarounds can be imagined for the other example settings).

The rest of the paper is structured as follows. Section 2 describes the setting of semantically annotated process models as well as the normative and observed execution traces that the proposed framework leverages. Section 3 describes the formulation of the robust process enactment problem as adversarial game tree search. Section 4 provides a detailed experimental evaluation while Section 5 provides concluding remarks.

## 7.2 Processes Annotated with Post-conditions

In this section, we define the class of process models that our proposal relies on, specifically *semantically annotated process models*. We also define the notions of normative and semantic execution trace that we shall leverage in the heuristic evaluation required by game tree search (these latter notions were first defined in [57], and our exposition below summarizes those results). The definition of a semantically annotated process model refers to effect scenarios, which provide answers to the following question posed at design time: *given a process design and a designated point in that process design, what postconditions/effects would hold if the process were to execute up to*
7.2. Processes Annotated with Post-conditions

that point?. We assume a setting where tasks are drawn from a capability library (that describes all of the tasks/capabilities that the enterprise is able to execute). We also assume that all tasks in the capability library are annotated with post-conditions that describe the context-independent effects of executing those tasks. To answer the question posed above, we need to accumulate these context-independent post-conditions to simulate the effects of process execution. Unlike a number of the process annotation approaches referred to in the introduction that accumulate task post-conditions by using the AI planning device of add-lists and delete-lists of effects, we use the state update operator approach from [135, 64] (recall that a state update operator takes a state description and the effects of an action to generate one or more descriptions of the state that would accrue from executing this action in the input state). In our setting, the answer to the question posed above is non-deterministic in general, and is provided as a set of (mutually exclusive) effect scenarios. There are two reasons why we need these answers to be non-deterministic. First, in any process with XOR-branching, one might arrive at a given task via multiple paths, and the accumulated effects depend on the path taken. Since this analysis is done at design time, the specific path taken can only be determined at run-time (thus leading to non-determinism in the accumulated effects). Second, state update operators typically generate non-deterministic outcomes since the inconsistencies that commonly appear in state update can be resolved in multiple different ways. When the execution of a process leads to a state that is (possibly partially) characterized by an effect scenario, the execution of the next task in the model, or the occurrence of the next event, can lead to a very specific set of effect scenarios, determined by the state update operator being used. In effect, the process model determines a transition system, which determines how the partial state description contained in an effect scenario evolves as a consequence of the execution/occurrence of the next task (event) specified
in the model. We assign each effect scenario appearing in a semantically annotated process model a unique ID (thus if the same partial description applies to a process at different points in its design, it would be assigned a distinct ID at each distinct point). We can thus refer to the *predecessors* (the effect scenarios that can lead to the current scenario via a single state update determined by the next task/event) and *successors* (the scenarios that can be obtained from the current scenario via a single state update determined by the next task/event) of each effect scenario with respect to the transition system implicitly defined by the process design.

Given these preliminaries, we define a **semantically annotated process model** \(\mathcal{P}\) as a process model (such as a BPMN model) or a process graph (in the usual sense that the term is used in the literature - a formal definition is omitted to save space) in which each task or event is associated with a set of *effect scenarios*. Each effect scenario \(es\) is a 4-tuple \(\langle ID, S, Pre, Succ \rangle\), where \(S\) is a set of sentences in the background language, \(ID\) is a unique ID for each effect scenario, \(Pre\) is a set of IDs of effect scenarios that can be valid predecessors in \(\mathcal{P}\) of the current effect scenario, while \(Succ\) is a set of IDs of effect scenarios that can be valid successors in \(\mathcal{P}\) of the current effect scenario.

A semantically annotated process model is associated with a set of normative traces, each providing a semantic account of one possible way in which the process might be executed. Formally, a **normative trace** \(nt\) is a sequence \(\langle \tau_1, es_1, \tau_2, \ldots es_{n-1}, \tau_n, es_n \rangle\), where

- Each of \(\tau_1, \ldots, \tau_n\) is either an event or an activity in the process.
- \(es_1 = \langle ID_1, S_1, \emptyset, Succ_1 \rangle\) is the *initial effect scenario*, normally associated with the start event of the process;
- \(es_n = \langle ID_n, S_n, Pre_n, \emptyset \rangle\) is the *final effect scenario*, normally associated with the end event of the process;
7.3 The Robust Process Enactment Problem

- $es_1, \ldots, es_n$ are effect scenarios, and for each $es_i = \langle ID_i, S_i, Pre_i, Succ_i \rangle, i \in [2..n]$, it is always the case that $ID_{i-1} \in Pre_i$ and $ID_i \in Succ_{i-1}$;

We shall refer to the sequence $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ as the identity of the trace $nt$. To simplify exposition, we will on occasion use $es$ to refer to only the $S$ in the 4-tuple denoting an effect scenario.

A semantic execution trace of a process $P$ is a sequence $\langle \tau_1, o_1, \tau_2, o_2, \ldots, \tau_m, o_m \rangle$ where each $\tau_i$ is either a task or an event, and each $o_i$ is a set of sentences in the background language that we shall refer to as an observation that describes (possibly incompletely) the state of the process context after each task or event. We shall refer to the sequence $\langle \tau_1, \tau_2, \ldots, \tau_m \rangle$ as the identity of the execution trace. Note that we do not require each $\tau_i$ to belong to the process design $P$ to allow the possibility of actual executions being erroneous, or to represent on-the-fly re-designs.

7.3 The Robust Process Enactment Problem

We address the robust process enactment problem, defined as follows:

Given

- A semantically annotated process model $P$,

- A capability library consisting of tasks with context-independent post-conditions $C$ (in more sophisticated settings, we might view each element of $C$ as having both a precondition $pre$ and a postcondition $post$),

- An environment behaviour model $M : S \rightarrow 2^S$, where $S$ is set of all possible states,
• A set of goal conditions (the achievement of any one of which would count as successful process execution) $\mathcal{G}$,

• The sequence of tasks $\langle \tau_1, \tau_2, \ldots, \tau_i \rangle$ that have been executed thus far,

• The current observed state $o_i$ of the process operating environment, and

• A state update operator $\oplus$,

Determine:

• A sequence of tasks $\langle \tau_1, \tau_2, \ldots, \tau_i, \ldots, \tau_n \rangle$ where each $\tau_i \in \mathcal{C}$ that is most likely to achieve a goal-satisfying state (i.e., a state that makes at least one of the goal conditions true) under the assumption that the environment behaves in a maximally adversarial fashion.

Note that this formulation permits us to also consider other variations, such as: (1) determining what the next task should be, (2) determining what the next $k$ tasks should be and (3) determining at the initial state what the complete sequence of tasks should be that would lead to a goal-satisfying state.

Consider for example a car servicing process in an auto repair/maintenance store that consists of a task that replaces the tyres of the car, then a wheel alignment task, followed by a road test. Suppose that the alignment machine in store is currently unavailable. As the result, the normative process task (i.e. wheel alignment) cannot be performed. The capability library of the store suggests some alternative tasks or task sequences can be executed to achieve the same process goal which may include (see Figure 7.1)

1. Ask the customer to take the car to another shop for wheel alignment and reimburse the customer against a receipt, then complete the remaining tasks in-store;
(2) The shop takes the car to a pre-arranged shop for wheel alignment, pays that shop, then completes the remaining tasks;

(3) Ask the customer to come back another day when the machine becomes available by delaying the currently impeded process;

(4) Rearrange the task sequence, i.e. skip the current task and complete the remaining tasks first, then do the wheel alignment later when the machine becomes available.

At first blush, all tasks/task sequences that resolve the current impediment seem to be equally feasible. However, if we look further into the future, the ramifications of
some of the options above may cause problems for later tasks and impact quality of service requirements such as the overall customer satisfaction, service standards, etc. For example, if option (4) is taken and the road test is performed before the wheel alignment, the test may be unsafe and the test result may be inaccurate. In addition, after wheel alignment is performed, another road test may be required. Option (3) requires the customer to schedule another time, which may lead to a dissatisfied customer. As the result, options (1) and (2) may be better options as they only delay the current process instance slightly during the time of the unavailable machine, and do not increase the possibility of any future impediments. They may however increase the cost, and in particular option (1) may reduce the customer satisfaction, as well as introduce some potential new impediments (worst case scenarios) such as the arranged shop refusing to provide wheel alignment service due to the large amount of requests that affect its normal operation, or the customer losing the receipt for reimbursement etc. Thus, the purpose of the game-tree search is to consider and evaluate all the feasible alternatives during process execution.

The general problem can be instantiated in a variety of ways. The assumption of maximally adversarial behaviour on the part of the environment is a form of worst-case reasoning. It entails that the environment will behave in a manner (consistent with the environment behaviour model) that most impedes process goal satisfaction. This does not necessarily mean that the environment is deliberately adversarial, but only that the worst-case behaviour of the environment has been taken into account in deciding what to do next during process enactment. We will say that a condition (made true by the environment) $c$ impedes the achievement of a goal condition $g$ if and only if $c \land g \models \bot$. In some cases, we might approximate the environment behaviour model via a set of conditions that the environment is capable of bringing about. In other cases, we might provide more sophisticated behaviour models in the
7.3. The Robust Process Enactment Problem

form of state transition systems or sets of event-condition-action rules. The sequence of steps executed thus far might be empty if a procedure for solving this problem is invoked at the start of the execution of a process instance, or if the intent is to compute a maximally robust process design. The current state of the process operating environment is important as an independent input since the sequence of process steps executed might lead to a predicted state of affairs (via the accumulation of task post-conditions as discussed earlier) that might be at variance with the current state (this, in itself, can be a trigger for process reconsideration). When the intent is design-time analysis for computing robust process models, the current state might be left empty, or populated by the expected start state of the process (one could also reason by cases and compute multiple process models if a set of mutually exclusive start states need to be accounted for). The accumulation of task post-conditions involves the application of a state update operator. Several such operators have been proposed in the literature, two prominent ones being the Possible Worlds Approach [56] and the Possible Models Approach [137].

Our proposal involves the use of game tree search to solve a 2-player adversarial game of perfect information in addressing the robust process enactment problem. The two players are the process and the environment. The game is one of perfect information since the state of the environment is equally accessible to both players. The simplest game tree search algorithm is minimax search, which involves a maximizing player (that seeks to maximize the payoff or utility) and a minimizing player (that seeks to minimize the payoff — all states of the game being assessed by the same utility/payoff function). The key data structure is a game tree where each node represents a state of the game and nodes at alternating levels represent states of the game that can be achieved via moves made by a given player. For a fully expanded game tree, the leaf nodes represent end-game states (these can be labelled with 1,
-1 and 0 to represent a win, loss or draw for the maximizing player, or with values from a real-valued interval to represent degrees of winning etc. for the maximizing player). The minimax algorithm proceeds by propagating these values up the game tree, with a node corresponding to a state where the maximizing player makes a move being labelled with the maximum of the utility values of its child nodes (and the converse for nodes where the minimizing player makes a move). The intent is to obtain a payoff/utility value labelling all of the child nodes of the root of the tree (the state at which a move must be made by one of the players). Once these labels are obtained, the maximizing player selects the move that leads to the state with the highest utility (converse for the minimizing player). For most complex games (such as chess), the full tree is too large to enumerate, and search proceeds by cutting off the tree at a fixed (parametric) depth and treating the nodes at that depth as pseudo-leaf nodes. Since these nodes do not represent end-game states, they do not have exact payoff values associated with them. Instead, a *heuristic evaluation function* is used to estimate the “goodness” of a given node (an approximate indicator of the likelihood that a move leading to that node will eventually lead to a win for the maximizing player). Minimax search with $\alpha$-$\beta$ cutoffs involves bound propagation on payoff values to prune the search tree (we do not provide a more detailed exposition due to space constraints).

In our setting, each node represents a state of the process operating environment. The moves available to the *process* player correspond to tasks in the capability library while the moves available to the *environment* player correspond to the conditions (potential impediments) that can be made true by the environment as per the environment behavioural model. Both task post-conditions and impediments can be viewed as sentences in the underlying language. Given such a sentence $e$ and a prior state $s$, the resulting state is denoted by $s \oplus e$ where $\oplus$ is the state update operator provided as input. State update operators generate possibly many non-deterministic outcomes in
the general case (the Possible Worlds Approach, for instance, generates as a resulting state $s' \cup e$ for each maximal — with respect to set inclusion — subset of $s$ that is consistent with $e$. The non-deterministic states associated with a given move represents a point of departure from standard minimax search (where a given move leads to a unique state). This can be handled easily by extending the worst-case reasoning approach that underpins minimax search. Thus, if a maximizing player contemplates a given move, it will pick the state with the highest payoff amongst the possibly many states that can result from that move as the resulting state (converse for the minimizing player).

Designing a heuristic evaluation function that is able to estimate the likelihood of a given state leading to a “win” for the maximizing player is another challenge. The evaluation function we use in generating the experimental results presented in the next section is conceived with the following intuition in mind. Instead of assigning numeric values for each state, this function generates a preference ordering on a set of states (which can be used in much the same way as a set of numeric payoff values). A state $s$ is preferred over another state $s'$ if $s$ is “closer” (in a sense to be made precise below) to either the nearest goal state or the nearest state in any normative execution trace with an identity (recall the definition at the end of Section 2) for which the sequence of tasks already executed serves as a prefix (and which has not been already traversed in the execution thus far). Given a set of sentences $t$ and a background knowledge base $KB$, we use $Cn_{KB}(t)$ to denote the set of all logical consequences of $t \cup KB$. Let the union of the goal states and the states in the normative execution traces discussed above be referred to as the set of desired states. One plausible and intuitive means (but by no means the only one) of assessing the proximity of a state $s$ to a desired
state $d$ (denoted by $f(s, d)$) is as follows:

$$f(s, d) = \frac{|Cn(s) \cap Cn(d)|}{|Cn(d)|} \quad (7.1)$$

This function obtains a higher value when the cardinality of the intersection of the set of consequences of $s$ and $d$ gets closer to the cardinality of $d$. In the experimental evaluation, we compute the number of clauses in a CNF representation of $d$ that are entailed by $s$, as one computational realization of the expression above. Since we are able to work with ground theories (universally quantified rules in the $KB$ are replaced by their ground instances — of which there is a relatively small number), we use the SAT4J SAT solver as our theorem prover.

Each step in the search process proceeds as follows. If the current (observed) state is $o$, and it is the process player’s turn to make a move, then the set of feasible next states (that determine the next level of the game tree) is given by the set $o \oplus post$ for each $post$ associated with available tasks in the capability library (in more sophisticated settings where we also have task pre-conditions, we determine whether $o \models pre$ before we conclude that a task is feasible to execute). If it is the environment player’s turn to make a move, then the set of feasible next states is determined by $\bigcup M(s_i)$ for each $s_i$ satisfying $o \models s_i$.

It is useful to consider the impact loops in a process in this context. It is fairly obvious that generating semantic annotations for process designs with loops is problematic (simply because we cannot predict at design time the number of times the process would loop). In the context of the robust process enactment problem, however, loops pose no problems. It is perfectly feasible for this framework to return a task sequence that includes multiple iterations of a task or task sequence, if that is the best strategy for dealing with potential obstacles (as a trivial example, we might end up needing to press a “temperature-up” button on a thermostat to achieve the desired temperature).
Let $I$ be the set of *impediments* that the environment is capable of making true. For the purposes of experimental evaluation, we adopt a maximally adversarial model of the environment. We generate the set of impediments $I$ by creating a distinct impediment from the negation of every clause in a conjunctive normal form (CNF) representation of a goal. Thus, given a set of goal states $G_p$ (in CNF) of the process $p$, the set of impediments $I_p$ is defined as:

$$I_p = \{\neg c | \forall c \in g, \forall g \in G_p\} \quad (7.2)$$

**Minimax Tree Search**

In this evaluation we use minimax tree search with $\alpha$-$\beta$ cutoffs. Due to the complexity of the search, we limit the depth of the game tree to 5. The heuristic evaluation function discussed in the previous section is applied to the pseudo-leaf nodes at this depth.

**Monte Carlo Tree Search**

We compare the performance of minimax search with $\alpha$-$\beta$ cutoffs against a popular Monte Carlo Tree Search (MCTS) algorithm, namely Upper Confidence Bounds for Trees (UCT) (the description of the algorithm can be found in [20]) with a random play-out simulation to evaluate a given state. Every time the process needs to select the next task to execute, the tree is sampled $5n$ times where $n$ is the number of possible tasks available for the process to select. We then select a leaf node that is not a terminal state (i.e., a goal state), expand it and perform random play-out at all the newly expanded nodes. The random play-out is a simulated game play where each player randomly selects a move at every turn until the game reaches a terminal state.
or until the time runs out (we use a timeout of 3 seconds). The value 1 is returned when a goal state is reached in the random play-out. Alternatively, 0 is returned if a goal state cannot be reached in the given time (3 seconds).

**Evaluation Data**

Semantically annotated process models that generate sensible results are difficult to randomly generate. Two sets of hand-crafted semantically annotated process models are used in this evaluation. The first set of 8 processes are structurally and semantically simpler. Process1 is the simplest, with only 4 tasks in sequence, 1 or 2 assertions in the effects of each task and 3 rules in the background knowledge base. Process2 has the same number of tasks as Process1 but with one extra XOR branch and slightly more complex semantic annotations. Process3 is also a sequence of tasks, but with more complex semantic annotations. For the rest of the processes (Process4 to Process8), we progressively increase either the structural complexity (i.e. more tasks and/or more XOR branches), or increase the complexity of the semantic annotations. Process8 is a “real world” process created using information available at workflowpatterns.com.

The second set of processes have considerably greater structural and semantic complexity relative to the first set. The number of unique tasks (capabilities) in each process in this set varies from 10 to 100, with the number of XOR gates varying from 1 to 15. In terms of semantic complexity, we use between 20 to 50 propositional state variables to describe states of objects in the environment (which is expressive enough for most task effects/postconditions of interest). We assume that each task will impact between 1 to 5 state variables. We use between 20 to 50 rules in a background knowledge base that constrain the state changes.

---

1Process8 is Process09 in Appendix A
Evaluation Setup

For every process, we first compute all the unique instances (i.e. sequence of tasks and events from beginning to the end of the process) and for each of these instances, we simulate the process execution by computing a normative trace (as defined earlier and in [57]). This is the base line simulation without any impediments, with each normative trace leading to a goal state (this is used to compute the success rate, i.e., the number of goal-satisfying instances divided by the number of distinct normative traces). Then we generate a sequence of impediments of length of \((n - 1)\) where \(n\) is the number of tasks in the process instance, by randomly selecting impediments from \(I_p\). We insert one impediment after every completed task (except when the goal is realized after a task, which is when we force the simulation to stop).

We run three kinds of simulations:

**Standard Process (execution)** In this simulation, we use the exact sequence of tasks in the process instances (but with impediments inserted after every task) to see if, in a maximally adversarial environment, the process is still able to achieve its goal.

**MCTS** Here, the next task for the process to execute is selected using MCTS. Again, one impediment will appear after each task unless after the task, the goal is achieved, or, there are no more impediments.

**Minimax Tree Search** The setup is the same with MCTS except the next task is selected using the minimax algorithm.

The evaluation is run on Intel® Core™ i5–4440 with 16GB memory in Ubuntu 16 and Java SE 8.
7.4. Evaluation

Table 7.1: Summary of Result — Simple Process Set

<table>
<thead>
<tr>
<th>Number of Available Tasks</th>
<th>Total Process Instances</th>
<th>Success Count</th>
<th>Average Simulation Time (seconds)</th>
<th>Average Decision Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard Process</td>
<td>MCTS</td>
<td>Minimax</td>
<td>MCTS</td>
</tr>
<tr>
<td>Over All</td>
<td>N/A</td>
<td>294</td>
<td>101</td>
<td>192</td>
</tr>
<tr>
<td>Process1</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Process2</td>
<td>4</td>
<td>43</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td>Process3</td>
<td>10</td>
<td>25</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Process4</td>
<td>7</td>
<td>41</td>
<td>26</td>
<td>23</td>
</tr>
<tr>
<td>Process5</td>
<td>7</td>
<td>58</td>
<td>26</td>
<td>42</td>
</tr>
<tr>
<td>Process6</td>
<td>9</td>
<td>43</td>
<td>21</td>
<td>34</td>
</tr>
<tr>
<td>Process7</td>
<td>10</td>
<td>48</td>
<td>11</td>
<td>32</td>
</tr>
<tr>
<td>Process8</td>
<td>9</td>
<td>29</td>
<td>8</td>
<td>13</td>
</tr>
</tbody>
</table>

Evaluation Results:

Table 7.1 shows a summary of the evaluation on the simple process set, where each row is the summary of instances of a process in the simple process set except the first row, which is a summary of all process instances. The number of available tasks indicates how many distinct tasks populate the capability library, which translates to the number of tasks that are used to construct the process model in the standard process simulation, and the number of “moves” for MCTS and minimax tree search to consider at each step (when it is the process players turn to make a move). The number of total process instances indicates the total number of process instances that have been simulated, where each instance uses the same sequence of impediments. The success counts record the number of times each method (standard process, MCTS, and minimax tree search) successfully achieves the goals of the process. The average simulation time measures the total time MCTS or minimax tree search takes to terminate in seconds (the termination means either the goal is realized or the two methods have used the same number of tasks compared to the standard process). The average decision time measures the average time taken by MCTS or minimax tree search to find the next best task to execute.

It is clear that the standard, predefined process is not reactive enough in this setting where the environment constantly acts against reaching a goal state. MCTS shows
improvements in the overall success rate but minimax tree search is able to achieve the highest rate of success. The downside to using minimax search is the computational cost (average decision time).

For processes with simpler semantics (Process1, Process2, and Process3), minimax tree search makes a decision much faster compared to MCTS, and the time spent increases when there are more available tasks to evaluate for both methods. However, for processes that have complex semantics (larger number of rules in the knowledge base and larger sets of assertions in the postconditions of each task, as exemplified by Process4, Process5, Process6, and Process7), the time spent by minimax tree search increases dramatically, and minimax tree search takes longer than MCTS except for Process6. Process6 is a special case, possibly because the set of rules in the knowledge base creates a simpler problem for the underlying reasoning machinery to solve. Overall, in all 294 process instances, it is clear that MCTS is more efficient for large complex processes. Minimax tree search achieves a higher success rate, but can take a very long time to decide on the next best move for large complex processes.

The next set of results involve the complex set of processes. The major issue in conducting this evaluation was that the minimax tree search took more memory than available (2GB) in the experimental setup leading to the simulation be terminated. Consequently, we are only able to complete a small number of simulations successfully given the time and space limitation with the unoptimized prototype used in the evaluation. Some of the issues we had with the more complex process models can be overcome by optimizing the game tree search as well as the actual implementation, which is beyond the scope of this chapter. Figure 7.2 shows the success rate of the 3 methods, and Figure 7.3 illustrates the time taken by MCTS and minimax tree search to select the next task to execute, which shows that MCTS is able to select tasks relatively quickly to achieve a more than 80% chance of success.
7.5 Related Work

Process flexibility has long been recognized as an issue in real world business process management [131, 21, 60, 69, 105]. Part of the existing literature on process flexibility addresses flexibility by design [62], including exception handling [69], or achieving minimal deviation from a design during execution [57]. Other parts of that literature addresses flexibility at runtime [87], by taking into account risk [27], by generating optimized enactment plans given multiple optimization objectives [68] and in contexts where processes are human-driven [6]. Agent technology has also been used to model flexible processes, as the agent architectures are designed to deal with a flexible environment [21]. Schuschel and Weske adapt planning algorithms developed in the agent community for process planning [112]. Our approach is effective in anticipating impediments and devising workarounds, issues which most existing proposals tend not
to address.

7.6 Conclusion

This chapter highlights a hitherto under-explored connection between game-tree search and business process management. Preliminary results suggest that incorporating a game-tree search based module which reconsiders the intended flow of a process in view of likely conditions that might occur in the operating environment which might impede the process can lead to more robust processes that achieve their goals despite these impediments.
Chapter 8

Computing process compensation in complex, dynamic and potentially adversarial domains

It is common that when a process is deployed in the business environment, some process instances may be facing the issues where the environment might impede the process of achieving its desired outcomes. The undecidability of the behaviour of a complex and dynamic business environment is largely ignored by the process design and execution. Following a process design blindly in a dynamic environment is equally bad as playing a competitive game with only a fixed strategy against a cunning opponent. In this chapter, we propose a framework that enables the prediction of the process outcome and on-line redesign (compensation) of the process instance with respect to the dynamic and undecidable behaviour of the environment to maximise the expected outcome of the process instance.
8.1 Introduction

Chapter 6 presents a framework that detects semantical non-conformant and systematically searching for compensation. The compensations are considered and deployed as soon as there is a semantical violation. In Chapter 7, we allow these semantically non-conformant states to exist in execution that provide some level of freedom to the process instance. The reason is that we now assume all the semantic non-conformance are the result of the adversarial behaviour of the environment. That is to say we assume all the tasks will always bring its effect to the state (at least briefly) and if we cannot observe the effect or we only observe partial effect, it is because the environment made an adversarial action between the completion of the task and our observation. We are then able to find out the best task to preform the next that will eventually lead to a goal compliant state. This framework does not follow the predefined process model, where in most of the real process instance, the process model should always be the first preference. The problem arises, if we follow the process model as is, the process instance will not be able to adapt and react to the changes in the environment. If we rethink what to do after every step, we are being extremely adaptive but not following the process model at all. We are trying to find a way to balance the adaptiveness and conservativeness.

This section provides some preliminary results of such a system, in which the process execution is considered as a game against the world. The system utilises the Monte-Carlo method as a tool for predicting and measuring the outcome of the current process instance, and preform process redesign (to the remaining part of the process that is not yet executed) whenever the bad outcome is predicted to be inevitable.

This chapter is structured as following. Section 8.2 provides preliminary definitions. Section 8.3 formalises the process execution as a two-player game. Section 8.4 describes the process engine that predicts execution outcomes using Monte-Carlo method. Section 8.5
provides a detailed experimental evaluation. Section 8.6 discusses how this chapter is compared with other works while Section 8.7 provides concluding remarks.

8.2 Preliminaries

To be more generalised than the previous chapters, here we allow the domain of environment variables to be either logical, discrete, or continuous, where each variable has its own domain, and some necessary operations defined to aggregate, manipulate, and update the value of the variable. Each of the variables can be used to describe one or more properties of the process, the process instance, and the process execution such as the functional and non-functional properties, resource, cost, execution progress and so on. For example, a functional property of the process can be described using a logical variable (e.g. an achievement goal). If we allow temporal logic rules in the background knowledge base, then a maintenance goal can be described too, which is more related to process compliances than robustness of the execution and is not part of our theme. The total resource availability and the cost of the process execution can be represented with a variable with a continuous domain, the availability of a particular resource may be represented with a boolean variable and the allocation of this resource is described by a discrete variable. We are not going to the detail of how and what each variable represents as it is domain specific. The general idea is that there exist variables that describe the necessary information required by the later analysis. We also assume that it is not always the case every variable is observable in every state of the execution (partial observability). Formally, we let \( s = \{v_1, \ldots, v_n\} \) denote a state of the world where \( \{v_1, \ldots, v_n\} \) is a set of values of the environment variables. We use \( S \) to represent a set of all possible states.

**Definition 8.1 (Effect).** The effect \( e = \{\delta_1, \ldots, \delta_n\} \) is a set of value changes for the environment variables \( \{v_1, \ldots, v_n\} \). We use \( \mathcal{E} \) for the set of all possible effects.
We use $\delta$ as an abstract concept of changes of a given variable. The representation of the changes and the operation that aggregates and updates the changes depends on the variable. For example, if the variable representing the time spent in execution, where the value of the variable is a real number of seconds. The effect of a task on this variable is $\delta_{\text{time}} = 60$, and the aggregation operator is addition. For logical variables, the notation of changes and how to update the value of such a variable may depend on the underlying logic system and the knowledge rules. The effect annotation discussed in Chapter 3 and the state update operator described by Definition 3.1 are an example, where the effect is described in logic assertion and the update is done by the possible world approach the respect to the knowledge base. Another possible representation is STRIPS-style representation [49]. If the variable is a discrete variable that represents the level of customer satisfaction, $\delta$ could be its new value, or increment, for example, that increases the value of the variable to the next level.

We assume there exists a capability library that includes the set of all possible tasks and their effects the organization is capable of completing under all the situations that are related to the given process model, where the tasks of the given process model are always in the capability library. This is the same as in Chapter 6 and Chapter 7. Similarly, we also assume there exists a set of impediments that the environment is capable to bring about.

8.3 A Game Against the World

Compared with Chapter 7, where the game-tree search is run every step after a task to maximise the behaviour adaptation by being extremely reactive and totally disregards the process model in the process, this chapter is trying to find a way of balancing the adaptation and conservation (following predefined workflow). In most of real process instances, it is commonly preferred to follow the process model unless some changes
are absolutely necessary. This time, we equip the process engine with a prediction
machinery that will predict the possible outcome of the process instance. The tree
search is used as a planning machinery this time whenever the prediction is worse
than a threshold.

**Definition 8.2** (Process Adversarial Game). Given

- A process $\mathcal{P}$
- A capability library $\mathcal{A}$ consisting of tasks with context-independent post-conditions
  (in a more sophisticated settings, we might view each element of $\mathcal{A}$ as having both
  a precondition and postcondition). Note that the tasks in the capability library is
  a set of tasks that at least contains all tasks of the process $\mathcal{P}$ (super set of the
  set of tasks in $\mathcal{P}$).
- A set of impediments (i.e. risks) $\mathcal{I}$ that the environment is capable of stopping
  the process from realizing its goal, which can be seen as all possible risks in
  executing the process model.
- An environment behavior model $E$ that maps the given state to a set of possible
  impediments.
- A set of goal conditions $\mathcal{G}$, and achievement of any of $g \in \mathcal{G}$ indicates the
  successful process execution.
- The sequence of tasks $\langle \tau_1, \tau_2, \ldots, \tau_i \rangle$ that have been completed thus far.
- The current observed state $o_i$ of the environment.
- A function $f : \mathcal{S}, \mathcal{E} \rightarrow \mathcal{S}$ that updates the a given state $s \in \mathcal{S}$ to a new state
  $s' \in \mathcal{S}$ with an effect or an impediment.

to determine:
the probability \( p \) of the current process instance to reach a goal state following the current process model \( \mathcal{P} \), and

- if there exists an alternative process model \( \mathcal{P}' \) that from the current point of execution, has a probability \( p' \) to reach a goal state such that \( p' > p \).

### 8.4 Process Engine with Adversarial Compensation

The procedure of our process engine (shown in Algorithm 8.1) is designed to execute tasks following a predefined process model \( \mathcal{P} \) as long as the run-time situation allows. The engine executes or instructs actors to execute the next available task \( \tau \), while calculates the expected state \( s \) of the environment once the execution of the task is completed. It then observes the state of the environment \( s' \), and calculates the differences \( e_I \) between the observation and the expected state, which is assumed to be the environment impediment. An environment behaviour model is updated with the latest instance of impediment. The environment behaviour model may be implemented in many different ways, such as a rule-based system, an instance-based system, a statistical model or even a deep neural network. We are not going to discuss the pros and cons of the different possible implementation as they may be domain dependent and comparing the performance of each implementation does not concern the design of the engine. The purpose of including an environment behaviour model is to model the behaviour of the environment player so that it can be used in prediction later.

The outcomes of the process instance are predicted and evaluated using Monte-Carlo method, where a number of symbolic simulations of the remaining part of process yet to finish is run to estimate the probability of goal realisation, i.e. winning the game (Algorithm 8.2). In the simulation, 2 player behaviour models are used, the process model and the environment behaviour model. In the normal execution, the process
Algorithm 8.1 Process Engine with Adversarial Compensation

```
procedure PROCESS(\mathcal{P}, s)
    \tau \leftarrow \text{the next task to execute } \mathcal{P}
    \text{preform task } \tau
    e_\tau \leftarrow \text{the effect of } \tau
    s \leftarrow f(s, e_\tau)
    \text{while there are more task in } \mathcal{P} \text{ do}
        s' \leftarrow \text{observed state of the environment}
        e_{\mathcal{I}} \leftarrow \Delta(s, s') \quad \triangleright \quad \Delta() \text{ calculates the differences of 2 given states}
        \mathcal{E} \leftarrow \text{updated environment model } \mathcal{E} \text{ with the new impediment } e_{\mathcal{I}}
        p \leftarrow \text{PREDICT}(\mathcal{P}, \mathcal{E}, s')
        \text{if } p < \text{threshold then}
            \mathcal{P} \leftarrow \text{COMPENSATE(} \mathcal{P}, \mathcal{E}, s')
        \text{end if}
        s \leftarrow s'
        \tau \leftarrow \text{the next task to execute } \mathcal{P}
        \text{preform task } \tau
        e_\tau \leftarrow \text{the effect of } \tau
        s \leftarrow f(s, e_\tau)
    \text{end while}
end procedure
```

player always follows the predefined process model. Therefore, in the simulation, the process player will always follow the given model. After each task done by the process, the environment will make a move according to its behaviour model. Since it is a simulation, the impediment then is added to the state after the task is completed using the state updated function \( f() \) again. Each simulation is run to the completion of the last task, then records the total number of simulations and total number of simulations in which the process reaches a goal state. This prediction algorithm considers the randomness and undecidability of the environment behaviour (we only know what may happen but never know what actually happens in any given process instance in runtime) in the simulation and runs multiple simulations to produce the probability of winning.

Once the predicted probability of winning drops below a threshold, it is an indication that being conservative is no longer a valid option to reach any of the goal state.
Algorithm 8.2 Predict

function \textsc{Predict}(\mathcal{P}, E, s)
\begin{align*}
n & \leftarrow 0, q \leftarrow 0 \\
\textbf{while} & \text{ within computing budget for prediction} \textbf{ do} \\
& \quad s' \leftarrow s \\
& \quad \tau \leftarrow \text{the next task in } \mathcal{P} \\
& \quad e_\tau \leftarrow \text{the effect of } \tau \text{ in } C \\
& \quad s' \leftarrow f(s, e_\tau) \\
& \quad \textbf{while} \text{ there are more task to execute in } \mathcal{P} \textbf{ do} \\
& \quad \quad e_I \leftarrow \text{an impedmiment from } E \\
& \quad \quad s' \leftarrow f(s, e_I) \\
& \quad \quad \tau \leftarrow \text{the next task in } \mathcal{P} \\
& \quad \quad e_\tau \leftarrow \text{the effect of } \tau \\
& \quad \quad s' \leftarrow f(s, e_\tau) \\
& \quad \textbf{end while} \\
& \quad n \leftarrow n + 1 \\
& \quad \textbf{if} \ s' \text{ is a goal state} \textbf{ then} \\
& \quad \quad q \leftarrow q + 1 \\
& \quad \textbf{end if} \\
& \textbf{end while} \\
& \textbf{return} \ q/n
\end{align*}
end function
Then some behaviour adaptations or compensations have to be considered to increase the odds of winning. Here we propose to use game-tree search as the base of our process redesign and compensation machinery since the classic planning algorithms only consider one player (not suitable for adversarial game). The adaptation of the game-tree search for compensation is very simple. Instead of taking the child at the root of the search tree, we recursively select the best child of the current node from the root to create a sequence of best moves for the process as our new process (at least the remaining part of the process that is yet to execute). The best child is for both the players, where the best moves for the process will be part of the compensation, and the best moves for the environment (and the worst for the process) are also selected (this is worst-case reasoning). It may rise some issues with some tree search algorithm like MCTS, where the game-theoretic-optimum decision is at the root, and deeper the tree, the less confidence the decision is (more likely to be sub-optimum). As a result, the compensation from MCTS may be good at the start of sequence of the tasks found, and the probabilities of winning may decrease again. We are not concerned because we allow the compensation machinery to find another new model whenever the predicted outcome is poor again.

8.5 Experimental Evaluation

The evaluation is to explore the potential to accurately predict the outcome of the process executed in an adversarial environment using Monte-Carlo method with an environment model that is built and improved during the process execution. When the predicted process outcome is not good enough, then the process will be redesigned via game-tree search to, hopefully, improve the execution outcome. Additionally, when the predicted outcome of the redesigned process is not good enough again, the alternative redesign will be considered again until execution budget of a single process instances
8.5. Experimental Evaluation

is reached. As the result, the outcome of this process instances (redesigned) will be a failure.

This evaluation uses 10 process models with different complexities including a few of real world process models (see Appendix A for more information). Each of the processes is simulated 2000 times continuously in total (2000 process instances). The first 1000 times is dedicated to build the environmental behavior only, when the prediction routine is run but no redesign of the process model. The predictions of the first 1000 instances are only for the purpose of finding out the correlation of prediction accuracy and the accuracy of the environmental model. The last 1000 instances simulated have the run-time redesign enabled, where half of the process instances are redesigned and compensated using MCTS and another using minimax tree search (applied in alternation). Furthermore, the evaluation is run on Intel® Core™ i5–4440 with 16GB memory in Ubuntu 16 and Java SE 8.

8.5.1 Environmental Behavior Model

In this evaluation, we assume the environment is adversarial, that is the environment will act against the desired process outcome. Assuming the process outcome is described with a set of logical assertions, the adversarial environment would then randomly make one of the assertions false during the process execution, which we call an impediment. In a more general model, such impediments may also include reducing the execution budget or resources, which is not considered in this evaluation. In the simulated process execution environment, an environment may be biased towards some impediments, that is to say, some impediments may more likely to happen than others, which challenges the prediction machinery of the process execution engine. Thus, it is necessary for the prediction machinery to build an accurate environmental behavior model. In reality, the probabilities of impediments may also be dependent on the
execution context as well. For example, a call center may have limited resource off-hours which means a service call handling process may more likely to be impeded in off-hours. It is possible to consider such context into building the environment behavior model, which has been done in, for example, server load prediction, demand forecasting, traffic forecasting, etc, using the variety of statistical and deep-learning models. To build such environment behavior model requires to build more compensative process simulation environment, and normally requires more data (e.g. large number of complete process instances), which is out of the scope of the discussions in this chapter.

In this evaluation, the ground-truth model is all the possible impediments and a “no impediment” impediment that does not affect anything in the environment, thus, no effect on the process execution. The sum of the probabilities of these impediments after each process task is 1, and there is one and only one impediment is allowed after any task in this evaluation. Note we are ignoring the context dependency of the impediments to simplify the evaluation as to build the environment behavior model with context-dependent impediments would require more instances of observed impediments in process’s execution history.

With this ground-truth model in mind, the environmental behavior model can be built by calculating the number of time each impediment occurs during the process execution, and be updated during the process execution.

To evaluate the accuracy of the environmental behavior mode, we calculate the root mean squared error (RMSE) of the probabilities of each impediment in the environment behavior model ($y_i$) and the probabilities of impediments in the ground-truth model ($\hat{y}_i$) after every instance of the process simulated. Figure 8.1 shows
8.5. Experimental Evaluation

Figure 8.1: changes of MSE over 2000 continuously simulated process instances

![Graph showing root mean squared error (RMSE) over 2000 process instances](image)

the RMSE reduces when more process instances are simulated.

\[
RMSE = \sqrt{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}
\]  

(8.1)

8.5.2 Predicting Process Outcome

With the environmental behavior model, it is only possible to know the probability of what could happen but not what will happen. To predict the outcome accurately, we need to consider what could happen with respect to the environmental behavior. In this case, we utilize the Monte-Carlo method to sample the possible outcomes. The process and the environment (environmental behavior model) will take turns to act just like our normal simulation until all tasks of the process are completed, where the process will always follow the sequence of tasks defined by the process model, and the environment will “randomly” act according to the environmental behavior model. In the end, the probability to achieve the process goal is calculated from the samples (see Algorithm 8.2). The predicted probability of success (i.e. reaching the process goals) is recorded at every 10% of progression of the process instance.

\footnote{The data collected from Process03 is invalid due to some simulation errors, so it is removed from the prediction and compensation analysis.}
Figure 8.2: Predictions Errors Grouped by Processes

Figure 8.2 shows the RMSE of the prediction made at every 10% progression of the simulated process instances grouped by process models, in which the prediction of each instance at a given execution progress is $y_i$ in [Equation 8.1] and the actual outcome of the process instance is $\hat{y}_i$, where $\hat{y}_i = 1$ if one of the process goals is reached by the instance, and $\hat{y}_i = 0$ otherwise. It is then clear that for all the process instance, the closer the prediction made to the end of the instance, the more accurate the prediction is, which is reasonable and obvious as the closer to the end of a game, the less diverse of the game is.

Figure 8.3 shows the average predicted probabilities of success at the start of the instance (0%), 20%, 40%, 60% and 80% of the instance for every 200 continuous simulated instances out of the 2000 instances, while the pink-shaded area shows the
8.5. Experimental Evaluation

Figure 8.3: Predicted Process Outcome and the Average Success Rate

The figure shows the percentage of the 200 instances who succeed in reaching one of their goal states. Same as Figure 8.2, if we are only looking at the first 1000 instances where the compensation machinery is disabled, we can conclude that the prediction is the most accurate towards the end of the execution (at 80% of the process instance), and the earlier the prediction made, the less accurate in the most cases, as the 80% lines are the closest lines to the border of the shaded area, except in Process09. The prediction accuracy of the later 1000 instances may differ due to the effectiveness of the compensation machinery in finding alternative process design. We may also note that some processes start with high probability of success, then the probability decreases (Process01, Process05, Process06 and Process07), some start low and increases (Process09 and Process10), and the rest varies. These may have something to do with the capability library and
8.5. Experimental Evaluation

Figure 8.4: Predictions Errors Averaged by Instances

where the “important” or “critical” tasks are located, the ones that irreplaceable, and/or contributes the most to the goal. Moreover, The percentage of goal-realized instances in model-dependent, which shows some model is not very robust in an adversarial environment, or simply not designed for such scenario such as the real-world models, Process08, Process09, and Process10. Note that the compensation only happens when the prediction is lower than 0.5, where only Process01 frequently triggers compensation (lower than .5 at about 60% of the execution) and Process02, Process04, Process05, and Process06 need a higher threshold in order to trigger the compensation.

Figure 8.4 shows the prediction error of every 500 continuously simulated instances to evaluate how the accumulated knowledge about environment behaviour affect the
8.5. Experimental Evaluation

We expect that when the environment model become more accurately reflecting the actual behaviour, the overall prediction error will decrease. However, according to Figure 8.4, the result is inconclusive, where we did not see that the line representing the average prediction error of 0–500 instances is consistently above other lines. It is possibly caused by that we use the same number of samples in predicting the outcomes of all processes, but what we should have used is the different number of samples for different processes based on their semantic complexity.

8.5.3 Compensation (Runtime Redesign)

The process redesign is achieved via game tree search with alpha-beta cut-offs or Monte-Carlo Tree Search (with Upper Confident Bounds for Tree). The tree search algorithm provides the options to explore the task sequences that are not bounded by the original process model. In this evaluation, the game tree will be built whenever the predicted probability of success drops below 0.5 during the execution. The search tree is built using all tasks in the capability library and all possible impediments from the environment. Due to the large branching factor of this game tree, the minimax tree search depth is limited to a constant number then a heuristic function is used to calculate the value [Equation 7.1]. In this case, the heuristic function returns the normalized shortest distance of the state description at the pseudo-left node to any known states in the semantically annotated process. Monte-Carlo tree search is limited to 100 samples where in each iteration of the search, only 1 sample is taken. Then a sequence of tasks is selected according to the most valuable branch in the search tree (i.e. the branch where each node holds the highest value in its siblings), and this sequence of tasks (compensation) will be what the process is executing now. In addition, if the compensation is predicted to have 0.5 or less chance to succeed, another compensation will be calculated to redesign the current compensation until
8.5. Experimental Evaluation

Figure 8.5: Percentage of Success without and with Compensations

The maximum number of tasks are reached (currently set to be twice as long as the original process instance in the evaluation).

Figure 8.5 shows the rate of success simulation of process executions without and with compensation (run-time redesign), as well as percentage of processes that are compensated and the percentage of successful compensation (reaching one of the process goals after compensation). In the second half, it can be seen that after compensation is enabled the success rate of Process01 goes up to 0.6 while it is about 0.25 without compensation. There are more than 0.75 of the instances are compensated and about 0.25 of all instances that are compensated and achieved the process goal. For Process02 and Process06, the percentage of compensated cases is lower because their success rate without compensation is about 0.5, and we only redesign when
the predicted probability drops below 0.5. (also see Figure 8.3) The percentage of successful compensations are very low in Process02 and Process06 possibly due to the limited capabilities where it is hard to find any alternative plans that are goal compliant. The percentage of compensated cases seems to be negatively correlated with the percentage of success, which is expected that the number of compensation increases only when the predicted success rate is low. We are not able to perform a significance test because for each simulated instances of a given process, the possibility of successful compensation is not statistically independent when the knowledge about the behaviour of the environment (i.e. the probability of each impediment to occur) is collected and carried over from the previous simulations. Such knowledge is used in both predicting when the compensation is necessary and finding compensated redesign of the process, therefore affecting the behaviour of the system. The possibility of successful compensation depends on the prediction that triggers the compensation and the result of the game tree search which changes when the knowledge about the environment’s behavior changes.

Similar to Figure 8.5, Figure 8.6 shows the 1000 instances without and with MCTS compensation, where the last 500 instances have the compensation enabled, the first 500 instances are prediction only. Additionally, Figure 8.7 shows the 1000 instances of the processes without and with Minimax-tree-search compensation. It seems MCTS does better on process01 but worse on the rest, where Minimax is the opposite. This is probably due to the 100 samples limit for MCTS which restricts the search capability too much on more difficult problem compared with the limitation put on Minimax tree search in this evaluation.

Overall, the result of compensation is not good, possibly due to many reasons such as limitation of the capability libraries, which is only restricted to the set of tasks that is presented in the original process models respectively, the artificial limits we put on
the game-tree search algorithms which restrict the search too much, and possibly the nature of the process model which makes it hard to compensate. These reasons have to be further investigated to develop the theory on what it takes for a process model and its capability library to be reactive and robust against an adversarial and dynamic environment.

8.6 Related Works

Process flexibilities have long been recognized as an issue in the real world business process management \[131, 21, 60, 69, 105\]. Some of the existing literature on process flexibilities are addressed in design \[62\], or exception handling by design \[69\]. In this

Figure 8.6: Percentage of Success without and with Compensations (MCTS)
chapter, we did not explicitly mention exception handling because we are not handling exceptions. Instead, we prevent exception from occurring by compensating early.

Others address the flexibilities at the process execution, such as by taking into account of risks \[27\], by generating optimized enactment plans according to multiple optimization objectives \[68\], by following a checklist where the processes are human-driven \[6\], or by allowing minimal deviation from a design during execution as proposed in Chapter 6. Generally speaking, in this literature, to change or augment the process instances according to the execution context, it is either to follow the predefined guideline \[62, 69\] or to generate the new process model for the instances in the runtime with some form of predefined objectives either automatically or by human actors \[68, 6\].
Additionally, there are proposals that utilize the agent technologies to create more flexible process models, as the intelligent agent architectures are designed to deal with a flexible environment [21]. Automatic planning from agent-related research also is adapted for process planning [112], which argues that the manual planning (process modelling) can be replaced with planning. As a result, a more optimized and instance-based plan (process model) can be created on-the-fly during process execution. We did not employ the classical planning technologies mentioned by the literature above, as our problem is different, where there are an adversarial player (the environment).

8.7 Conclusion

This chapter provides a complete process engine that balances adaptiveness and conservativeness with respect to the process model of the behaviour of process execution and is trying to find a right trigger for adaptation. In Chapter 7, we have shown that in an adversarial environment, the process model can not always be followed, and by not following the process model (using game tree search instead) the outcome of process instances can be improved, which is to adapt at every single step in an execution. The argument against this extreme version of behaviour adaptation is that the process model no longer matters, which could make the behaviour unbounded and dangerous in a real business setting. We define and use a prediction machinery as an indicator for the necessary behaviour adaptations, which triggers compensation when the machinery realises the current execution will likely lead to a failure in execution. Compared with Chapter 6, where we use semantic violation as a trigger for compensation, the framework presented in this chapter allows some level of semantic violation as long as the predicted outcome is good enough.

The preliminary results suggest that the effectiveness of this approach differs from case to case. It may be due to the structural and semantical features of the processes,
some processes are harder to compensate, some goals are harder to achieve, and some are robust enough that less frequently require compensation.
Chapter 9

Conclusion

9.1 Overview of the results presented

9.1.1 Contributions to the body of research results

It is useful at this point to revisit the general problem being addressed. Behaviour adaptation is a key requirement in all forms of computation. Pre-programmed behaviour is limiting in a number of ways. It cannot anticipate the potentially vast range of possible situations in which that behaviour would need to be deployed or the potential adversarial behaviour of other entities in the operating context. This dissertation addresses this challenge in the context of two currently popular classes of computational machinery: intelligent agents and business processes. Pre-programmed behaviour and the inability to adapt on the fly can limit the capabilities of agent systems. This is also true of business processes with pre-defined and inflexible process designs (these limitations are particularly well-recognized in the business process community).

The fact that the focus of this thesis is on these two kinds of machinery should come as no surprise. Besides that fact that considerable investments have been made (and continue to be made) on these technologies, it is also generally recognized
that intelligent agent systems and business process execution frameworks have deep connections/similarities. In both cases, the solutions presented leverage two important observations:

1. The annotation of agent programs and business process models with post-conditions enables us to define triggers for behaviour adaptation and compute modified behaviours.

2. Viewing the interaction between the computational machinery and the environment as adversarial game-playing enables us to leverage game tree search as a means of computing robust adaptations to (a worst-case assumption of) maximally adversarial behaviour on the part of the environment.

In the following, it would be instructive to revisit the original research questions posed in [Chapter 1] and look at how these have been addressed.

The key research questions addressed in this dissertation are as follows:

- **RQ-1:** Can agent programs be annotated in a manner that permits the user to compute the post-conditions achieved by an agent at any point in its execution via design-time analysis?

  This thesis presents a novel scheme for annotating BDI agent programs with post-conditions in a manner that permits the user to analyze the effects achieved by agent execution purely through design-time analysis. This enables compliance analysis, goal analysis and the analysis of intermediate effects, amongst others. Ultimately, post-conditions play a vital role in generating robust adaptations, both in the case of agents and business processes.

- **RQ-2:** Can agent programs use adversarial game-tree search to compute optimal behaviour choices in dynamic uncertain environments?
9.1. Overview of the results presented

This thesis presents an approach to robust behaviour adaptation in BDI agents by leveraging post-condition annotations and adversarial game-tree search.

- **RQ-3:** Can a scheme for annotating agent programs with post-conditions provide the basis for a principled approach to merging agent programs?

The thesis also shows how post-condition annotation of BDI agent programs can lead to a principled scheme for merging such programs.

- **RQ-4:** Can business process execution be monitored by leveraging intended intermediate effects?

This thesis offers important advances in the way in which business process execution is monitored by defining a notion of *semantic conformance* (as opposed to the traditional notion of conformance — which we refer to as *structural conformance*).

- **RQ-5:** Can compensations for processes that deviate by failing to deliver the intended intermediate effects be computed efficiently?

The dissertation shows that it is possible to compute *compensations* (i.e., alternative process completions) for process instances that are found to be semantically non-conformant. The guiding principle is to compute compensations that deviate minimally from the structure mandated by the process design, that restore semantic conformance as early as possible while still ensuring that process goals are satisfied.

- **RQ-6:** Can adversarial game-tree search be used to compute robust process adaptations in dynamic uncertain environments?

This dissertation offers a scheme that used simple models of environment behaviour, coupled with game-tree search techniques to compute robust process compensations
9.2 Limitations of this work and directions for future research

There are a number of ways in which this work can be extended. These constitute interesting directions for future research.

A critical form of behaviour adaptation stems from the application of machine learning techniques. These have not been explored in this thesis. There are a few areas where machine learning techniques could be useful, such as to replace the heuristic function (e.g. Equation 4.1 and Equation 7.1), to replace the environment behaviour in dynamic, uncertain environments.

9.1.2 Contributions to the practitioner community

The contributions of this dissertation can translate quite quickly into practice. In the case of business processes, a semantic conformance checking module can be easily added to a business process engine to deliver sophisticated process monitoring functionality. In a similar vein, modules for computing process compensations as well as modules that perform game-tree search based analysis can be added to most existing process engines.

Lighter-weight versions of these techniques can also be of interest. One could avoid post-condition annotations in adversarial game-tree search by creating tables identifying actions that impede other actions. Similarly business process consultants who would prefer to perform purely informal analysis can use the techniques presented here to extract methodological guidelines to support very similar analysis.

The frameworks for agent programs presented here can also be translated into practice in very similar ways.

9.2 Limitations of this work and directions for future research

There are a number of ways in which this work can be extended. These constitute interesting directions for future research.

A critical form of behaviour adaptation stems from the application of machine learning techniques. These have not been explored in this thesis. There are a few areas where machine learning techniques could be useful, such as to replace the heuristic function (e.g. Equation 4.1 and Equation 7.1), to replace the environment behaviour
model in Chapter 8, or even the state update operator. The challenges of utilising machine learning techniques may include having large amount of past instances with annotated outcome values as well as the historical behaviour of the environment. Additionally, any statistical model may be only applicable to a very specific domain (e.g. for a specific process or environment). With more and more operational data collected in businesses, it is possible, in near future, that there are enough detailed data to support building machine learning or deep learning models.

There are a range of issue relating to game-tree search that have not been explored in this thesis. These include the use of other game-tree search techniques, probabilistic game-tree search, other variants of MCTS and so on. In the case of minimax search and its variants, the use of machine learning techniques in computing the evaluation function have not been explored.

The question of merging business process models (sometimes referred to as the process integration problem), in a manner similar to the merging of agent programs has not been explored in this thesis. However, the chapter on merging agent programs provides enough pointers to make this a potentially easy exercise.

Other possible formal notations for states and state update operators are not explored. The semantics of agent programs and processes are described using propositional logic. The extensions to first-order logic, default logic, and temporal logic will offer much greater expressiveness in different application domains. However, such extensions also increases the computation complexity, and potentially makes the approach less practical in runtime. The use of different formal languages also results in changing the state update operator and its computation complexity.

Other future research directions may include behaviour understanding (intention mining/recognition), discovering new behaviour and detecting undesirable behaviour in autonomous system, preventing autonomous system from reaching undesirable state,
etc.
Appendix A

Process Models

Process01

Model

Capability Library

T01 \{a\}

T02 \{b, c\}

T03 \{¬b\}

T04 \{¬c\}

Knowledge Base

\[ a \land \neg b \rightarrow c \]
\[ b \rightarrow a \]
\[ c \rightarrow a \]

**Process02**

**Model**

![Process Diagram]

**Capability Library**

\[ T01 \{a, b\} \]
\[ T02 \{¬b, ¬d\} \]
\[ T03 \{¬a, c\} \]
\[ T04 \{d\} \]

**Knowledge Base**

\[ a \land ¬b \rightarrow c \]
\[ c \land ¬d \rightarrow b \]
Process03

Model

Capability Library

T01 \{a, b, c, d, e, f, g\}

T02 \{¬b, ¬d, ¬f\}

T03 \{¬a, ¬c, ¬e, ¬g\}

T04 \{b, d, f\}

T05 \{a, c, e, g\}

T06 \{¬b\}

T07 \{¬f, ¬g\}

T08 \{¬a, ¬d\}

T09 \{¬c\}

T10 \{b, c\}
Knowledge Base

\[ a \land \neg b \rightarrow c \]

\[ c \land \neg d \rightarrow e \]

\[ e \land \neg f \rightarrow g \]

Process04

Model

Capability Library

T01 \{a, b, c, d\}

T02 \{-b, \neg d, \neg f, i\}

T03 \{-a, \neg c, \neg e\}

T04 \{b, d, f, \neg h\}

T05 \{a, c, e, g\}

T06 \{h, j, \neg l\}

T07 \{k, m\}
Knowledge Base

\[ a \land \neg b \rightarrow c \]
\[ c \land \neg d \rightarrow e \]
\[ e \land \neg f \rightarrow g \]
\[ \neg b \rightarrow a \]
\[ \neg d \rightarrow a \]
\[ h \rightarrow \neg a \]
\[ \neg j \rightarrow \neg a \]

Process05

Model

Capability Library

T01 \{a, b, c, d, e, f, g\}

T02 \{-b, -d, -f, i\}

T03 \{-a, -c, -e, -g, -i, j, k\}

T04 \{b, d, f, -h, -j, l, m\}
T05 \{a, c, e, g, i, \neg k, \neg m\}

T06 \{h, j, \neg l\}

T07 \{k, m\}

Knowledge Base

\[
\begin{align*}
a \land \neg b & \rightarrow c \\
c \land \neg d & \rightarrow e \\
e \land \neg f & \rightarrow g \\
\neg b & \rightarrow a \\
\neg d & \rightarrow a \\
h & \rightarrow \neg a \\
\neg j & \rightarrow \neg a
\end{align*}
\]

Process06

Model

Capability Library

T01 \{a\}
T02 \( \{c, d\} \)

T03 \( \{¬c, d\} \)

T04 \( \{¬d\} \)

T05 \( \{¬a, c\} \)

T06 \( \{¬c, d, ¬e\} \)

T07 \( \{e\} \)

T08 \( \{¬b, e\} \)

T09 \( \{¬a\} \)

Knowledge Base

\[
a \land ¬b \rightarrow c
\]

\[
¬a \rightarrow e \lor d
\]

Process07

Model
Capability Library

T01 \{a, d, e, f, g\}

T02 \{-f, h, i\}

T03 \{-a, \neg c, \neg e, \neg g, \neg i, j, k\}

T04 \{-d, f, \neg h, \neg j, l, m\}

T05 \{a, c, e, g, i, \neg k, \neg m\}

T06 \{-b, h, j, \neg l\}

T07 \{k, m\}

T08 \{l\}

T09 \{-b, \neg k, \neg m\}

T10 \{b, c, \neg d\}

Knowledge Base

\begin{align*}
a \land \neg b \land \neg m & \rightarrow c \\
c \land \neg d & \rightarrow e \\
e \land \neg f & \rightarrow g \\
\neg b \lor \neg d & \rightarrow a \\
h \land \neg j & \rightarrow \neg a \\
l & \rightarrow k \lor m
\end{align*}
Process08

Model

The process (simplified) for an Australian student visa application.

Capability Library

T01 \{level1, level2, level3, level4\}

T02 \{doc01, doc02, doc03, doc04, doc05, doc06\}

T03 \{doc08, doc09, doc10, doc11\}

T04 \{doc07\}

T05 \{onlineApplicationFormCompleted\}

T06 \{feePaidOnline\}

T07 \{157AFormCompleted\}
Knowledge Base

\[\text{level1} \rightarrow \neg \text{level2} \land \neg \text{level3} \land \neg \text{level4}\]
\[\text{level2} \rightarrow \neg \text{level1} \land \neg \text{level3} \land \neg \text{level4}\]
\[\text{level3} \rightarrow \neg \text{level1} \land \neg \text{level2} \land \neg \text{level4}\]
\[\text{level4} \rightarrow \neg \text{level1} \land \neg \text{level2} \land \neg \text{level3}\]
\[\neg \text{level2} \land \neg \text{level3} \land \neg \text{level4} \rightarrow \text{level1}\]
\[\neg \text{level1} \land \neg \text{level3} \land \neg \text{level4} \rightarrow \text{level2}\]
\[\neg \text{level1} \land \neg \text{level2} \land \neg \text{level4} \rightarrow \text{level3}\]
\[\neg \text{level1} \land \neg \text{level2} \land \neg \text{level3} \rightarrow \text{level4}\]
\[\text{online} \rightarrow \neg \text{offline}\]
\[\text{offline} \rightarrow \neg \text{online}\]
\[\text{offline} \rightarrow \text{inPerson} \lor \text{byPost}\]
Process09

Model


Capability Library

T01 \{\text{requested}\}

T02 \{\text{available}, \overline{\text{available}}\}

T03 \{\text{rejected}\}

T04 \{\text{infoChecked}\}

T05 \{\text{valid}, \overline{\text{valid}}\}

T06 \{\text{open}\}

T07 \{\text{activate}\}
Knowledge Base

The knowledge base is empty because the effect of the tasks and the domain knowledge do not require any extra rules in the knowledge base.

Process10

Model

Capability Library

T01 \{invoiceIn\}

T02 \{invoiceChecked\}

T03 \{newEntryCreated\}

T04 \{invoiceDetailInserted\}

T05 \{customerDetailInserted\}

T06 \{mismatchChecked\}

T07 \{invoiceBlocked\}

T08 \{invoiceOut\}
Knowledge Base

[empty]

The knowledge base is empty because the effect of the tasks and the domain knowledge do not require any extra rules in the knowledge base.
References


of dMARS. Technical Note 72, Australian Artificial Intelligence Institute, nov 1998.


on Autonomous agents and multiagent systems - AAMAS ’06, AAMAS ’06, page 161, New York, NY, USA, 2006. ACM.


