ARTS: Agent-Oriented Robust Transactional System

Mingzhong Wang

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

September 2009

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia

Produced on archival quality paper
To my parents, Wang Guangrong and Zhang Yuzhu, and my wife, Hu Rui, who bestow me privileges of focusing on study and research.
Abstract

Internet computing enables the construction of large-scale and complex applications by aggregating and sharing computational, data and other resources across institutional boundaries. The agent model can address the ever-increasing challenges of scalability and complexity, driven by the prevalence of Internet computing, by its intrinsic properties of autonomy and reactivity, which support the flexible management of application execution in distributed, open, and dynamic environments. However, the non-deterministic behaviour of autonomous agents leads to a lack of control, which complicates exception management in the system, thus threatening the robustness and reliability of the system, because improperly handled exceptions may cause unexpected system failure and crashes.

In this dissertation, we investigate and develop mechanisms to integrate intrinsic support for concurrency control, exception handling, recoverability, and robustness into multi-agent systems. The research covers agent specification, planning and scheduling, execution, and overall coordination, in order to reduce the impact of environmental uncertainty. Simulation results confirm that our model can improve the robustness and performance of the system, while relieving developers from dealing with the low level complexity of exception handling.

A survey, along with a taxonomy, of existing proposals and approaches for building robust multi-agent systems is provided first. In addition, the merits and limitations of each category are highlighted.

Next, we introduce the ARTS (Agent-Oriented Robust Transactional System) platform which allows agent developers to compose recursively-defined, atomically-handled tasks to specify scoped and hierarchically-organized exception-handling plans for a given goal. ARTS then supports automatic selection, execution, and monitoring of appropriate plans in a systematic way, for both normal and recovery executions. Moreover, we propose multiple-step backtracking, which
extends the existing step-by-step plan reversal, to serve as the default exception handling and recovery mechanism in ARTS. This mechanism utilizes previous planning results in determining the response to a failure, and allows a substitutable path to start, prior to, or in parallel with, the compensation process, thus allowing an agent to achieve its goals more directly and efficiently. ARTS helps developers to focus on high-level business logic and relaxes them from considering low-level complexity of exception management.

One of the reasons for the occurrence of exceptions in a multi-agent system is that agents are unable to adhere to their commitments. We propose two scheduling algorithms for minimising such exceptions when commitments are unreliable. The first scheduling algorithm is trust-based scheduling, which incorporates the concept of trust, that is, the probability that an agent will comply with its commitments, along with the constraints of system budget and deadline, to improve the predictability and stability of the schedule. Trust-based scheduling supports the runtime adaptation and evolvement of the schedule by interleaving the processes of evaluation, scheduling, execution, and monitoring in the life cycle of a plan. The second scheduling algorithm is commitment-based scheduling, which focuses on the interaction and coordination protocol among agents, and augments agents with the ability to reason about and manipulate their commitments. Commitment-based scheduling supports the refactoring and parallel execution of commitments to maximize the system’s overall robustness and performance. While the first scheduling algorithm needs to be performed by a central coordinator, the second algorithm is designed to be distributed and embedded into the individual agent.

Finally, we discuss the integration of our approaches into Internet-based applications, to build flexible but robust systems. Specifically, we discuss the designs of an adaptive business process management system and of robust scientific workflow scheduling.
Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,
(ii) due acknowledgement has been made in the text to all other material used,
(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

________________________________________
Mingzhong Wang
September 2009
Publications

During the course of the research, a number of refereed journal and conference articles have been published based on the work presented in this thesis. They are listed here for reference.

Journal Papers


Conference Papers

Book Chapter

Acknowledgements

Due to the invaluable help from a number of people, I gradually mature in the research world along with sweet memory in three and half years of student life at the University of Melbourne. My gratitude for them starts before the thesis begins, and will never end.

First, I would like to thank my supervisor, Kotagiri Ramamohanarao (Rao) for his exceptional support and guidance on my research. Rao has proficient knowledge in many fields and can always give me constructive comments and advice. His encouragement of viewing problems from different and practical perspectives helps me to broaden my views without sliding into fiction.

I also want to thank James Bailey, Amy Unruh, and Jinjun Chen. All of them have provided insightful suggestions and advice regarding my research. Specifically, James and Amy led me to the study of robust multi-agent systems, and Jinjun inspired me to apply our models to workflow-based systems.

I am grateful for the University of Melbourne for generously awarding me MIRS and MIFRS scholarships which relieved me from financial stresses. I also appreciate the Department of Computer Science and Software Engineering, the School of Engineering, and the ARC Discovery project of Transaction-Oriented Computational Models for Multi-Agent Systems for sponsoring my conference trips.

I wish to acknowledge the support from the administrative staff in the department and the faculty. Especially, I wish to thank Pinoo Bharucha who arranged all my meeting appointments with Rao in the past years.

Personally I wish to thank the fantastic people at the CSSE department, especially the colleagues and friends of room 4.31 and 4.18, including Yi Li, Robert Shelton, Shane Culpepper, Xiaonan Ji, Yuye Zhang, and Yong Zhen Guo, and William Webber, just to name a few. They make my time extremely enjoyable. In addition, I wish to express my heartfelt appreciation for William Webber whose careful proofreading assures the quality of this thesis.

Finally, I wish to show my sincere thanks to my family for their continuous and unselfish support and encouragement throughout my study. They have taken care of most distractive chores in daily life, and have shared all the stresses I have encountered during the study. I hope they can share my happiness together with me now.
# Contents

Abstract v  
Declaration vii  
Publications ix  
Acknowledgements xi  
List of Algorithms xvii  
List of Figures xix  
List of Tables xxi  
Glossary of Notation xxiii

1 Introduction 1  
1.1 Background and Problem Statement 2  
1.1.1 Internet Computing and Multi-agent Systems 2  
1.1.2 Autonomy and Manageability 3  
1.1.3 Technical Problems 4  
1.2 Research Objective 5  
1.3 Basic Concepts 6  
1.3.1 BDI Agent Model 6  
1.3.2 System Robustness 8  
1.3.3 Transactions 9  
1.3.4 Trust and Reputation 11  
1.4 Overview of the Thesis 12  
1.5 Contributions of the Thesis 13  

2 A Survey of Robust Multi-Agent Systems 17  
2.1 Introduction 17  
2.2 Taxonomy of Robust Multi-Agent Systems 18  
2.3 Critical Review of Existing Approaches for Robust Multi-Agent Systems 20  
2.3.1 Exception Avoidance 20  
2.3.2 Stateless Exception Recovery 21  
2.3.3 Transactional Exception Recovery 21  
2.3.4 Non-Transactional Exception Recovery 25  
2.3.5 Separation of Exception Recovery Knowledge 27  
2.4 Related Work 30  
2.5 Summary 31
### 3 Specification and Organization of ARTS Model

| Section                                                                 | Page |
|                                                                        |      |
| 3.1 Introduction                                                      | 33   |
| 3.1.1 A Motivating Example                                            | 34   |
| 3.1.2 Overview of the ARTS(Agent-oriented Robust Transactional System) Model | 36   |
| 3.2 The Task Model in ARTS                                            | 37   |
| 3.2.1 Modelling a Single Task                                         | 38   |
| 3.2.2 Modelling Task Composition                                      | 43   |
| 3.3 The ARTS Model                                                    | 47   |
| 3.3.1 Exception-Handling Knowledge in ARTS                            | 47   |
| 3.3.2 Generating a Dependency Set from Plans                          | 51   |
| 3.3.3 Organizing the Dependency Set                                   | 54   |
| 3.3.4 Execution of Plans                                              | 59   |
| 3.4 Discussion and Conclusion                                          | 61   |

### 4 Multiple-step Backtracking in BDI Transactional Agent Execution

| Section                                                                 | Page |
|                                                                        |      |
| 4.1 Introduction                                                       | 64   |
| 4.2 Motivation                                                         | 66   |
| 4.3 Multi-step Backtracking Recovery for Exceptions                    | 69   |
| 4.3.1 Principles of Agent Execution                                   | 69   |
| 4.3.2 Key Concepts for “Jumping”                                       | 71   |
| 4.3.3 Recovery Mechanism and Algorithm                                 | 74   |
| 4.3.4 Relation to Existing Approaches                                  | 77   |
| 4.4 Transactional Execution of BDI agents                              | 78   |
| 4.4.1 Transactional Encapsulation of Plan Execution                    | 79   |
| 4.4.2 Transaction Management                                           | 81   |
| 4.5 Experiments and Evaluations                                        | 84   |
| 4.5.1 Simulation Results                                               | 87   |
| 4.6 Discussion and Conclusion                                          | 91   |

### 5 Robust Scheduling and Runtime Adaptation of Multi-agent Plan Execution

| Section                                                                 | Page |
|                                                                        |      |
| 5.1 Introduction                                                       | 94   |
| 5.2 Trust Model and Problem Definition                                 | 96   |
| 5.2.1 Trust-aware Platform for Agents                                  | 96   |
| 5.2.2 Problem Definition                                               | 97   |
| 5.3 Trust-based Plan Management                                        | 99   |
| 5.3.1 Plan Evaluation                                                  | 100  |
| 5.3.2 Plan Scheduling                                                  | 107  |
| 5.3.3 Execution Monitoring                                             | 109  |
| 5.4 Experiments and Evaluations                                        | 113  |
| 5.4.1 Experiment design                                                | 113  |
| 5.4.2 Result of plan scheduling                                       | 115  |
| 5.4.3 Result of plan evaluation                                       | 116  |
| 5.5 Related Work                                                       | 118  |
| 5.6 Discussion and Conclusion                                          | 119  |

### 6 Reasoning Intra-Dependency in Commitments for System Robustness and Performance

| Section                                                                 | Page |
|                                                                        |      |
| 6.1 Introduction                                                       | 122  |
| 6.2 Background and Motivation                                          | 124  |
| 6.3 Intra-Commitment Dependency                                         | 126  |
| 6.3.1 Commitment Classification                                        | 126  |
| 6.3.2 Syntax of Commitments                                            | 131  |
| 6.3.3 Semantics of Commitments                                         | 132  |
# List of Algorithms

3.1 substitute($t$) ......................................................... 53  
3.2 getTaskDependencySet($t$) ........................................ 54  
3.3 getSimpleDependencySet($t$) ....................................... 55  
3.4 getPlanDependencySet($p$) ......................................... 56  
4.1 Execution flow of "jump" ............................................. 74  
4.2 Pseudo-code for the travel agent ..................................... 83  
5.1 goalToNonSelectivePlans() ......................................... 102  
5.2 findCriticalPath($p$) ................................................ 104  
5.3 findMostPromisingPlan($\mathcal{P}$) .................................... 107  
5.4 scheduleWithGA($p$) .................................................. 110  
5.5 findSubstitution($I$, $t_i$) .......................................... 112  
6.1 Debtor’s commitment manipulation for concurrent execution .. 140
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Architecture of agent transactional platform</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>PRS-style structure for a BDI agent (Adapted from [42])</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Event driven and means-end reasoning for BDI agent execution</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>Overview of Chapter 3 to Chapter 6 in BDI execution process</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Hierarchical taxonomy of robust multi-agent systems</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Related work on Taxonomy of Exception Handling (Adapted from [120])</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>A Surgery preparation task in a hospital environment</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Interface of a single task</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>State transition graph for a single task</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>A procedure for the Surgery Manager</td>
<td>52</td>
</tr>
<tr>
<td>3.5</td>
<td>Dependency graph with hierarchical relations</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>Travel arrangement agent</td>
<td>68</td>
</tr>
<tr>
<td>4.2</td>
<td>Choice point stack for the travel example</td>
<td>72</td>
</tr>
<tr>
<td>4.3</td>
<td>Different recovery strategy for different type of path coordination</td>
<td>73</td>
</tr>
<tr>
<td>4.4</td>
<td>Integration of multi-step backtracking and BDI agent execution</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Concurrent execution after jump</td>
<td>81</td>
</tr>
<tr>
<td>4.6</td>
<td>Probability of system failure with various task exception probabilities</td>
<td>88</td>
</tr>
<tr>
<td>4.7</td>
<td>Time save ratio with various task exception probabilities</td>
<td>89</td>
</tr>
<tr>
<td>4.8</td>
<td>System failure rate with various recovery rates</td>
<td>90</td>
</tr>
<tr>
<td>4.9</td>
<td>Time Save Ratio with various recovery rates</td>
<td>90</td>
</tr>
<tr>
<td>4.10</td>
<td>System behaviour with various resource usage densities</td>
<td>91</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution flow of Coordinator</td>
<td>100</td>
</tr>
<tr>
<td>5.2</td>
<td>Example for task slack time and critical path</td>
<td>106</td>
</tr>
<tr>
<td>5.3</td>
<td>Chromosome encoding for DAG</td>
<td>108</td>
</tr>
<tr>
<td>5.4</td>
<td>System success probability with various constraint levels</td>
<td>115</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison of estimation by plan evaluation and simulated outcome by plan scheduling</td>
<td>117</td>
</tr>
<tr>
<td>6.1</td>
<td>Scenario of purchase protocol</td>
<td>125</td>
</tr>
<tr>
<td>6.2</td>
<td>Temporal relationship for the pre- and post- conditions of a commitment</td>
<td>128</td>
</tr>
<tr>
<td>6.3</td>
<td>Concurrent execution of guarded commitments</td>
<td>141</td>
</tr>
<tr>
<td>6.4</td>
<td>Time usage ratio with various probability of a commitment being SUC</td>
<td>149</td>
</tr>
<tr>
<td>7.1</td>
<td>Travel workflow example</td>
<td>161</td>
</tr>
<tr>
<td>8.1</td>
<td>Relations among transactions, workflows, and multi-agent systems</td>
<td>164</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Constraints of a successful task execution</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Constraints of the payment task</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Decomposition of the surgery preparation task</td>
<td>53</td>
</tr>
<tr>
<td>3.4</td>
<td>The dependency set for $\text{start}(t)$ of Surgery Manager</td>
<td>57</td>
</tr>
<tr>
<td>3.5</td>
<td>The dependency set for $\text{start}(t) \cdot \text{Compensated}(t_3)$ of Surgery Manager</td>
<td>57</td>
</tr>
<tr>
<td>3.6</td>
<td>The dependency set for $\text{start}(t) \cdot \text{Compensated}(t_3) \cdot \text{Aborted}(t_{13})$ of Surgery Manager</td>
<td>57</td>
</tr>
<tr>
<td>3.7</td>
<td>The dependency set for $\text{start}(t) \cdot \text{Compensated}(t_6)$ of Surgery Manager</td>
<td>57</td>
</tr>
<tr>
<td>3.8</td>
<td>The dependency set for $\text{start}(t) \cdot \text{Compensated}(t_6) \cdot \text{Aborted}(t_4)$ of Surgery Manager</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Plans for travel agent</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>Key parameters in the experiments of multi-step backtracking</td>
<td>86</td>
</tr>
<tr>
<td>5.1</td>
<td>Monitoring rules of Coordinator</td>
<td>110</td>
</tr>
<tr>
<td>5.2</td>
<td>Parameter settings for the genetic algorithm</td>
<td>114</td>
</tr>
<tr>
<td>6.1</td>
<td>Commitments in each state of Interaction Protocol</td>
<td>125</td>
</tr>
<tr>
<td>6.2</td>
<td>Properties of commitment safety levels</td>
<td>138</td>
</tr>
<tr>
<td>6.3</td>
<td>Cost and probability of success for NetBill example</td>
<td>145</td>
</tr>
<tr>
<td>6.4</td>
<td>Key parameters in the experiments of concurrent scheduling for strictly unordered commitments</td>
<td>148</td>
</tr>
</tbody>
</table>
Glossary of Notation

\[ A \] a set of agents
\[ a, a_0, a_1, \ldots \] a single agent
\[ P \] a set of agent plans
\[ p, p_0, p_1, \ldots \] a single agent plan
\[ T \] a set of tasks
\[ t, t_0, t_1, \ldots \] a single task
\[ t^- \] the compensation task of \( t \)
\[ G \] a set of agent goals
\[ g, g_0, g_1, \ldots \] a single agent goal
\[ E \] a set of events
\[ e, e_0, e_1, \ldots \] a single event
\[ B \] the agent belief set
\[ b, b_0, b_1, \ldots \] an agent belief
\[ S \] a set of computational states
\[ s, s_0, s_1, \ldots \] a computational state
\[ s' \] the state of task \( t \)
\[ s(x), s \models x \] predicates \( x \) is true in \( s \)

\( Pre \) precondition set
\( Post \) post-condition set
\( M \) maintenance condition set
\( Inv \) invariant condition set
\( parent \succ child \) the parent-children relationship in a hierarchy
\( ancestor \succ descendant \) the ancestor-descendant relationship in a hierarchy
\( \epsilon \) null reference
∅ empty sequence or empty set
; sequence composition
, selection composition
∥ parallel composition
\textit{depend}(t, e) \quad \text{the triggering of the task } t \text{ depends on the occurrence of the event } e
\textit{DEP}(T \times E) \quad \{ \text{depend}(t, e) \mid t \in T \land e \in E \}, \text{ Cartesian product for the dependency relationship between tasks and events}
\bigwedge(T_1 \times T_2 \times \ldots \times T_n) \quad \{ t_1 \land t_2 \land \ldots \land t_n \mid t_i \in T_i \}, \text{ Cartesian product for task sets}
\bigcup_{1 \leq i \leq n} Set_i \quad \text{brief notion of set union operation}

P(e) \quad \text{the probability of event } e \text{ to take place}

\mathcal{A}_i \quad \text{agent set whose elements are capable of achieving task } t_i
a_{ij} \quad \text{the } j\text{th member of } \mathcal{A}_i
time_{ij} \quad \text{the execution time of } a_{ij} \text{ to achieve } t_i
cost_{ij} \quad \text{the execution cost of } a_{ij} \text{ to achieve } t_i
trust_{ij} \quad \text{the trust of } a_{ij} \text{ to achieve } t_i
B \quad \text{the cost constraint (budget) for a goal}
D \quad \text{the time constraint (deadline) for a goal}
time(i) \quad \text{the execution time of } t_i
cost(i) \quad \text{the execution cost of } t_i
trust(i) \quad \text{the trust of achieving } t_i
Cost(p) \quad \text{the execution cost of the plan } p
Time(p) \quad \text{the execution time of plan } p
\bar{x} \quad \text{the median value of the variable } x
MAD(x) \quad \text{the median absolute deviation of the variable } x

CC(x, y, \Phi, \Psi), CC(\Phi, \Psi) \quad \text{conditional commitment}
C(x, y, \Psi), C(\Psi) \quad \text{unconditional commitment}
\sigma = < s_0, s_1, \ldots > \quad \text{execution path}
CC(\Phi \cap \Psi) \quad \text{strictly ordered commitment}
CC(\Phi \cup \Psi) \quad \text{weakly ordered commitment}
Publications

\[ CC(\Phi \parallel \Psi) \quad \text{strictly unordered commitment} \]
\[ CC(\Phi \odot \Psi) \quad \text{weakly unordered commitment} \]
\[ \Phi, \Phi_i \text{ and } \Psi, \Psi_i \quad \text{state formula} \]
\[ \phi, \phi_1, \phi_2, \ldots \quad \text{commitment-formula} \]
Chapter 1

Introduction

The prevalence of the Internet is enabling computation models to move rapidly, from the conventional individual standalone computer system, to a situation in which the real power of computers is realised through distributed, open, and dynamic systems [97]. Unlike in the traditional centralized and static environment, computing entities can no longer be restricted to merely performing routine activities without considering the non-deterministic impact of the environment. In other words, they need to have a certain degree of autonomy to deal with uncertainty. However, the character of autonomy also makes the coordination and management of these computing entities more complicated. As a result, a new computing paradigm is required to keep the tradeoff between the autonomy and manageability.

Software agents have the ability to provide autonomous and reactive behaviour, and to support decomposition and abstraction of functionality, making the agent model useful in analyzing, designing, and implementing complex software systems, especially in distributed, open, and dynamic environments [77]. In order to push agent technology into the mainstream of software development, various agent architectures [174] and agent programming languages (APLs) [14, 15] have been proposed.

Although current research has made great strides, the desire for a powerful yet easy-to-use mechanism for maintaining system robustness and reliability, with respect to the correct execution of agent actions even in the presence of abnormalities, still remains unfulfilled. As a result, while some aspects of an agent language may be high-level, developers still must consider low-level details of disturbances, failure, or uncontrolled interactions between the agents in a system.
Because agents are often used in an open environment where they do not have full control of resources, and usually run autonomously in distributed and parallel fashion, it is very difficult for developers to correctly address these complexities. Thus, high-level support for robustness and reliability is one of the keys for agent technologies to become successful.

This thesis presents a thorough study of processing stages in reactive agent systems, and proposes several mechanisms and models to extend and enhance them, to improve system robustness and reliability while reducing the complexity of system development and implementation for system/application developers.

1.1 Background and Problem Statement

1.1.1 Internet Computing and Multi-agent Systems

Internet computing is highly network-integrated computing on existing computers, on the grand scale of the Internet [119]. Emerging Internet uses, such as peer-to-peer (P2P) [109], web services [116], grid computing [11], and cloud computing [20, 41], provide both a glimpse of, and the impetus for, evolving the Internet into a distributed computing platform unprecedented in scale [106].

The core concept of Internet computing is the virtualization of resources that a node may want to share, such as data, computation cycles, or storage. Therefore, available resources can be aggregated dynamically to complete a given task, such as executing a weather forecast application, which usually exceeds the capability of a single node. For the Internet computing model to be successful and effective, it should have the following features: load balancing, by distributing work to different nodes; scalability, by allowing the dynamic participation of computing nodes; reliability and availability, by means of resource redundancy (for example, adding backup computing nodes); and high system performance, by possible parallel execution.

A key to the success of Internet computing is the self-management and self-organization of resources, in order to manage system size and complexity. This form of computing is termed autonomic computing [55, 71, 83, 84]. In autonomic computing, each resource behaves autonomously in managing its own internal states and its interactions with external resources, as well as the environment, to achieve its design goals.
The field of multi-agent systems (MASs), which studies cooperation and coordination between distributed and autonomous entities, provides a practical approach to achieving the goal-driven and reactive features of autonomic computing systems [149]. Multi-agent systems, which represent each autonomous resource as an agent, serve as a good metaphor to model autonomous systems, thus being an important approach to accomplish the vision of Internet computing [141]. In fact, the core concept of the agent model is to tackle the complexity of distributed, open, and dynamic environments by rendering autonomy to participating entities. Therefore, the agent model has an inborn incentive to address the desire for autonomous management in Internet computing.

1.1.2 Autonomy and Manageability

Although autonomy makes an agent more flexible in adapting itself to dynamic environments, it also makes managing the system more complicated. In centrally managed systems, the composing participants reliably execute the instructions of the system manager. In some sense, the manager has complete control over every part of the system. However, the control is weakened, if not lost, in autonomous systems.

In autonomous systems, each agent has its own goal to achieve and can independently decide the corresponding actions to perform. Agents may work together not only cooperatively, but also competitively. Some agents may even behave dishonestly and sometimes maliciously to pursue their own interest and benefit; this is the price of autonomy. The behaviour of an autonomous agent is no longer completely predictable by the system manager. Consequently, the possible occurrence of exceptions and failures becomes even harder to enumerate and handle.

The complexity created by autonomy brings new challenges for managing and organizing autonomous individuals. If these difficulties cannot be harnessed, Internet computing would be too fragile and its application domain would become very restrictive.

To address these challenges, the thesis proposes several models and mechanisms, based on studying and extending reactive agent models to keep the tradeoff between the autonomy and manageability of the system. The goal of the study is to minimize the side effects of autonomy, thus improving system robustness and reliability.
1.1.3 Technical Problems

To achieve an agent's goal, the corresponding life cycle of the agent can be divided into three stages:

- Planning and scheduling. For an expected goal, the system needs to select and compose operational tasks, and to assign resources as required.

- Execution. The specified actions are executed to achieve the goal. Exceptions and failures may occur, and need to be resolved, during this stage.

- Coordination. Generally, a goal is not isolated from other parts of the system. For example, it may be a subgoal of others, or share resources with others. Therefore, communication among agents is necessary to coordinate the execution.

In Internet computing and multi-agent applications, these stages are carried out in an interleaved fashion. Within each stage, different approaches can be applied to improve system robustness and reliability.

In the planning and scheduling stage, various tasks and resources can be composed in different ways to achieve a goal. Of all possible outcomes, the system needs to select the one with the highest probability of success. The planner and scheduler should address not only the large number of possible choices, but also the ever-evolving execution environment, which places the full predictability of the system out of reach. Therefore, the challenge lies in the combinatorial explosion of choices and the uncertainty of dynamic and open environments.

During the execution stage, each well-planned actions may encounter exceptions, since the presumed environment may unpredictably evolve into different, even adverse, states. As a result, replanning and rescheduling need to be performed as frequently as possible, to keep pace with environment changes. However, the computational cost of frequent planning and scheduling may not be affordable. Therefore, the challenge is to keep a trade-off between plan execution and the reconsideration of plan validity [174].

Since individual agents run autonomously and concurrently, they need to be synchronized and coordinated to enforce their organizational structure. In the coordination stage, the dependence, as well as the cause-and-effect relationship, among agents needs to be tracked, along with environmental updates. The challenge comes in the control of concurrency, and especially in the coordinated recovery from exceptions.
1.2 Research Objective

In summary, the common theme of the challenges faced is the uncertainty in open and dynamic environments composed of autonomous resources. Without systematic support, enormous programming efforts are required for specifying event handlers for each possible situation. In addition, the system may also need to track every detail of the environment, which is expensive in both time and computation. Therefore, practical mechanisms are required to maintain the manageability of autonomous systems.

1.2 Research Objective

The objective of this research is to integrate mechanisms for the intrinsic support of concurrency control, recovery, and robustness into multi-agent systems, thus helping developers to design and implement complex systems. The research covers the stages of planning and scheduling, execution, and coordination, to reduce the impact of environmental uncertainty.

At its conclusion, we make several contributions to the body of research in Internet computing and multi-agent research. The goal is to produce an intermediate transactional layer, as shown in Figure 1.1, which will provide scheduling, runtime execution monitor and adaptation, concurrency control, and exception-handling facilities for agent execution, between the underlying agent programming platforms and the upper multi-agent applications, to support the robustness and reliability of multi-agent systems.

Figure 1.1: Architecture of agent transactional platform
Within this platform, programmers are relieved from considering low-level details of concurrency control and failure recovery. They only need to specify the desired relationships among the agents declaratively, using the concise programming constructs provided by the transactional layer. The transactional layer executes using the underlying agent platforms, and provides logging, locking, and exception-handling facilities for their execution.

1.3 Basic Concepts

In this section, we introduce some basic concepts on which this thesis is based.

1.3.1 BDI Agent Model

Agents are persistent entities that can perceive, reason, and act in some environment. [174] defines an agent as:

“a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives”.

The definition emphasizes four main features of an agent:

- **autonomy**: agents are able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behaviour;
- **reactivity**: agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it, in order to satisfy their design objectives;
- **pro-activeness**: agents are able to exhibit goal-directed behaviour by taking the initiative, in order to satisfy their design objectives;
- **social ability**: agents are capable of interacting with other agents (and possibly humans), in order to satisfy their design objectives.

Among various types of agent architectures, **BDI (Belief, Desire, Intention)** [61, 130, 131] is probably the most mature and accepted model. Most BDI platforms using the PRS (Procedural Reasoning System) [42] style share the following three features.

- An agent contains four key data structures, as shown in Figure 1.2. **Beliefs** are the informational state representing what an agent knows about itself and the world; these
beliefs may be incomplete or even incorrect. Goals are the motivational state and correspond to what the agent wants to achieve. Plans represent the procedural knowledge about how to achieve a certain goal or react to a specific situation. Intentions are plans selected for execution and represent the deliberative state of an agent.

- The execution of an agent is event driven. Plans, usually denoted as $event \leftarrow preconditions \mid action\_sequence$, are defined to react to a certain event, which can be internal modifications to its goals and beliefs or external changes of environment. After the event is triggered, the preconditions will be tested before action_sequence can be chosen for execution. Because events can occur non-deterministically, plans are executed reactively.

- The execution path to achieve a goal of an agent is generated by means-end reasoning. That is, the goal is treated as an initial event triggering a corresponding plan to run. The action_sequence of the plan may contain primitive actions as well as sub-goals. All sub-goals will be treated in turn as events that trigger sub-plans to run. This process continues recursively until the actions in all sub-plans are primitive or atomic.

![Figure 1.2: PRS-style structure for a BDI agent (Adapted from [42])](image)

The execution process of a BDI agent can be abstractly depicted in Figure 1.3. This process is described by Rao and Georgeff [131], and applied directly or indirectly by many agent development platforms.
The plan matcher retrieves an event from the queue, and searches through the plan library to find the set of plans which can handle this event in that particular situation (determined by its beliefs). There might be more than one suitable plan found, and the plan selector will choose one of them and push it onto the intention stack. Finally, the intention is executed, which results in internally updating the BDI state, including beliefs, desires and intentions, or operating externally to sense and change the environment. Some of these actions also signal new events to the agent.

To fulfill the power and benefits of the BDI model, many APLs, as well as integrated development environments, have been proposed, including JACK [49], 3APL [35], Jason [16], and Jadex [125], to name a few.

### 1.3.2 System Robustness

Robustness is usually recognized as the quality of being able to withstand stresses, pressures, or changes in procedure or circumstance [134]. Robustness requires a system to be capable of coping well with variations, which may be unpredictable, in its operating environment, with minimal damage, alteration, or loss of functionality.

Robustness is a key requirement for any practical application. However, to develop robust multi-agent systems is challenging, because both the participating agents and their executing environment are often unpredictable, and hence can be unreliable and out of control [88]. Therefore, robust performance is a particular need of agent systems.

Traditionally, there are two major fault tolerant techniques to improve system robustness and
reliability [6, 66]:

- Redundancy-based systems. Replicated components are used as a replacement in case of failure, with the assumption that faults in replicated components are not co-related.
- Transactional systems. The system is executed as a transaction with all-or-nothing semantics.

Many hardware systems, such as disk arrays and clusters, are designed and constructed using redundancy-based approaches, because of their simplicity. In service-oriented computing and multi-agent systems, various services and agents with the same capability can also be viewed as providing a kind of redundancy. However, redundancy-based approaches are usually expensive to implement and to maintain [66].

Transactions are widely applied in database and workflow systems to guarantee consistent and durable execution with the support of concurrency control. Although the implementation of transactions applies the technique of redundancy of transaction logs, it models the system semantics from a higher-level view of atomic execution. The next section will provide a more detailed introduction to transactions.

The thesis will investigate and integrate both redundancy-based and transactional approaches to improve the robustness of the system.

1.3.3 Transactions

With system-level support of concurrency control and failure recovery, a transaction framework enables programmers to focus solely on designing and developing application’s logic, instead of considering complicated and error-prone lower-level details of action interactions.

Traditionally, a transaction is a transformation of state which satisfies the ACID (Atomicity, Consistency, Isolation and Durability) properties [66, 128], which prevent inconsistency and integrity problems.

- Atomicity means the transition from the initial state to the result state has no observable intermediate states. Atomic execution means that either all or none of the transaction’s operations are performed.
- Consistency means that a transaction should only produce consistent results satisfying all related integrity constraints; otherwise it aborts.
• Isolation means that a transaction executes as if it were running in single-user mode, that is, there is no observable interference from other concurrently running transactions. Consequently, the property of serializability, which requires the outcome of the concurrent transaction execution to be equal to that of sequential execution, becomes the major correctness criterion for the execution of concurrent transactions.

• Durability means that the results of a successfully committed transaction must become permanent, surviving any subsequent failure.

By definition, each transaction has exactly one of two outcomes: committed or aborted. Once a transaction commits, its effects can only be altered by running further transactions. Such post facto transactions are called compensating transactions [65].

As simple they are, traditional transactions are not suitable to model processes with complex structures or long-lived activities because of their flat structure. Therefore, a number of extended or advanced transaction models (ATMs) are proposed [99, 128, 170]. They not only extend the internal transaction structure, but also ease transaction ACID requirements by selective relaxation of atomicity or isolation and may not require serializability as a global correctness criterion. They frequently use inter-transaction execution dependencies that constrain the scheduling and execution of component transactions [133]. Most of them use some form of compensation.

Although transaction models are a powerful abstraction for concurrent computing, their direct adoption into multi-agent systems is likely to cause problems with long-running transactions (one transaction can hold resources exclusively for a long time) or cascade rollback (one exception causes all work to be undone) [124, 167]. What is more, due to the dynamic and non-deterministic features of the agent environment, recovery on the reverse order of agent execution history will probably become meaningless or infeasible in many situations.

However, transactions still provide invaluable concepts and features (e.g., failure atomicity, concurrency control, nested structure, compensation, and forward recovery) which can be used in components that form a part of an agent system, especially for the problem of error handling and recovery.
1.3 Basic Concepts

1.3.4 Trust and Reputation

In Internet computing, an application usually needs the contributions of various services or agents. However, since individuals run autonomously to pursue and maximise their own interest, the delegator (the application) faces the risk that the delegatee (the service agent) may not behave as it has advertised or agreed to. For example, the delegatee may delay the outcome because of low processing efficiency, or deliver unsatisfactory results with a low quality standard. Even worse, the delegatee might quit the system as soon as it gets paid, disregarding the contract with the delegator. Without the notion of trust, risk is unmanageable and financial transactions between agents simply cannot take place [73].

Therefore, trust becomes a fundamental concern in multi-agent systems. Trust is a measure composed of many different attributes, including reliability, dependability, honesty, truthfulness, security, competence, and timeliness. In particular, trust is a belief an agent has that the other party will do what it says it will, given an opportunity to defect to get higher payoffs [129].

Since trust is a subjective belief of confidence or a feeling of certainty, trust models are usually constructed based on reputation systems and past behaviour. Reputation can be considered as a collective measure of trustworthiness (in the sense of reliability) based on referrals or ratings from members in a community [21, 78]. The common approach is to let members rate each other according to their experiences of past interactions, and then use some aggregated ratings about a given member to derive a trust or reputation score, which can assist others in deciding whether or not to trust that member in the future. Details about algorithms for reputation management systems can be found in [1, 17], and [80] presents a solution in distributed and peer-to-peer environments. The use of trust and reputation has a positive effect on market quality since it encourages good behaviour. Reputation systems are already used successfully in commercial online applications, such as eBay.

The concepts of trust and reputation, though related, are not identical. However, for the purpose of the thesis, we will use them interchangeably. In this thesis we also assume the availability of trustworthiness values for service providers, and the thesis does not include the building of trust models.
1.4 Overview of the Thesis

The thesis presents a study on the research topic of building robust multi-agent systems. The main content of the thesis is organized into eight chapters, focusing on different aspects of the topic.

Chapter 1 describes the problems and challenges of managing autonomous systems in open and dynamic environments, and introduces related concepts. Chapter 2 gives a detailed literature review of various proposals and models for the construction of robust multi-agent systems.

Chapter 3 presents the ARTS (Agent-oriented Robust Transactional System) model, which applies transaction concepts to provide agent developers with high-level support for agent system robustness and reliability. It allows agent developers to compose recursively-defined, atomically-handled tasks.

Chapter 4 proposes a multi-step backtracking model to achieve exception handling in agent systems. It maintains a tradeoff between replanning, which discards all alternatives found in previous execution, and step-by-step backtracking, which iterates back through all previous selection decisions.

Chapter 5 addresses the scheduling issue in multi-agent systems. It incorporates the concept of trust, which indicates the probability that an agent will comply with its commitments, into scheduling, thus improving the predictability and stability of the schedule. It also supports the runtime adaptation and evolution of the schedule to deal with exceptions during execution by interleaving the processes of evaluation, scheduling, execution, and monitoring in the life cycle of a plan.

Chapter 6 augments agents with the ability to reason about and manipulate their commitments to maximize the system utility. It gives a novel classification of commitments by analysing the intra-dependency between each commitment’s preconditions and result. For each commitment type, refactoring and reasoning strategies are proposed to maximize overall system robustness and performance.

Chapter 7 presents the applications of our research, resulting in an adaptive business process management system, and trust-based robust scheduling in scientific workflows.

Finally, we make conclusions based on our study and propose future work in Chapter 8.

Chapter 3 to Chapter 6 provides our core design and extended models for building robust multi-agent systems. Each chapter can be mapped into BDI execution stages as illustrated in Figure 1.4.
1.5 Contributions of the Thesis

The numbered circles indicate that the corresponding chapter mainly focuses on the related stage. Together, they systematically improve system robustness and reliability by addressing different aspects of autonomous systems.

Figure 1.4: Overview of Chapter 3 to Chapter 6 in BDI execution process

1. ARTS – plan organization and programming
2. Backtracking recovery based on dynamically generated AND-OR graph
3. Selection, scheduling, and runtime adaptation based on trust
4. Commitment-based agent interaction and scheduling

1.5 Contributions of the Thesis

In this thesis, the following contributions are made.

Chapter 2

• We categorize the existing approaches to building robust multi-agent systems according to their features.
• We highlight the merits and limitations of each type of approach.
Chapter 3

- We abstractly model agents as executors of encapsulated task entities which comply with a set of execution constraints on both normal execution and exception recovery to regulate contract-like interactions between agents.
- We propose a model for specifying hierarchically-organized compensation and exception-handling plans for a given goal, and for systematically selecting and executing these plans.
- We propose a programming model that reduces design complexity while increasing system robustness, by allowing agent developers to compose recursively-defined, atomically-handled tasks.

The preliminary results of this chapter have been previously published in


Chapter 4

- We summarize several principles of agent execution.
- We combine and utilize several beneficial features of the BDI agent model, such as its data structure and deliberation cycle, together with an open nested transaction system, to inherit the benefits of architectural-level concurrency control and distributed management of participating plans.
- We propose a multi-step backtracking recovery mechanism, which utilizes previous planning results in determining response to failure. It allows the substitutable path to start prior to or in parallel with the compensation process, thus making an agent achieve its goals more directly with higher efficiency.

The preliminary results of this chapter have been previously published in


Chapter 5

- We incorporate agent trust to manage the life cycle of an agent plan, thus improving the robustness and predictability of the overall system.
1.5 Contributions of the Thesis

- We devise a heuristic plan selection method, based on the median values of cost, time and trust of each task, to select the most preferable plan, thus reducing the effort needed by the scheduler.
- We design and implement the scheduling process based on genetic scheduling algorithms, with the use of trust to improve the predictability and stability of the schedule.
- We adapt and evolve the schedule at runtime to deal with exceptions during execution by interleaving the processes of evaluation, scheduling, execution, and monitoring in the life cycle of a plan.

The preliminary results of this chapter have been previously published in


Chapter 6

- We propose a novel classification of commitments by investigating the intra-dependency between each commitment’s preconditions and result.
- We present a formalism to express various types of complex commitment based on the classification.
- We provide a set of inference rules to benefit an agent by means of commitment refactoring which enables composition and/or decomposition of its commitments to optimize runtime performance.
- We discuss the pros and cons of an agent scheduling and executing its commitments in parallel. We propose a reasoning strategy and an algorithm to minimize possible loss when the commitment is broken and to maximize the overall system robustness and performance.

The preliminary results of this chapter have been previously published in


Chapter 7

- We apply the concept of the BDI agent model to design and construct business process management systems which inherit the advantages of adaptability and flexibility. We
incorporate the multi-step backtracking mechanism and open nested transaction model to gain system level support of concurrency control and automatic recovery.

- We apply trust-based plan management, including plan evaluation, plan scheduling and runtime plan adaptation, into scientific workflow management systems.

The preliminary results of this chapter have been previously published in


Chapter 2

A Survey of Robust Multi-Agent Systems

Since agents usually work in open, dynamic, and error-prone environments, they are more liable to conflicts and more likely to encounter exceptions, which in turn can cause failures. Therefore, robustness is an intrinsic need for multi-agent systems. Without satisfying this requirement appropriately, an agent system can only remain an experimental toy. Therefore, substantial efforts have been carried out to automate exception recovery and execution resumption, with the goal of making agent programming a serious platform for developing complex applications. This chapter surveys and classifies various proposals and approaches to improving the robustness and reliability of agent-based systems.

2.1 Introduction

Exception handling is a key technique for building a robust system, since unhandled exceptions account for about two thirds of system crashes and fifty percent of system security vulnerabilities [103].

Definition 2.1 (Exception). An exception is any unexpected occurrence that is not accounted for in a system’s normal operation.
Examples of exceptions include incorrect user inputs, memory or data corruptions, environmental anomalies, network disruptions, and software bugs that cause a system to enter an undefined state.

**Definition 2.2 (Failure).** A system failure occurs when the delivered service deviates from the system specification.

Unhandled exceptions are the main cause of errors or failures in the system.

**Definition 2.3 (Exception Handling).** Exception handling (EH) is a method to detect and recover the system from exceptions. It usually provides a control structure to replace the normal program execution with an exception execution when the occurrence of an exception is detected.

Exception handling and the provision of error recovery are difficult to achieve, as it is usually not possible to anticipate all unusual occurrences when the system is designed. They are one of the major challenges in multi-agent systems, in which agents execute in a concurrent and distributed manner. A high-level and systematic mechanism supporting exception handling becomes necessary for multi-agent systems. Otherwise, agent developers need to consider and enumerate many low-level issues, failures, or uncontrolled interactions between the agents in a system, so as to define possible exceptions and their corresponding handlers, which is not only difficult and error-prone, but also sacrifices the abstraction power of the agent model.

Since exception handling has a close relationship with system robustness and reliability, this chapter will focus on various systematic exception handling mechanisms suggested in multi-agent systems. It provides an overview of each approach as well as a taxonomy of them.

### 2.2 Taxonomy of Robust Multi-Agent Systems

Many different approaches and models have been proposed in an attempt to achieve exception handling and robustness in multi-agent systems. Due to the wide variation among these approaches, it is difficult to meaningfully compare and evaluate them. Therefore, we propose to
provide a common terminology and taxonomy of these approaches, necessary for understanding the relationships between them. The details of each approach will be explained in the next section.

The structure of the taxonomy is organized as a hierarchy shown in Figure 2.1. At the leaf node, the content in bracket (“[ ]”) lists the common practice of the corresponding category.

![Hierarchical taxonomy of robust multi-agent systems](image)

Figure 2.1: Hierarchical taxonomy of robust multi-agent systems

At the highest level, a distinction is drawn between exception avoidance (Section 2.3.1) and exception recovery (other parts of Section 2.3). Exception avoidance uses planning and scheduling strategies to reduce the probability of exception occurrences. In contrast, exception recovery focuses on fixing the side-effects after exceptions occur, and then attempts to restore the system to a valid state. By definition, exception handling shares the same meaning as recovery.

The next level in the hierarchy (beneath exception recovery) is a choice between stateful (Section 2.3.3, 2.3.4, and 2.3.5) and stateless (Section 2.3.2) recovery. This choice indicates whether the recovery relies upon the execution history. In the case of stateful recovery, information regarding all the past execution, including the execution structure, is assumed to be available. In contrast, in the case of stateless recovery, the basic idea is to continue the execution by making decisions merely on the current situation.

Stateful recovery can be further classified as transactional recovery (Section 2.3.3) and non-transactional recovery (Section 2.3.4 and 2.3.5). Systems that apply transaction models or concepts, including ACID or advanced transaction models, to organize and monitor system execution, and to automate and accompany the recovery process, are performing transactional exception recovery. Otherwise, they are non-transactional. The simplest form of transactional recovery models usually relies on relational database and ACID transactions to serve the purpose
of exception recovery and concurrent control. By contrast, non-transactional recovery models usually provide developers with a programming framework to simplify the specification of exception handlers.

2.3 Critical Review of Existing Approaches for Robust Multi-Agent Systems

Based on the taxonomy introduced in the previous section, a survey, as well as a critical review, of existing approaches for robust multi-agent systems is presented in this section.

Many existing agent programming languages provide some basic support for the modelling and handling of exceptions. They usually use a specified exceptional event to trigger a new plan which defines handling actions. Such an approach is taken by 3APL [35], dMARS [42] and JACK [49]. However, they lack a systematic way to organize and manage exception handling, requiring developers to code all details. Therefore, they are not included for review and taxonomy here.

2.3.1 Exception Avoidance

Exception avoidance is aimed at reducing the probability of exception occurrences. In multi-agent systems, it usually resides in the planning and scheduling processes, and works by selecting more stable execution paths, or assigning jobs to more reliable execution nodes, to prevent exceptions from happening. For example, consider the scenario of purchasing an iPod from eBay. A seller with high reputation is more likely to deliver the advertised item than others, thus avoiding exceptions during the transaction. However, this model of computation is feasible only in highly constrained systems. Hence, when the assumptions are wrong, these systems can eventually collapse.

Some researchers propose to apply the concept of resource redundancy into the planning and scheduling stage to loosen the constraints of each execution step, thus allowing runtime variations. For example, if the duration of an action is estimated to be one hour, the planner or scheduler can assign one and half hours to it, thus allowing for possible execution delays. [36] provides each task with slack time to execute, thus resulting in more exception-tolerant schedules. [3] uses a more
2.3 Critical Review of Existing Approaches for Robust Multi-Agent Systems

complex redundancy measure for robustness with respect to desired system performance features against multiple perturbations in various system and environmental conditions.

We address this approach in more detail in Chapter 5 and Chapter 6 of the thesis, by introducing two scheduling algorithms, based on the concept of trust and commitment protocols respectively, to achieve a high degree of exception avoidance.

2.3.2 Stateless Exception Recovery

Replanning and plan patching can achieve exception recovery without knowledge of execution history. Replanning discards all previous work when an exception occurs, and starts the planning process again, with the current state of the world, to find a new plan towards the goal. By contrast, plan patching tries to continue the original plan execution, after finding and inserting an additional plan to recover from the exceptional states. By definition, both replanning and plan patching follow the same process as planning, but they occur when an exception takes place.

The feature of statelessness simplifies the design of exception handling, because it no longer needs to take the execution history into account. In other words, it does not differentiate the exceptional state from the normal one, and leaves all work to the planner.

However, replanning and plan patching can be computationally expensive and in many cases not practical. In general, the execution context provides precious clues about the exception, as well as about the recovery methods. Without making use of this information, the planner has to consider the whole problem domain to obtain a solution, which is computationally expensive. What is more, the design of the planner would require complete knowledge about the environment and business domain, thus making the application complex and expensive to implement.

2.3.3 Transactional Exception Recovery

Research on transactions is fruitful and the transaction mechanism is very mature. The concept of transactions has also been widely applied to the workflow [59] and agent domains. In this section, we provide a survey of various approaches to integrating transactions into multi-agent and mobile agent systems. Chapter 3 and Chapter 4 of the thesis present our extension and contribution to the seamless integration of the concepts of transactions and agents.
Transaction Oriented Multi-Agent System (TOMAS)

TOMAS [18, 19, 127] combines a nested transaction model with the BDI agent architecture to manage teams of cooperative agents. It was designed to support the concurrent execution of agent plans, and to guarantee the consistency of the agent belief set in case of failure/update.

In TOMAS, every update to beliefs is treated as a flat transaction and placed into the nested transaction tree of a plan execution, and every agent acts as the monitor of its own transactions. A locking mechanism is applied to guarantee the isolation of operations executed on beliefs, to avoid conflict updates. The communication between agents is abstracted as two actions: send and post. The Two-Phase Commit protocol [66] is used to enforce the synchronization of agent coordination.

However, a nested transaction model requires full control of resources and does not expose partial results. Due to these restrictions, its applicability is restricted. Even if all agents perform database manipulation only, the model is still too rigid to be practical in a multi-agent system context, because long-running activities can lead to the locking of resources for very long periods, leading to deadlocks. One way to overcome this is to let the developers make sure such long transaction executions are not allowed. Once again such responsibility will burden the programmers.

Transactional Agents

[110] takes a more workflow-like approach by treating every action as an ACID entity and putting them into an open nested transaction model to create transaction trees. In this model, each agent is divided into two parts: a planning agent and an execution agent. The planning agent is in charge of perceiving the environment and cooperating with other planning agents to develop a logical plan, which is represented as a transaction tree, for their joint goal. The planning agent then delegates its local plan to a peer execution agent for completion.

The capabilities, which are building blocks of plans, of each agent are defined in the action repository. When an action is added into the transaction tree, it is encapsulated within an ACID transaction. Event-Condition-Action (ECA) rules are used to coordinate two agents if one uses the partial results of another. An event at the sender’s side triggers a message, containing conditions
and actions, to the receiver's side.

Leaf nodes of the transaction tree have all the ACID properties, since they are flat transactions. By contrast, the atomicity of control nodes is achieved by forcing the execution agent either to execute the transaction to its completion, or to appear as if nothing has been done by means of compensation in case of exceptions. This method also shares many of its features with TOMAS.

However, this model does not allow changes to execution plans, which are quite common in agent applications due to the dynamic nature of the environment. What is more, arbitrary and unstructured linkages among different agents breach the design principles of modularization and loose coupling.

**Goal-Based Semantic Compensation**

As an extension to the compensation concept in advanced transaction models, [153, 154] discuss the use of goal-based semantic compensation in the context of agents. Their subsequent paper [155] describes a logging-based approach to support the compensation methodology.

Semantic compensation is introduced to address the problem that many actions in multi-agent applications cannot be delayed and always commit. Therefore, to deal with exceptions, the corrective stages must be implemented by forward recovery or failure compensation, that is, by performing additional actions to correct the problem, instead of a transaction rollback.

The model employs a goal-based specification of failure-handling knowledge. Two types of failure-handling knowledge are required to be provided for each agent in a system: information about what needs to be achieved to clean up (e.g., compensate) the failure of a task, and a set of goal-level failure-handling strategy rules. The information is specified in terms of the goals that need to be achieved in order for the compensation to be successful, not in terms of action or plan steps that might implement these goals. In addition, the rules specify how to employ failure-handling. The rule matches on goal-level information about a task, not execution details.

However, the model focuses on specifying, at an abstract level, what to do on a failure, not how to achieve the implementation. To specify “what to do” is not an easy task since it involves substantial analysis. Furthermore, to find “how to do it” is difficult for agents to achieve.
■ Transactional Mobile Agent

Research on mobile agents [92,172] focuses on the mobility of agents. A mobile agent is allowed to move through a network of heterogeneous machines, and continue its execution on the destination computer. Because of the distribution of the execution, robustness and reliability are even harder to achieve.

Most proposals for building mobile agents apply transaction models to guarantee the robustness and reliability of agent execution [124]. In general, the complete execution of an agent can be partitioned into a sequence of stages according to the execution location. In each execution stage, the agent may visit and modify some local resources. The challenge is to ensure the consistency and atomicity of the overall execution with regard to the distributed stages.

The common approach is to encapsulate the execution of each stage as an ACID transaction, and then compose these transactions as a transaction tree to model the overall execution. However, each approach has its own focus. [50, 122, 123] use open nested transactions to organize and monitor the executions. [148] addresses the issue of partial rollbacks to a savepoint in a mobile agent execution through the use of physical logging and compensation operations. [139] applies a two-phase commitment protocol to guarantee the completion of all execution stages. [13] concentrates on the isolation property, which prevents undesirable interference between different transactional agents. [81] discusses different types of commitment constraints in the presence of computer faults.

■ Phoenix Application Recovery Framework

The Phoenix project [9, 10, 96] exploits database recovery techniques to improve application availability and error handling robustness. It proposes the transparently persistent stateful programming model which delegates the tasks of state management and state persistence to the Phoenix infrastructure, enabling the programmer to focus almost exclusively on the business logic of the application, and thus making applications simpler, more maintainable, and more likely to be correct.

Phoenix uses automatic redo logging to capture the non-deterministic events which permit applications to be replayed to restore their state in case of failure. The platform ensures exactly-
once execution semantics of applications; that is, despite a wide range of possible failures, the program completes as if it has executed without failures and exactly once.

Phoenix introduces interaction contracts between communicating pairs of application components regarding message and state preservation; only events which may lead to non-deterministic execution are logged. Therefore, the cost of logging is significantly reduced and system performance is improved.

Though the Phoenix project has not been implemented for multi-agent systems, its approaches and techniques can be applied to enhance the recoverability and reliability of the interaction protocols among agents.

### 2.3.4 Non-Transactional Exception Recovery

This section introduces two representative approaches in the category of non-transactional exception recovery.

- **SaGE Framework**

The SaGE framework [144] is built on the service concept. Every agent is abstracted as both a service provider and a consumer. Cascading requests among agents result in cascading service executions that form a logical structure (tree) of execution contexts. Coordination relationships among agents, as well as the exception handling process, are represented by the dependencies between services.

- **System Organization.** The system has three vertical management levels: service, agent, and role. Services represent the actual functionalities and are provided by agents, while roles are used to represent and manage a set of agents that share a common ability. Correspondingly, exception handlers are defined and organized in the same hierarchy. Each service of an agent is associated with a service handler to catch and deal with exceptions that are raised, either directly or indirectly, during the execution. Thereafter, an agent should be associated with an agent handler for all the services it provides, because it can give a common solution to similar exceptions. For the same reason, a role should be associated with a role handler, to deal with exceptions that concern all agents which play a given role.
- **Exception Handling.** When an exception is signalled, the execution of the defective service is suspended. The search starts there, first in the service itself, and then in the agent for which it executes. If no handler is found, the service caller is considered to be in an exceptional state and the search process repeats until an adequate handler is found or the top-level is reached.

SaGE integrates concerted exception support in its exception propagation mechanism, to avoid reacting to sub-critical situations, and to collect exceptions so as to reflect a collective or a global defect. Services or roles store exceptions in a log to maintain the history of the exception as so far propagated. Whenever a newly propagated exception is logged, the concerted exception function associated with the recipient service/role is invoked to evaluate the situation. Depending on the nature or the number of logged exceptions, it may determine the exception is critical enough to be handled; or else it may cause a new exception to be raised from a set of logged exceptions, the conjunction of which creates a critical situation.

- **Conclusions and Limitations.** SaGE is a totally distributed proposal, with exceptions solved at the agents in which they are necessary and able to be solved. The only requirement is that a service execution starts before any of its sub-services, and terminates only after all of them have terminated. But this model leaves a heavy burden on programmers, who need to explicitly organize all possible exceptions. Programmers also need to consider agents’ competitive accesses to shared resources, and to avoid deadlock among agents. For this reason the SaGE model cannot be used to build complex and large-scale applications unless transaction concepts are introduced into their services.

**Exception Model on Commitment Protocols**

[100] discusses design-time modeling and runtime handling abstractions for exceptions in the framework of commitment protocols [101, 177, 181].

They argue that representing processes in terms of protocols, which are specifications about the interactions between autonomous agents, simplifies exception handling because protocols provide the bounds for the scope of an exception. The work is then centered around the concept of commitment protocols, each of which is modeled as a set of computations; each protocol
represents a set of allowed interaction sequences and shows the evolving commitments of the participants. Exceptions are then modeled via preference structures (some sequences of execution are more desirable than others) induced on these sets of computations.

The model claims to deal with both expected and unexpected exceptions. For expected exceptions, the preference structures statically show how they can be handled. For unexpected exceptions, protocol splicing with the merge operator is defined to compose protocols and exception handlers, whereby appropriate exception handlers can be dynamically introduced into a protocol as needed. Splicing exception handlers at runtime requires a library of handlers and a search through this library to find the appropriate recovery process.

The use of preference specification can speed up the execution of an interaction protocol. However, it involves considerable design-time effort and extensive domain-specific knowledge, because it requires protocol designers to have a good understanding of the protocol and the possible exception conditions that may arise. In contrast, splicing exception handlers at runtime no longer requires the effort of protocol designers, because of the assumption that the exceptions are not anticipated. However, it requires a complete library of handlers addressing all possible events, and a search through this library, which could be computationally demanding.

The work did not take the issue of concurrent execution into consideration, and planned to incorporate the concept of transactions in future work.

### 2.3.5 Separation of Exception Recovery Knowledge

Orthogonally from the taxonomy of dynamic exception recovery, Guardian [107, 151] and Citizens [87, 88] propose to separate the mechanisms and knowledge about exception handling from the agent system to a centralized exception manager. Both models apply a supervisory agents (called a guardian in Guardian and a sentinel in Citizens) to monitor the status of each participant. However, a guardian is a centralized control agent, while a sentinel is distributed to be part of the execution agent itself.
Guardian

In Guardian [107, 151], the system is composed of two kinds of agents: the guardian, and the application agents. The guardian is a special agent whose primary purpose is to monitor and control application agents with respect to exception conditions. Thus, the global level exception-handling concerns are separated from the application agents, and are encapsulated into the guardian.

- **Agent Management.** The structure of the overall system includes several two-level trees. The root of each tree is the guardian, and the leaf nodes of the root are application agents. It is required that no application agent exists if there is no guardian controlling it. If an agent loses its guardian, for example due to a host crash or communication link loss, then it should either bind to a new guardian, or terminate itself. On the other hand, if the guardian loses an agent, then it may create a new agent to replace the lost one, or do nothing.

- **Exception Handling.** If an exception signalled in an agent cannot be fully handled inside the agent itself, it will be directly sent to the agent’s guardian for further instructions. Then to recover, the agent just needs to follow the instructions from the guardian.

  The guardian performs the exception analysis to determine the best solution. For this purpose, the guardian has the power to examine the state of any related agents by communicating with them or their environments. It may also control the behaviour of its monitored agents by terminating them or changing their execution plans if necessary.

  When receiving exceptions from multiple agents simultaneously, the guardian will serve as a central coordinator to resolve them, possibly by combining some exceptions, and to find a suitable handler for the resolved exceptions.

- **Conclusions and Limitations.** The relationship among different guardians is not considered. Although there can be more than one guardian, the choice of the most appropriate one for the application agent is not discussed.

  The guardian does not record the dependencies among agents, and it relies on exchanging information with participants to infer their relationships during exceptions. In this situation, all participants’ activities need to be frozen to avoid changes to execution context. This requirement greatly limits the practicability of Guardian model.

  Moreover, the system is vulnerable to the crash or unavailability of the central leader.
Building an appropriate guardian, dealing with real agents, is itself a bottleneck and can cause many faults in the system.

**Citizens**

Haegg [67] introduces the concept of the sentinel agent, which controls an agent’s communication for error detection and recovery, to guard certain functionality and to protect from undesired states. Klein *et. al.* [87, 88] extend the concept to the Citizen model.

Citizens are application agents under the supervision of sentinels. When exceptions occur, the sentinels consult a set of domain-independent services with exception handling expertise, called EH services, which maintain knowledge bases of all possible exceptions and their handlers. This approach is built on the assumption that the interactions among agents can be treated as dependent on the coordination protocol used, but as independent of the domain the agents work in. Therefore, the Citizen Model is only concerned with exceptions that result from violations of coordination protocol assumptions.

- **Agent Management.** Every agent is supervised by a sentinel that will observe and influence agent behaviour as necessary to ensure the robust functioning of the system as a whole. Sentinels monitor message traffic to develop a model of the commitments their agents are involved in, use the appropriate anticipation and/or detection handlers to uncover when these commitments are violated, diagnose the underlying causes to identify the appropriate avoidance and/or resolution handlers, and enact these handlers to help re-establish the violated commitments, or at least to minimize the impact of them having been violated.

- **Exception handling.** When an agent enters a multi-agent system supported by the EH (exception handling) service, it needs to register the kinds of coordination protocols it will use and the types of exception handling behaviour it can support. Based on this information, the specified sentinel is assigned to the agent to check for the violation of related commitments.

  If a violation is detected, sentinels consult EH services to identify the characteristic exceptions, as well as the appropriate handlers, for the coordination protocols being used in a particular multi-agent system. Then the sentinels will execute the chosen handler cooperatively to solve the problem.
• **Organization of EH Services.** The key component of the Citizen system is the construction of well-organized exception handling knowledge bases. One scheme is based on three interlinked taxonomies. The first taxonomy captures MAS coordination protocols, arranged in an abstraction hierarchy. Each protocol has pointers to the exceptions that characterize it, and all exceptions are stored in an exception taxonomy. Exceptions are themselves linked to the potentially applicable handlers in an exception handler taxonomy.

• **Conclusions and Limitations.** The architecture of the Citizen model is fundamentally hybrid: it is based on a distributed sentinel population plus EH services, which are essentially database applications. But this design results in complicated and inefficient sentinel behaviour. To begin with, the sentinels’ monitoring of every message in and out of the agents will greatly reduce system performance. Moreover, the sentinels rely heavily on answers from the EH services, which are designed to handle only protocol-related exceptions, without awareness of the context and dependencies among agents. This lack of awareness can result in the selection of inefficient/bad exception handlers.

### 2.4 Related Work

[120] proposes a different taxonomy of exception handling in multi-agent systems along the dimensions of distribution and autonomy, as shown in Figure 2.2:

![Figure 2.2: Related work on Taxonomy of Exception Handling (Adapted from [120])](image)

The vertical axis represents the degree of distribution. Although multi-agent systems are typically distributed, their exception handling frameworks can follow either a centralized or a
distributed design. The horizontal axis is the degree of autonomy. The participants of the system can be objects, reactive agents, or proactive agents. Objects are passive entities that rely on method invocation. Reactive agents are autonomous entities which are capable of perceiving their environment and adjusting their behaviours reactively. Proactive agents are autonomous entities which can reason about the environment and make plans to manipulate the environment actively.

Compared with our taxonomy, which is a process-centered view of the organization and mechanisms of exception handlers, [120] focuses on the features of the participating agents. Moreover, they do not present a thorough coverage of the research on building robust systems. In fact, Figure 2.2 can be treated as part of the stateful recovery branch, excluding the group of transactional approaches.

[124] provides a survey and classification of mobile agent platforms, with the criteria of being transactional or non-transactional. Therefore, their work can be directly merged into our taxonomy framework.

2.5 Summary

In this chapter, we have provided a study of existing work on building robust and reliable multi-agent systems. The survey is centered around the topic of exception handling; it reviews various representative state-of-art models and approaches. The merits and limitations of each approach are analysed to show its applicable conditions. A hierarchical taxonomy of these approaches is introduced to help the better understanding of the relationships between them.

All of these approaches share one common feature: they do not consider the low level details of within-plan or within-agent execution for handling exceptions. Instead, they focus on detecting and handling the exceptions related to inter-plan and inter-agent interaction and communication. In fact, without high-level support for exception handling, the multi-agent model will lose its benefit of being a high level abstraction and design tool, since developers have to consider every detail in all layers of system construction, making application development difficult, and resulting in poor reliability.
Chapter 3

Specification and Organization of ARTS Model

The ARTS (Agent-oriented Robust Transactional System) model applies transaction concepts to provide agent developers with high-level support for agent system robustness and reliability. This chapter discusses the specification and organization of the model.

In ARTS, agents are abstractly represented as executors of encapsulated task entities which comply with a set of execution constraints on both normal and compensation (repair) executions. ARTS then defines the task interface in terms of predictable terminating states, to support a contract-like interaction among agents. In conjunction with this encapsulation of task semantics, ARTS defines a model for specifying scoped compensation and exception-handling plans for a given task, and for systematically selecting and executing these plans — triggered by subtask events — so that the enclosing plan semantics are enforced. These capabilities together define a model that reduces design complexity while increasing system robustness, by allowing an agent developer to compose recursively-defined, atomically-handled tasks.

3.1 Introduction

Agents are a powerful abstraction concept for analyzing, designing, and implementing complex software systems [77]. However, something that still remains out of touch is an easy-to-use mechanism, at the corresponding level of agent abstraction, for maintaining system robustness
and reliability, with respect to the correct execution of agent actions, even in the presence of abnormalities. As a result, while some aspects of an agent language may be high-level, developers must still consider low-level details of exceptions, failure, or uncontrolled interactions between the agents in a system. Because agents are often used in an open environment where they do not have full control of resources, and usually run autonomously in distributed and parallel fashion, it is very difficult for developers to correctly address the handling of these complexities. Therefore, high-level support for robustness and reliability is a key requirement for the success of applying the agent model to practical applications.

The approach described in this chapter provides such support by defining a model for the encapsulation of task result, repair, and exception-handling semantics. This allows agent system design to be viewed as the composition of atomically-handled task entities, thus reducing system complexity and unpredictability, and supporting robust system development.

3.1.1 A Motivating Example

We now describe a scenario which will be used as a running example throughout the chapter. The scenario, in a “hospital” domain, is that of a Surgery Manager, whose responsibility is to manage and coordinate all activities related to a patient’s surgery. In practice, there may be many subsystems in a hospital to manage different affairs, such as Scheduling and Ward Inspection. The Surgery Manager will assign tasks to some of them and then coordinate them to achieve its goal of running the surgery smoothly.

Example 3.1. One common responsibility of the Surgery Manager is to prepare the surgery on the appointment date. The detailed process is illustrated in Figure 3.1, showing normal operation only.

Before entering this stage of the Surgery Manager’s activities, some Preconditions must already be satisfied (marked Pre in the figure), such as appointed surgery time and assigned doctors. After the completion of surgery preparation, Post-conditions (Post) must hold: everything has been arranged for the surgery. On the scheduled day, the Surgery Manager will ask participating agents to check the patient in (t2) and confirm the attendance of the assigned staff (t3) before the specified time for surgery preparation. It will also ask for one available operating
3.1 Introduction

Figure 3.1: A Surgery preparation task in a hospital environment

theatre to be prepared before the surgery time ($t_4$, $t_7$ and $t_8$). There are also some other tasks required to confirm the permissibility of the operation, such as checking tolerance for anaesthetic drugs ($t_5$) and other preoperative tests ($t_6$). Finally, it will debit the cost of operation from the patient ($t_{11}$) after the completion of the operation ($t_{10}$).

The process above works well if the surgery manager can fully control each participant, but contingencies may occur at any point due to unpredictable sub-system behaviour. For example, the patient may be absent on that day, the assigned doctor may be ill, or the preoperative test may indicate that surgery is not suitable. There also exist some Maintenance conditions ($M$) which, if they do not hold, render the whole scenario moot. For example, the patient may decide not to have surgery at all, or the doctor’s diagnosis may be proved incorrect. Additionally, it is possible that exogenous events may trigger the need to cancel the surgery, such as a problem with the hospital electrical system.

With existing agent programming languages (APL), developers have to struggle with the complexity of predicting all possible interactions among participating agents, of hard-coding agent constraints, and of managing many related or unrelated exceptions. It is typically even harder to model the cancellation of completed tasks. A further challenge is integrating these different types of knowledge in a consistent manner. We introduce the ARTS (Agent-oriented Robust Transactional System) model to address these issues.
3.1.2 Overview of the ARTS(Agent-oriented Robust Transactional System) Model

Given the success of transaction processing [66] in guaranteeing correctness and consistency in databases and distributed systems, we propose the application of transaction concepts to multi-agent architectures. Traditional transactions have ACID (Atomicity, Consistency, Isolation and Durability) properties, which prevent inconsistency and integrity problems. However, ACID transactions require the activity in question to obtain full control of, and exclusive access to, its resources. In contrast to database applications where transactions can lock records to get exclusive access, and can restore any data from history, agents usually work in an open environment, and operate on physical objects where actions “always commit”, and it is impractical or even impossible to satisfy either the locking or the restoral requirement. For example, a flight reservation agent cannot lock the schedule to avoid flight changes, or restore a bank account to the original amount by itself when cancelling a booking. Thus, traditional transaction mechanisms need to be extended for an open and shared environment before they can be applied to agent systems.

To address the demands of agent developers and the differences between ACID transactions and agent applications, in this chapter we propose the ARTS model, which defines a concise hierarchical model for the composition of distributed agents. The model applies transaction concepts to agent systems in the following ways:

- It defines encapsulated task entities that comply with a set of execution constraints on both normal and compensation semantics for the task.
- It defines a set of predictable terminating states as the interface for each task entity, and then regulates a contract-like interaction among agents based on these states.
- It defines a model for specifying scoped and hierarchically-organized compensation and exception-handling plans for a given task, and for systematically selecting and executing these plans — triggered by subtask events — so that the enclosing task semantics are enforced.

These capabilities together define a model that reduces design complexity while increasing system robustness. ARTS allows an agent developer to compose recursively-defined, atomically-treated tasks, and to consider exception-handling at a higher-level of abstraction. It ensures modular agent system design and reduces the coupling among agents. Thus, the complexity
of dealing with combining the possible termination states of all participating agents is greatly reduced.

In this chapter, we primarily address aspects of task organization and failure atomicity. Our goal is not to create yet another new agent programming language; rather, our model is a programming paradigm which can be integrated with and embedded into existing agent programming platforms as a transactional support layer, to reduce the complexity of maintaining system robustness. This paradigm can also be applied to other methods of constructing complex systems, such as workflow and web services.

The remainder of this chapter is organized as follows. Section 3.2 illustrates the abstract model for both single and composite tasks, the building blocks for agent plans in ARTS. Section 3.3 defines the specification of the plan information for ARTS, and then describes the use of this specification to support the robust execution of agents. Section 3.4 discusses related work and presents our conclusions.

### 3.2 The Task Model in ARTS

The ARTS framework focuses on the beliefs and plan composition parts of the BDI agent model. In general, a plan specifies a task network, to compose and organize a group of tasks so as to achieve a well-defined goal or react to an event in the environment, such as trip planning or order handling in an enterprise. The agent may break tasks into subtasks, performing some subtasks itself and delegating some to other agents. The decomposition process usually produces a hierarchical tree structure.

**Definition 3.1** (Task). A task $t$ is an identifiable and measurable amount of work performed to convert inputs into outputs according to specified specifications.

A task can either be viewed externally as an atomic task, or be studied internally as a complex task composition. In this section, we will introduce formal representations on both views, and organize the agents’ beliefs into several categories with respect to knowledge about the execution and composition of a task: preconditions, maintenance conditions, post-conditions, invariants, and execution and termination states.
3.2 Specification and Organization of ARTS Model

Figure 3.2: Interface of a single task

Table 3.1: Constraints of a successful task execution

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre</td>
<td>$s_0(Pre)$</td>
</tr>
<tr>
<td>$M$</td>
<td>$s_i(M)$ for $\forall i (0 &lt; i &lt; n)$</td>
</tr>
<tr>
<td>Inv</td>
<td>$s_0(Inv) \land s_n(Inv)$</td>
</tr>
<tr>
<td>Post</td>
<td>$s_n(Post)$</td>
</tr>
</tbody>
</table>

As agents take over tasks in the execution tree, they build up the dependency relationship between them accordingly. In this chapter, we will not differentiate agent dependency from task dependency.

3.2.1 Modelling a Single Task

A single task is treated as an atomic process whose implementation details are not visible. Therefore, only the interface it exposes to the other agents in the system is the concerns of users. Figure 3.2 shows the interface of a task. To achieve a certain goal, task $t$ has a set of specifications in the form of constraints labelled Precondition ($Pre$), Maintenance condition ($M$), invariants ($Inv$) and Postcondition ($Post$).

During the execution of a task, $t$, its state at a certain point of time is denoted as $s_i$ ($0 \leq i \leq n$); this state consists of a set of predicates about the environment as well as about the agent itself. The state information is stored in the belief base $B$ of the agent which hosts the execution. For a successful execution, $t$ must start in an initial state $s_0$ satisfying $Pre$ and terminate in a final state $s_n$ satisfying $Post$. During the execution, $t$ must guarantee $M$ to ensure its validity. Additionally, $Inv$ is required to be true to guarantee the consistency of the transformation from $s_0$ to $s_n$. We use $s_i(x)$ to denote that $x$, a set of predicates, is satisfied by $s_i$. That is, $s_i(x) \equiv s_i \models x$. Thus, the constraints of a task can be formally represented as shown in Table 3.1.
Definition 3.2 (State Consistency). $s_i (0 \leq i \leq n)$, the state of a task, is consistent if $s_i (\text{Inv})$.

Definition 3.3 (Successful Task). The execution of a task is successful if $s_n (\text{Post}) \land s_n (\text{Inv})$.

Definition 3.4 (Compensated Task). The execution of a task is compensated if $\neg s_n (\text{Post}) \land s_n (\text{Inv})$.

Even though a compensated task fails to achieve its goal, it has recovered from the side-effects of the execution and terminated in a consistent and acceptable state.

Definition 3.5 (Aborted Task). The execution of a task is aborted if $\neg s_n (\text{Inv})$.

An aborted task fails both to achieve its goal, and to restore the result from its inconsistency. User interactions are usually required in this case.

Example 3.2. In the hospital as in Example 3.1, $t_{11}$ is in charge of debiting the cost of the operation from the patient’s account. To demonstrate the feature of different constraints, the internal execution is represented as two separate steps with the assumption that the cost is 200.

<table>
<thead>
<tr>
<th>Table 3.2: Constraints of the payment task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inv:</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Pre:</strong></td>
</tr>
<tr>
<td><strong>M:</strong></td>
</tr>
<tr>
<td><strong>Post:</strong></td>
</tr>
</tbody>
</table>

patient.balance = patient.balance - 200; \hspace{1cm} (1)

hospital.balance = hospital.balance + 200; \hspace{1cm} (2)

Pre is required to make sure that the patient has sufficient funds before the payment can proceed, while Post will ensure the receipt of the payment after task termination. During the task execution, if $M$ becomes false at any time, the whole task is not valid any longer. By contrast,
it is only necessary for Inv to hold before and after task execution. That is, Inv can be violated during the execution. In the example, the second requirement of the invariants becomes false right after step 1 and before step 2.

However, the constraints of a task might be violated during its life cycle, especially in a dynamic and non-deterministic environment, because of exogenous events, unanticipated interactions between agents, or non-deterministic action results. It is generally desirable that the failed task can “undo” the previous execution and terminate in a well-defined state.

**Definition 3.6** (Task Failure Atomicity). In case that \( \neg s_i(M) \exists i(0 \leq i \leq n) \), or \( \neg s_n(Inv) \lor \neg s_n(Post) \), if task \( t \) can automatically compensate the side effect of the previous execution and restore the consistency of the system, \( t \) has the feature of failure atomicity.

With the feature of failure atomicity, a task is encapsulated as an abstract entity with a neat interface exposed. Therefore, developers can focus on the interaction of task composition instead of low-level details of task internal instructions. However, the key to failure atomicity is the definition of the compensation process of a task. The common methods of defining the compensation processes of a task are as follows:

- The simplest method is to design a task as being read-only. Read-only tasks are always in a consistent state since they cannot make any modification to the system.
- If the task is composed solely of recoverable actions, a log can be maintained to record the execution history. Thereafter, the compensation process can reverse the execution step by step until it reaches the initial state. This technique is commonly used in database applications.
- If there are some non-recoverable actions, such as sending an email notification, they should be carried out as late as possible in the task. Therefore, if the task fails before these actions take place, the task is still recoverable. Alternatively, one can develop protocols that indicate that some actions are deemed nullified after receiving a compensating request.
- Finally, task developers can explicitly define semantic compensation processes [153, 154] for non-recoverable actions inside the task implementation. Note that **semantic**
compensation does not necessarily imply an exact “undo” of the task, and it is usually application- and situation-dependent.

**Example 3.3.** Since the operations of both credit and debit are recoverable, at the end of the payment task $t_{11}$ in Example 3.1, if the amount of debit and credit are not equal at the end, $t_{11}$ should restore the system to the state that it was in prior to the task execution.

Finally, we can give a formal definition of atomic tasks.

**Definition 3.7** (Atomic Task). Formally, an atomic task $t$ can be denoted as a quintuple

$t = \langle S, C, s_0, \delta, F \rangle$.

- $S$ stands for the set of possible states during the execution of task $t$. $S = \{\text{Inactive, Active, Finishing, Successful, Compensating, Compensated, Aborted}\}$. A state $s \in S$ reflects the perception of the executing agent about its status and evolves with the execution of task $t$. The details of the state information are stored in the belief base of the host agent ($s \subseteq B$).
  - **Successful, Compensated and Aborted** are possible terminating states of $t$ as described previously.
  - **Inactive**: $t$ has not been required to execute.
  - **Active**: $t$ is executing its “forward” or normal work.
  - **Finishing**: $t$ has done all its work, but has not yet verified that it has achieved the conditions Post and Inv.
  - **Compensating**: $t$ is performing a semantic compensation of its work.
- $s_0$ is the initial state of $t$ and $s_0 = \text{Inactive}$.
- $C$ stands for the constraints on $t$, including Pre, M, Post and Inv. Each kind of constraint needs to be satisfied by beliefs during the corresponding execution state.
- $\delta : S \times C \rightarrow S$ is the state transition function of $t$.
- $F \subset S$ is the set of final states and $F = \{\text{Successful, Compensated, Aborted}\}$.

Figure 3.3 shows the execution state transition graph which enforces the task’s interface in Figure 3.2. The details of internal task implementation and execution, such as variable valuation,
are not taken into consideration, in order to focus on task functionality. With the transition predicates, we make the closed-world assumption, with negation as failure.

After task $t$ has terminated successfully, users may find that the result of $t$ is no longer valid and the execution of $t$ should be rolled back. However, since $t$ may contain non-recoverable actions, or its externalized results may have already affected the execution of others, actual roll-back in a situated agent system is typically not possible. Instead, a compensation task is introduced to semantically address the changes that have occurred.

**Definition 3.8** (Compensation Task). *The compensation task of* $t$, denoted as $t^{-}$ or undo($t$), *is a task whose purpose is counteracting the side-effects of* $t$. $t^{-}$ can only start after $t$ is in the Successful state.

**Example 3.4.** *In Example 3.1, if the patient must cancel his surgery after the completion of check-in ($t_2$), it is not acceptable to erase the check-in record. Rather, a compensation task of $t_2$ should be carried out to append a record of the cancellation into his case history.*
3.2 The Task Model in ARTS

3.2.2 Modelling Task Composition

In general, atomic tasks only have limited functionality for a specific purpose, and no single one of them can fulfill user requirements completely. Therefore, in practice, they are composed and organized to achieve complex goals, exceeding the capacity of individuals. Meanwhile, although users of a task, who only care about its functionality and interface, can view it as a single and atomic entity, the developers of the task have to consider the internal implementation details of how the functionality is achieved. Thus, this section describes the task model from its composite perspective, with a focus on the relationship between the parent and its children in the hierarchical decomposition structure.

Task decomposition is the process of breaking down a complex task into parts that are easier to understand, implement, and maintain. It is a general approach, applying the divide and conquer strategy [33] to deal with complexity. It recursively decomposes each task into subtasks, which construct a hierarchical structure, until all leaf tasks can be fulfilled atomically.

At each internal level of the hierarchy, each task can heuristically be seen as creating a new functionality exceeding the boundaries of each individual subtask. The procedural knowledge of how the subtasks are composed is encapsulated inside the parent task, and can be hidden from higher level users.

**Definition 3.9** (Task Inheritance). *If the functionality of a task \( t \) is fulfilled by composing a group of subtasks \( \{t_1, t_2, \ldots, t_n\} \), \( t \) is the parent of the subtasks, denoted as \( t \triangleright t_i \) (\( 1 \leq i \leq n \)). If \( t_a \triangleright t_m \) and \( t_m \triangleright t_d \), then \( t_a \) is the ancestor of \( t_d \), denoted as \( t_a \triangleright t_d \). ■*

The execution of each task, no matter whether it is an internal or a leaf node in the task decomposition tree, will be carried out by an agent. Correspondingly, the organizational relationship among tasks is passed down to the participating agents. Generally, the agent in charge of a parent task usually takes the managerial role of organizing and coordinating the agents responsible for children tasks. Since the dependency relationship between agents is directly mapped from that of tasks, we will not differentiate them in the rest of thesis.

**Example 3.5.** *In Example 3.1, the task of surgery preparation on the operation day (\( t \)) in hospital is achieved by composing 11 subtasks (from \( t_1 \) to \( t_{11} \)). Hierarchically, \( t \) is the parent of the subtasks*
which are treated as atomic entities. The task $t$ also has a set of constraints, including Pre, M, Post, and Inv, to judge the validity of its execution. In addition, as a composite task, $t$ also contains procedural knowledge about subtask dependencies, which are specified as an AND-OR graph in Figure 3.1. After all tasks are assigned to agents for execution, the main role of the agent for $t$ is to coordinate and manage the execution of agents for the subtasks of $t$.

From the point of view of the parent task, each of its child tasks is atomic, with a simple interface: once activated, it only has one of three possible results of being Successful, Compensated, or Aborted. Each subtask also holds a contract to its parent. That is, it attempts to achieve the state of Successful; if it cannot, it will compensate the work done thus far to reach an acceptable state of Compensated. But if acceptable repair cannot be achieved, it will terminate in the Aborted state and leave cleanup work to the caller, in the context in which the task was requested. However, the details of how it achieves and compensates the goal are hidden inside the black box of the task.

This abstraction, or encapsulation, is appropriate for three reasons. First, agents, as the task executors, are autonomous entities which operate independently and retain control over their execution. So even though they may be able to view the processing details of other agents, they cannot affect them. Second, a complex system is more robust if one component can ask others to fulfill certain functionalities and then wait for results without requiring details about how those results are achieved. Otherwise, they are closely coupled, which is likely to make system management complex. Third, different agents may use different representations of the world internally. As with the hospital example in Figure 3.1, the surgery manager wants to know if the operating theatre is ready or the test for anaesthetic tolerance is passed, but does not care about the details of preparing theatres or performing anaesthesia tests.

In viewing a task as a building block, we can construct hierarchical structures to model complex tasks.

**Definition 3.10 (Composite Task).** A composite task achieving a given goal can be denoted as $t = < T, S, C, s_0, \delta, D, F >$.

- $T$ is the set of composing subtasks. $T = \{ t_1, t_2, \ldots, t_n \}$. 
3.2 The Task Model in ARTS

- $\overrightarrow{S}$ is the set of vectors representing the execution state of $t$ itself and its subtask. Let $\tau$ be either the parent of $t$ or a child $t_i (1 \leq i \leq n)$. The execution state of $\tau$ is denoted as $s^\tau \in \mathbb{S}$ (refer to Definition 3.7). $\overrightarrow{s} \in \overrightarrow{S}$ is a vector containing the information of the execution states for all participating tasks at a certain time, defined as $\overrightarrow{s} = \langle s^t, s^{t_1}, s^{t_2}, \ldots, s^{t_n} \rangle$. $\overrightarrow{s}[\tau] = s^\tau$ represents the projection of $\overrightarrow{s}$ on $\tau$.

- $C$ is the constraint set of $t$.

- $\overrightarrow{s}_0 \in \overrightarrow{S}$ is the initial state of $t$, and $\overrightarrow{s}_0[t] = \text{Inactive}$.

- $\delta: \overrightarrow{S} \times C \times D \rightarrow \overrightarrow{S}$ is the state transition function of $t$.

- $D$ is the dependency set of subtasks.

- $F \subset \overrightarrow{S}$ is the final state of $t$, and $\forall \overrightarrow{s}_f \in F$ ($\overrightarrow{s}_f[t] \in \{\text{Successful, Compensated, Aborted}\}$)

The definition of a composite task extends that of an atomic task in several aspects. First, $T$ is added to provide reference to each of the subtasks. In addition, $D$ is appended to represent the dependency relationship among them. $D$ is a set of dependencies specifying connections between each participating task, as will be described further in Section 3.3.

The dependencies effectively encode the task execution paths, defined by the domain’s business logic and exception handling knowledge, in conjunction with ARTS’ methodology for organizing and using this knowledge. There has been much previous research towards task dependency management, e.g. [5,25]. However, our proposal merges dependencies into a concise programming setting, and provides them with an operational semantics to guarantee the features of robustness and reliability. The process of generating $D$ according to the procedures specified by developers is discussed in the next section.

Since composite tasks contain more than one child task, and each child has its own state, therefore $\overrightarrow{S}$, which uses a vector space of dimension $n + 1$, is also extended from $S$ to record the execution states of all involved tasks. $\overrightarrow{S}$ is a subset of the Cartesian product $S^{n+1}$, because the state of $t$ depends on the progress of its subtasks, and even subtasks are not independent from each other. Before the execution starts, $t$ and its subtasks are all in the Inactive state. However, whenever a $t_i$ is no longer Inactive, neither is $t$. As well, after the execution ends, in spite of the status of its subtasks, $t$ can only terminate as being Successful if both Inv and Post are satisfied, Compensated if only Inv is true, or Aborted when Inv is false.
Because agents run autonomously, during the execution, the agent in charge of the parent task needs to communicate with the agents pursuing children tasks, to collect their state information. The communication protocol is not the theme of the thesis. However, we assume that timeout can imply the Aborted state of that task.

Finally, the constraints set $C$ and the state transition function $\delta$ are extended to incorporate the knowledge of task dependencies. In general, a task should remain Inactive and cannot commence execution unless its preceding task has finished. Moreover, the dependency set $D$ contains more specification about the enabling condition of each subtask. In addition, the maintenance conditions of $t$ can usually be customized to enforce the requirements on the child tasks. For example, if it is required that all children must finish successfully, then $\forall i ((1 \leq i \leq n) \land (\exists [t_i] \neq \text{Compensated}) \land (\exists [t_i] \neq \text{Aborted}))$ is added into $M$. This maintenance rule will guarantee that if any child is in the state of Compensated or Aborted, the parent will go into the Compensating state.

### Reconciliation of Single and Composite Task Perspectives

Although two kinds of tasks, a single task and a composite task, have been discussed above, they map to the same underlying model. Whether a task is viewed as single, or composed of other tasks, depends upon the abstraction level at which it is considered. If we need to access a task’s internal organization, we model it as compositional. If we just need to use its functionality as a building block, we model it as a single task. In short, a single task focuses on what to do while a composite task concerns about how to do.

In the rest of this chapter, we refer to $t$ as the parent task and use $t_i$ to refer to a child from which $t$ is composed. Because we are only concerned with the interface but not the internal execution details of each child task, $t_i$ is treated as a single task. Moreover, $t$ is treated as a composite task for the study of the interaction among $t_i$s. The task execution from a single-task perspective is trivial (Section 3.2.1). Therefore, the next section will talk about how to organize and manage single tasks to support the execution and composition of complex tasks.
3.3 The ARTS Model

ARTS has two primary aspects: task knowledge — procedures — specified by the agent developer, and the execution platform supporting them. This section first describes the specification of the procedures, then describes the method for translating the specification into dependencies, and the management and use of the dependencies for coordinating the task execution.

3.3.1 Exception-Handling Knowledge in ARTS

The programs in ARTS are a set of procedures indicating how to react to the events produced by each agent. As will be detailed below, the procedures have nested scope, so that each plan to handle an exception can be associated in turn with its own set of exception-handling plans specific to that context. Exception-handling plans may encode both “undo” and “retry” semantics: the developer determines the compensation semantics appropriate to the situation.

In this section, we first describe how ARTS uses logical operators to define procedures. These operators include the standard connectives, such as \( \land, \lor \) and \( \neg \) from first-order logic, and three new connectives: \( ; \) for *Sequential composition*, \( \parallel \) for *Parallel composition* and \( , \) for *Selective composition*. All the definitions in this sections are given in EBNF-like format. As discussed above, the possible types of event of interest to the parent are well-defined.

**Definition 3.11 (Event).** An atomic event is signalled directly by the changes of a task’s state. \( \epsilon \) stands for the null event, which always evaluates to be true. \( t_i \) is the ID of a child task.

\[
< \text{internalEvent} > ::= \"\epsilon\" \mid \"\text{Successful}(t_i)\" \mid \"\text{Compensated}(t_i)\" \mid \"\text{Aborted}(t_i)\" \\
\quad \mid < \text{internalEvent} > \land < \text{internalEvent} > \\
< \text{externalEvent} > ::= \"\text{start}(t)\" \\
< \text{Event} > ::= < \text{internalEvent} > \mid < \text{externalEvent} >
\]

An event is defined from the perspective of the parent task. \( \text{Successful}(t_i) \), \( \text{Compensated}(t_i) \), and \( \text{Aborted}(t_i) \) stand for an event signalled internally by subtask \( t_i \) which terminates in a
Successful, Compensated, or Aborted state respectively. By contrast, start(t) is the event captured externally which asks t to commence the execution.

In response to a given event, the parent will compose its children to form reactive plans that address the event. The possible types of task composition are given in Definition 3.12. Sequential composition means that each component will be executed one at a time. Parallel composition means that all components will be launched together, and selective composition means that alternative choices will be tried one by one until one succeeds. Bilaterally, the composition of tasks can be illustrated as an AND/OR graph in which a single edge represents a sequence composition, an AND branch corresponds to a parallel composition, and an OR branch maps from a selection composition. Parentheses are used to identify the boundary of the composition block of each group of task. They can be removed when the representation of the task block is obvious.

**Definition 3.12 (Task Composition).** Task composition serves to organize a group of single tasks according to their functionalities for achieving a certain goal (postcondition of the parent).

\[
< \text{atomicTask} > ::= \text{“} \epsilon \text{”} | \text{“} t_i \text{”} | \text{“} t_i^- \text{”}
\]

\[
< \text{Task} > ::= < \text{atomicTask} > | < \text{sequenceComposition} > | < \text{parallelComposition} > | < \text{selectionComposition} >
\]

\[
< \text{sequenceComposition} > ::= \text{“}(\text{“} < \text{Task} > \text{“}; \text{”} < \text{Task} > \text{“})\text{”}
\]

\[
< \text{parallelComposition} > ::= \text{“}(\text{“} < \text{Task} > \text{“}\text{||} < \text{Task} > \text{“})\text{”}
\]

\[
< \text{selectionComposition} > ::= \text{“}(\text{“} < \text{Task} > \text{“}, \text{”} < \text{Task} > \text{“})\text{”}
\]

In the definition, \( t_i^- \) is the compensation task of \( t \) which is in charge of “undoing” the successful execution of \( t_i \), and \( \epsilon \) is a null task in the system whose Pre, M, Post and Inv are always satisfied. Although \( \epsilon \) does nothing and always succeeds, it acts as a bridge to connect unrelated branches in Parallel or Selection composition. In practice, we assign it a name with the same form of \( t_i \) without treating it differently. For instance, \( t_1 \) and \( t_9 \) in Figure 3.1 are examples of the \( \epsilon \) task.

**Definition 3.13 (Plan).** A reactive plan specifies a composition of a group of tasks to react to an
A reactive plan describes the process of how to deal with a particular event, according to domain knowledge. To simplify the notation, we assume that each atomic task appears at most once in the same plan. The keyword “continue” means to resume the parent task execution, and will be further explained in Section 3.3.4.

In the definition, a plan is equivalent to a task, except that it also contains the triggering condition and the execution context of the task. The task part of a plan specifies what a plan will do, while the event part says when the task should be triggered.

A plan can be converted into a task by treating the event as part of the precondition of its task, and vice versa. Consequently, all discussions about tasks in Section 3.2, concerning their execution states as well as their transitions, are also applicable to plans. Inside a composite task graph, if a task is viewed as a plan, each edge implicitly behaves as a Successful event of the preceding task. However, the introduction of the plan concept enriches the applicable context of tasks, especially for the cases of Compensated and Aborted executions.

As we can see, tasks are organized in a hierarchical structure, in order to handle a given event. In practice, we need different plans to deal with events situated in different contexts. Thus, we define a hierarchy to encode the context of event handling and confine the effective scope of each plan.

**Definition 3.14 (Procedure).** A procedure contains a plan, as well as a set of amendment procedures to deal with exceptions within the scope of the plan. “::” stands for the separator of different procedures.

\[
< \text{Procedure} > ::= (\varepsilon | < \text{Plan} >) \cdot \cdot \cdot < \text{amendmentProcedures} > \\
< \text{amendmentProcedures} > ::= \{ " < \text{Procedure} > \cdot \cdot \cdot " < \text{Procedure} >\}^* \cdot " \}
\]
Each procedure consists of two parts: a plan part to define the purpose and behaviour of a procedure, and an amendment part to deal with exceptions that might occur during the execution of its plan part. This definition allows organization of the plans into a nested structure. The set of procedures, which are separated by “::”, define handlers for possible exceptional events. Only one of them will be selected according to the actual situation of execution.

The introduction of amendment procedures helps and guides developers to identify and organize exceptional situations hierarchically according to the execution context. During the development process, developers first specify a set of plans for normal execution without considering exceptions. They can then specify a set of exception handling plans, which address different exception events, for each normal plan without the interference of others. To continuously improve the system robustness, developers can recursively specify a set of exception handling plans for an arbitrary exception handling plan at a certain hierarchical level. In fact, the development process reduces system complexity by decomposing the overall system into smaller problems and organizing them in a hierarchical structure. Therefore, developers can apply our model to design and manage applications in more complicated domains.

**Definition 3.15 (Procedure Inheritance).** If a procedure Proc\(_d\) is in the amendment part of another procedure Proc\(_a\), then Proc\(_a\) is the ancestor of Proc\(_d\), denoted as Proc\(_a\) ⊑ Proc\(_d\). A procedure which does not belong to any amendment part is called a Root Procedure.

If Proc\(_p\) ⊑ Proc\(_c\), and there is no procedure Proc\(_m\) such that Proc\(_p\) ⊑ Proc\(_m\) and Proc\(_m\) ⊑ Proc\(_c\), then Proc\(_p\) is the parent of Proc\(_c\), denoted as Proc\(_p\) ⋗ Proc\(_c\).

**Definition 3.16 (Plan Inheritance).** Proc\(_a\) and Proc\(_d\) are two procedures. If Proc\(_a\) ⊑ Proc\(_d\), then their plan parts have the same relationship, Plan\(_a\) ⊑ Plan\(_d\). The plan part of a root procedure is called a Root Plan.

If Proc\(_a\) ⋗ Proc\(_d\), then their plan parts have the same relationship, Plan\(_a\) ⋗ Plan\(_d\).

The definitions of procedure and plan inheritance extend Definition 3.9 of task inheritance with a precise organizing structure which encapsulates tasks with more specific execution context. In
addition to the containment relationship between tasks, the situation when a task can be triggered by an event is also taken into consideration.

**Definition 3.17** (Scope of Execution). If $\text{Proc}_p \succ \text{Proc}_i$ and $\text{Proc}_p \succ \text{Proc}_j$, then $\text{Proc}_i$ and $\text{Proc}_j$, as well as their corresponding plan parts, are defined to be in the same scope, within $\text{Proc}_p$. All root procedures/plans are in the same scope.

In this nested structure, a plan is only effective in the scope of its parent plan, which provides the execution context for it. This concept is crucial to understanding the coding and execution specification.

We can now define an ARTS program.

**Definition 3.18** (ARTS Program). An ARTS program consists of a set of root procedures.

An ARTS program can be considered as several independent plan trees whose roots are the corresponding root plans. To make the definitions above more concrete, we use the procedure in Figure 3.4 to express part of the knowledge of the Surgery Manager from Figure 3.1. The figure does not include the specification of task constraints or the procedures for $t$’s subtasks. We introduce two new tasks. The task $t_{12}$ will inform the system administrator that something uncontrollable has happened while $t_{13}$ is the staff scheduler, which assigns a new surgeon.

For simplicity, here we present only a small set of procedures, omitting exception handlers for events that the developer would typically address. As will be described in Section 3.3.4, ARTS specifically supports incremental plan specification, reacting gracefully to gaps in specification of exception-handling plans, and facilitating the testing and debugging of existing specifications.

### 3.3.2 Generating a Dependency Set from Plans

Plans are the building blocks for the agents in a system, used to achieve goals, address problems, and perform restorations. They encode both “normal” business logic, and scoped exception handling knowledge in the form of amendments. Plans are translated to dependency sets, which specify the connections between each participating task and agent, and the transfer of plan control,
based on these building blocks. The dependencies encode the execution paths for each task, as defined by the plans, in conjunction with ARTS’ methodology for organizing and using this knowledge.

Each plan in the program produces a dependency set denoted as $D(p)$. In the following rules, we will first generate the dependency set for task definitions which only use a single type of composition operator, and then recursively group them according to the composition structure to get $D(p)$. During this process, $D(t)$ denotes the dependency set for the task $t$, $first(t)$ represents the set of subtasks one of whose elements could be selected as the first step to be executed in $t$, and $last(t)$ represents the set of subtasks one of whose elements could be selected as the last step to be executed in $t$.

Let $t_1 \land t_2 \land \ldots \land t_n$ stand for all $t_i$ that need to be executed. The expected event produced by $t_i$ is $\beta^i = Successful(t_i)$. The expected event produced by $t_1 \land t_2 \land \ldots \land t_n$ is $\beta^1 \land \beta^2 \land \ldots \land \beta^n$. For a set of tasks $T$, we define the expected event set produced by it as $\beta^T = \{ \beta^t | t \in T \}$. Based on these definitions, we use the predicate $depend(t, e)$ to represent that the triggering of the task $t$ depends on the occurrence of the event $e$.

The composition of tasks can be as simple as a single atomic subtask, or as complex as the recursive combination of composite subtasks with various types of composition operations,
Algorithm 3.1: substitute(t)

Require: task t
Ensure: t', a simple form of t containing no nested structure
1. \( t' = t; \)
2. substitutionSet = \( \emptyset; \)
3. foreach top-level subtask composition block of t do
   4. set substitution task \( t_{bi} = \text{block}_i; \)
   5. update \( t' \) by replacing block\(_i\) with \( t_{bi}; \)
   6. add \( t_{bi} = \text{block}_i \) into substitutionSet;
4. end
7. return \( t' \);

Table 3.3: Decomposition of the surgery preparation task

<table>
<thead>
<tr>
<th>Task Notation</th>
<th>Command</th>
<th>Substitution Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( t = t_1; \left( (t_2</td>
<td></td>
<td>t_3); t_5; t_6 \right); \left( (t_4; (t_7, t_8)); t_9 \right); t_{10}; t_{11} )</td>
</tr>
<tr>
<td>(2) ( t_{2-9} = \left( t_2</td>
<td></td>
<td>t_3; t_5; t_6 \right); \left( (t_4; (t_7, t_8)); t_9 \right) )</td>
</tr>
<tr>
<td>(3) ( t_{2-6} = (t_2</td>
<td></td>
<td>t_3); t_5; t_6 )</td>
</tr>
<tr>
<td>( t_{4-9} = t_4; (t_7, t_8); t_9 )</td>
<td>substitute ((t_{4-9}))</td>
<td>( t_{4-9} = t_4; t_{7-8}; t_9 )</td>
</tr>
<tr>
<td>(4) ( t_{2-3} = t_2</td>
<td></td>
<td>t_3 )</td>
</tr>
<tr>
<td>( t_{7-8} = t_7, t_8 )</td>
<td>substitute ((t_{7-8}))</td>
<td>( t_{7-8} = t_7, t_8 )</td>
</tr>
</tbody>
</table>

such as the definition of the surgery preparation task in Figure 3.4. To generate the dependency set for a complex task composition, we can syntactically transform it into a simple form, in which no nested structure of subtask composition exists, by recursively substituting part of its components for a composite task notation. The process of substitution is given in Algorithm 3.1. Following Definition 3.12, the algorithm uses parentheses to identify the boundary of each subtask composition block. Example 3.6 shows the recursive use of Algorithm 3.1.

Example 3.6. The surgery preparation task in Example 3.1 can be simplified recursively by substituting a sequence of subtasks for a composite one. The step by step decomposition process is shown in Table 3.3.

In Step 1, the complex form of \( t \) is converted into a simple sequential composition. In Step 2, \( t_{2-9} \), a subtask of \( t \), is also represented in a simple form of parallel composition. In Step 3, \( t_{2-6} \) and \( t_{4-9} \), the subtasks of \( t_{2-9} \), are further simplified by substitution. Finally, \( t_{2-3} \) and \( t_{7-8} \) are already in the simplest form; therefore, the function substitute() has no effect on them.
Algorithm 3.2: getTaskDependencySet(t)

Require: t is composed of t₁, t₂, ..., tₙ

Ensure: D(t), the dependency set obtained from t

1. \( t' = \text{substitute}(t) \);
2. \( D(t) = \text{getSimpleDependencySet}(t') \);
3. \( \text{return } D(t) \);

Algorithm 3.2 is designed to compute the dependency set of a task. It first calls the function substitute() to get the simplified form of a task representation, then uses Algorithm 3.3 to compute and merge the dependency set of each subtask recursively.

Since composite tasks each contain a set of subtasks, we first introduce two relations in which the operator “×” represents Cartesian product: \( \text{DEP}(T × E) = \{ \text{depend}(t, e) | t ∈ T ∧ e ∈ E \} \), and \( \bigwedge (T_1 × T_2 × \ldots × T_n) = \{ t_1 ∧ t_2 ∧ \ldots ∧ t_n | t_i ∈ T_i \} \). To simplify the notation, \( \bigcup_{1 \leq i \leq n} \text{Set}_i = \text{Set}_1 ∪ \text{Set}_2 ∪ \ldots ∪ \text{Set}_n \).

The simplest situation is that \( t \) is a single task. It serves as the initial case to produce the set presentation. Based on the set operations, the computation of the dependency set for different types of composition operators is defined. For the sequential composition, \( \text{DEP}(\text{first}(t_i) × \text{last}(t_{i-1})) \) is added to \( D \), since the start of \( t_{i+1} \) depends on the completion of \( t_i \). For the parallel composition, the \( t_i \)s are combined together as a single element in \( \text{first}(t) \) and \( \text{last}(t) \), because they are required to be executed at the same time. Moreover, they do not depend on each other for triggering of execution. By contrast, for a selective composition, each \( t_i \) is considered as an element in \( \text{first}(t) \) and \( \text{last}(t) \), because only one of them can be selected for execution at a time.

Since a plan \( p \) is defined as \( p = e ← t \), it follows that \( D(p) \) can be computed straightforwardly after \( D(t) \), as described in Algorithm 3.4.

### 3.3.3 Organizing the Dependency Set

With the algorithms introduced in the previous section, the dependency set of each plan can be generated separately. As with their corresponding plans, these dependency sets can also be organized hierarchically. To confine the effective scope of a plan, and to distinguish between
3.3 The ARTS Model

Algorithm 3.3: getSimpleDependencySet(t)

Require: t is composed of \( t_1, t_2, \ldots, t_n \)

Ensure: \( D(t) \), the dependency set obtained from \( t \)

/* t is a single task */
if \( n = 1 \) then
  \( \text{first}(t) = \{ t_1 \} \);
  \( \text{last}(t) = \{ t_1 \} \);
  \( D(t) = \emptyset \);
else if \( n > 1 \) then
  /* t is composed by several subtasks */
  foreach \( t_i \) do
    \( D(t_i) = \text{getTaskDependencySet}(t_i) \);
  end
  if \( t = t_1; t_2; \ldots; t_n \) then
    /* t is sequential composition */
    \( \text{first}(t) = \text{first}(t_1) \);
    \( \text{last}(t) = \text{last}(t_n) \);
    \( D(t) = \bigcup_{1 \leq i \leq n} D(t_i) \cup \bigcup_{1 \leq i < n} \text{DEP}(\text{first}(t_i) \times \text{last}(t_{i-1})) \);
  end
else if \( t = t_1 || t_2 || \ldots || t_n \) then
  /* t is parallel composition */
  \( \text{first}(t) = \bigwedge (\text{first}(t_1) \times \text{first}(t_2) \times \ldots \times \text{first}(t_n)) \);
  \( \text{last}(t) = \bigwedge (\text{last}(t_1) \times \text{last}(t_2) \times \ldots \times \text{last}(t_n)) \);
  \( D(t) = \bigcup_{1 \leq i \leq n} D(t_i) \);
end
else if \( t = t_1, t_2, \ldots, t_n \) then
  /* t is selective composition */
  \( \text{first}(t) = \bigcup_{1 \leq i \leq n} \text{first}(t_i) \);
  \( \text{last}(t) = \bigcup_{1 \leq i \leq n} \text{last}(t_i) \);
  \( D(t) = \bigcup_{1 \leq i \leq n} D(t_i) \);
end
return \( D(t) \);

multiple appearances of the same event in different plans, each dependency set \( D(p) \) is tagged with an attribute of the Path of Dependency

Definition 3.19 (Dependency Path). Path of Dependency contains the location knowledge of a plan’s dependency set in the hierarchy. It is defined with a hierarchical naming pattern, with
Algorithm 3.4: getPlanDependencySet(p)

Require: \( p = e \leftarrow t \), \( e \) is an event and \( t \) is a task
Ensure: \( \mathcal{D}(p) \), the dependency set of \( p \)

- \( \mathcal{D}(t) = \text{getDependencySet}(t) \);
- \( \mathcal{D}(p) = \mathcal{D}(t) \cup \text{DEP}(\text{first}(t) \times \{e\}) \);
- return \( \mathcal{D}(p) \);

levels in the hierarchy separated by periods (.). It begins with an event of the root plan, and then recursively traverses the dependent events of each child plan.

With the assistance of the dependency path attributes, the dependency relationship of an ARTS program can be obtained simply by uniting all its plans.

Definition 3.20 (Program Dependency Set). \( p_i \) is a plan in the ARTS program \( P \). The dependency set for \( P \) is

\[
\mathcal{D}(P) = \bigcup \mathcal{D}(p_i)
\]

Example 3.7. The dependencies for the program in Figure 3.4 are shown in Tables 3.4, 3.5, 3.6, 3.7, and 3.8, which are organized hierarchically by the dependency path attribute.

The dependencies in \( \mathcal{D}(\text{SurgeryManager}) \) implicitly form a dependency graph. For each dependency \( \text{depend}(t, e) \), \( t \) is mapped into a node (or parallel nodes if \( t = t_1 \land t_2 \land \ldots \land t_n \)) in the graph, while \( e \) is mapped into the label of the edge (or edges if \( e = e_1 \land e_2 \land \ldots \land e_n \)) coming out from the node(s) whose expected result is \( e \). The attribute of the dependency path is self-explanatory from the graph.

Although this dependency graph of the Surgery Manager can be considered as a flat workflow containing all knowledge about both normal and compensation execution modes, it is beneficial to decompose it for the reasons of modularity, manageability, and scalability. For this reason, ARTS produces a tree-structured organization of the plans. Thus, there are two types of hierarchical
### Table 3.4: The dependency set for \(\text{start}(t)\) of Surgery Manager

\[
\text{start}(t) : \{
\quad \text{# the path of the dependency}
\quad \text{depend}(t_1, \text{start}(t)),
\quad \text{depend}(t_2 \land t_3 \land t_4, \text{Successful}(t_1)),
\quad \text{depend}(t_5, \text{Successful}(t_2) \land \text{Successful}(t_3)),
\quad \text{depend}(t_6, \text{Successful}(t_5)),
\quad \text{depend}(t_7, \text{Successful}(t_4)),
\quad \text{depend}(t_8, \text{Successful}(t_4)),
\quad \text{depend}(t_9, \text{Successful}(t_7)),
\quad \text{depend}(t_9, \text{Successful}(t_8)),
\quad \text{depend}(t_{10}, \text{Successful}(t_6) \land \text{Successful}(t_9)),
\quad \text{depend}(t_{11}, \text{Successful}(t_{10}))
\}
\]

### Table 3.5: The dependency set for \(\text{start}(t)\).\text{Compensated}(t_3)\) of Surgery Manager

\[
\text{start}(t).\text{Compensated}(t_3) : \{
\quad \text{depend}(t_{13}, \text{Compensated}(t_3))
\}
\]

### Table 3.6: The dependency set for \(\text{start}(t)\).\text{Compensated}(t_3)\.\text{Aborted}(t_{13})\) of Surgery Manager

\[
\text{start}(t)\text{.Compensated}(t_3)\.\text{Aborted}(t_{13}) : \{
\quad \text{depend}(t_2 \land t_4, \text{Aborted}(t_{13}))
\}
\]

### Table 3.7: The dependency set for \(\text{start}(t)\).\text{Compensated}(t_6)\) of Surgery Manager

\[
\text{start}(t)\text{.Compensated}(t_6) : \{
\quad \text{depend}(t_2 \land t_3 \land t_4, \text{Compensated}(t_6))
\}
\]

### Table 3.8: The dependency set for \(\text{start}(t)\).\text{Compensated}(t_6)\.\text{Aborted}(t_4)\) of Surgery Manager

\[
\text{start}(t)\text{.Compensated}(t_6)\text{.Aborted}(t_4) : \{
\quad \text{depend}(t_{12}, \text{Aborted}(t_4))
\}
\]

information in ARTS. One is the task tree, which specifies agent tasks hierarchically in terms of tasks and the subtasks from which they are composed. The second is the plan tree for a given task.
level, which organizes plans into nested procedures to define the context for events.

Figure 3.5(a) is the plan tree for the hospital example, abstracting each plan as a node. Figure 3.5(b) shows the complete dependency graph of the program, by expanding each plan node to show its internal task organization, in which the root plan, the same as that in Figure 3.1, is abbreviated and \( \epsilon \) tasks are added to make the relationships more clear. Each rectangle represents a procedure, and the containment relation between them shows the nesting of corresponding plans.

Even though the details on the internal subtask organization of \( t \) are analyzed in Figure 3.5(b), \( t \)'s external interface complies with the definition of an atomic task which is illustrated in Figure 3.2. In other words, whether \( t \) is viewed as being atomic or composite depends only on the abstraction level of the user’s concern.
3.3 The ARTS Model

3.3.4 Execution of Plans

The automatic execution and compensation of a plan is supported by inference rules in ARTS, which act as bridges and directors for the user-defined procedures, and in accordance with the dependency relationships in $D(p)$. For the simplest form of a plan, which consists of an atomic task, its execution is guided and constrained by a set of inference rules derived from the state transitions in Figure 3.3.

However, plans are usually built with composite tasks and organized in a hierarchy to deal with the complexity of real world applications. In the ideal case of exception free operation, the execution of a plan of arbitrary composition can be treated by the context that invoked it as a simple workflow, following the dependency determined by the subtask composition. Correspondingly, the state transition of the parent task is intuitively from Active to Successful along with the activities of subtasks.

To deal with exceptions, especially for a parent plan whose role is to coordinate the execution of its components according to the hierarchical structure of amendment plans, we need to introduce two more types of inference rules, one for plan termination by completion, and another for plan termination by exception, concerning the context switch caused by plan terminations. These rules are summarized from the dependency set of an ARTS program, focusing especially on the reaction to unexpected events.

- Inference rules for plan termination by exception

The first group of inference rules is used to switch the context if an exception $\mu$, which is signalled by a subtask $t_i$ terminating in an unexpected state, happens inside a plan.

1. If there is no child plan to react to $\mu$, $\mu$ is unhandleable for the plan. In this case, $p$ can only enter the state of Aborted.

2. If a child plan is found, and it includes the keyword “continue” — which means the successful completion of this child plan can rectify the problem and allow the parent to continue running — it is triggered to execute directly.

3. If a child plan is found without “continue”, then the agent converts into the state of Compensating and begins to execute that plan.
Example 3.8. In Figure 3.5(a) which is the plan tree of the surgery manager example, the “continue” creates a loop back to the root plan. That is, after the occurrence of Compensated\((t_3)\) during \(t\)’s execution, \(t\) will keep on working in the Active state, because the plan triggered by Compensated\((t_3)\) indicates that the exception can be fully rectified.

■ Inference rules for plan termination by completion

The second group of inference rules concerns the context switch at the successful completion of plan \(p\).

1. If \(p\) is the root of a given plan tree, and there are no other root plans which can react to the last event produced by \(p\), the program will transform the state from Active to Finishing.
2. If \(p\) is the root, and there are other root plans which can react to it, the postcondition of the program will be tested first. If the postcondition is satisfied, the program will enter the state Successful directly; otherwise, the descendent root plan will be triggered to run.
3. If \(p\) is not the root, but it has “continue” at the end, the control should pass back to its parent plan as if the action triggering this amendment plan is successful.
4. If \(p\) is not the root, and does not have “continue”, the parent plan must be already in the state of Compensating and should enter the state of Compensated or Aborted according to the satisfaction of invariants. We do not allow the transition from one non-root plan to another because they are designed merely to amend the exceptions and should be independent from each other.

Example 3.9. In the same example, if \(t_{13}\) returns Successful, \(t\) will continue to run its root plan as if \(t_3\) has succeeded. Two children of the root have no connections even though Compensated\((t_6)\) ← … might produce the event Compensated\((t_3)\).

The plan enters the Aborted state when it has no applicable plans for handling \(\mu\), or the invariants are not met at its completion. In this case, the problem will be dealt with by the parent context in the plan tree. Thus, the Aborted events both signal to the developer those situations where domain information is incomplete, and support a graceful and consistent means of handling...
this missing information.

3.4 Discussion and Conclusion

In this chapter, we have discussed the details on the specification and organization of the ARTS (Agent-oriented Robust Transactional System) model for agent design. By applying concepts of transaction management, ARTS allows a developer to modularly specify task requirements and behaviour with respect to both normal and exceptional situations, allowing tasks to be treated as “atomic” building blocks, and reducing the complexity of coordination and composition in a system of agents. ARTS supports a model of hierarchical task and plan composition, which increases system robustness and reliability by constraining the way in which agents interact, enforcing the agents’ work towards achieving higher-level goals, and enforcing the use of repair models in a predictable and consistent manner.

Agent systems and workflow systems overlap significantly in terms of their approaches to task composition [160]. For many years, the workflow research community has considered extended transaction models as an essential way to ensure the correctness and reliability of applications. Various extended models have been proposed and applied in the field of transactional workflow systems [175]. They suggest that nested structure, compensation, and forward recovery are the most beneficial concepts in building robust and reliable workflow systems. Our proposal also follows these principles, but expands their applicability to the world of agents, which is more dynamic, non-deterministic, and difficult to model. Conversely, our approach can also be applied to a workflow system directly, with added benefits such as verification and validation.

Compared with existing research in the application of transaction concepts to agent systems [19, 110, 121, 155], ARTS does not apply existing extended transaction models directly to agent systems, since these models are inappropriate to an open and dynamic environment in which agents do not have total control over resources: that is, undo/reverse operations cannot be performed in all situations. For example, after firing a rocket, there is no way to “unfire” it. Rather, we assimilate the essential concepts from these models to design a flexible and well-organized agent recovery structure, from a programming and software engineering perspective, with the focus on task composition as well task interrelationships. ARTS’ hierarchical organization model
for agents and their exception handling knowledge provides developers with a powerful abstraction for modeling exception events and for encoding knowledge of exception handling plans and the way that these plans relate to each other.

As we have discussed, our model does not define a specific new agent programming syntax, nor is it targeted at integration into a specific existing agent system. Rather, our model and programming paradigm can be used as a transactional layer residing between the upper level of agent applications and the lower level of existing agent programming platforms, to provide architectural-level support for robustness and reliability. In particular, ARTS is consistent and integrable with the BDI agent model. ARTS can be viewed as imposing constraints on how certain classes of knowledge about a task should be represented in an agent’s belief base, how they must relate to each other, and how this knowledge should constrain the agent’s deliberation cycle selection. In ARTS, within a task context, each agent pursues the implicit goal of “complete the execution” or “complete the compensation”. After decomposing plans into sets of dependencies, the agent can use its belief base, which consists of both its running state and the outcomes of participating actions, to guide the correct selection of the next task.

Equipped with this agent model, developers can focus on higher-level concerns of handling integration exceptions. Because this design also ensures the engineering principle of modularization and reduces the coupling among agents, it helps programmers control the complexity introduced by a wide range of exceptions, and allows them to incrementally improve system robustness.
Chapter 4

Multiple-step Backtracking in BDI

Transactional Agent Execution

In Chapter 3, we have discussed task structure and plan organization in the ARTS model, which relies on users to specify amendment plans for the purpose of exception handling. However, it is also common in multi-agent environments that unexpected exceptions occur to interrupt execution towards the goal. Therefore, in this chapter, we will introduce multiple-step backtracking, which extends the existing step-by-step plan reversal, to serve as the default exception handling mechanism in ARTS.

Replanning, which discards all alternatives found in previous execution, and step-by-step backtracking, which iterates back through all previous selection decisions, are two extreme ends of exception handling in agent systems. We describe a multiple-step backtracking model which maintains a tradeoff between them. Choice points, which are maintained in a stack, are introduced to record all plans found by the agent’s planning procedures. By iterating on the stack, and reasoning about plan characteristics, an agent can find and launch a suitable plan prior to or in parallel with a compensation process for the failed path, thus achieving its goals efficiently in the presence of exceptions. Our approach combines and utilizes several beneficial features of a Belief-Desire-Intention (BDI) agent, such as its data structure and deliberation cycle, together with an open nested transaction system which supports architectural-level concurrency control and distribution management. Experiments show that these systems, even though they are composed of unreliable agents, are more reliable and robust in achieving the design goal with higher efficiency.
when the multiple-step backtracking model is applied.

4.1 Introduction

With its features of event-driven and means-ends reasoning, the BDI (Belief, Desire, Intention) [131] agent is a powerful high-level decomposition and abstraction tool in analyzing, designing, and implementing complex software systems [77]. However, in order to fulfil the requirement of robustness and reliability with respect to the correct execution of agent actions even in the presence of abnormalities, developers are forced to consider low-level details of disturbances, failure, or uncontrolled interactions between agents, because of the lack of architectural-level support for exception handling.

Many fault-tolerant approaches for automating exception recovery and execution resumption have been proposed to make agent programming more effective and easier for developers. At one extreme is replanning, which discards all previous work when an exception occurs, and starts the planning process again with the current state of the world. However, replanning is computationally expensive and rarely practical, because it requires extensive knowledge about the environment and business domain, requires a lot of programming, and often wastes resources. At the other extreme, backtracking may be applied to attempt to reverse through work previously done in a depth-first manner of search. Despite the fact that backtracking requires a rigid execution tree structure to be feasible, it is widely applied for exception handling in multi-agent systems, because it maps nicely to the hierarchical program architecture, and enables systematic and platform-level recovery support.

Most existing approaches, such as [155], Jason [16] and SaGE [144], apply the conventional rigid backtracking mechanism that traces back step by step until one substitutable path is found. Moreover, they follow the defensive route of the recovery-then-try pattern. However, due to the dynamic and nondeterministic features of complex environments, rigid recovery in reverse chronological order of execution history will in many instances become meaningless and inappropriate. Rigid backtracking is also inefficient in achieving goals in the presence of exceptions, because it sequentializes the procedures of exception handling and goal achieving.

To address this issue, we propose a flexible multi-step backtracking model to maintain a
4.1 Introduction

A tradeoff between replanning and traditional backtracking strategies. Our model extends the existing rigid backtracking strategy, to support and enable execution backtracking in reverse chronological order with certain steps skipped. Instead of step-by-step backtracking through the execution tree, the system can “jump” back to an arbitrary history node to continue its execution towards the goal, when the agent knows going back one level in the execution tree cannot solve the problem. Therefore, achieving its goals resides in the center of the agents’ concerns, and the requirement of maintaining a static execution structure for the purpose of reversed recovery is relaxed. The model assigns higher priority to goal achievement than exception handling, since the latter can delay the recovery process when an eligible execution path toward the goal exists.

As the recovery of failed execution and the trying of alternatives might run in parallel, concurrency control becomes a key problem. To provide automatic and system-level support for concurrency control and exception recovery, an open nested transaction model [170] is integrated into our multiple-step backtracking model.

Similar work has been carried out to apply transaction models [66] to multi-agent systems [19, 110, 155]. However, existing work has primarily treated agents as execution entities in nested transactions, without considering agents’ features of activeness and autonomy, and their dynamic and non-deterministic execution environment, which often makes direct utilization of transaction models infeasible. Even if applying existing transaction models straightforwardly is applicable in theory, as discussed in [124], it is likely to cause problems with long-running transactions or cascade rollback.

Compared with previous approaches, our model unites the beneficial features of event-driven and means-end reasoning from BDI agent systems and utilizes a flexible backtracking approach to allow the execution to “jump” back several levels at once in order to continue its execution towards the goal, in the case that backtracking one level in the execution tree does not solve the problem. Experiments prove that systems composed of unreliable agents can achieve better robustness and reliability when applying our multiple-step backtracking model.

The remainder of this chapter is organized as follows. Section 4.2 begins with a motivating example to show the necessity of a flexible backtracking model. Section 4.3 describes details and algorithms of the multi-step backtracking model. Section 4.4 explains how the open-nested transactional model is integrated into the features of BDI agents, and the use of our
backtracking model to support robust execution. The experimental design and results are presented in Section 4.5. Finally, in Section 4.6 we present a summary of and conclusions about our approach.

4.2 Motivation

During the deliberation cycle of a BDI agent as described in Section 1.3.1 and Figure 1.3, the agent may find more than one feasible plan (the selective composition in ARTS) when processing a goal. These plans are grouped together and recorded in the agent belief set as a choice point, which is a key concept in our model.

**Definition 4.1 (Choice Point).** A choice point of the agent records the applicable plans found by the plan matcher to deal with a certain event $e$. It is denoted as $cp_e = \{p_1, p_2, \ldots, p_n\}$, where $p_i$ is an applicable plan. $p_{i}^{cp}$ stands for the selection of plan $p_i$ from the choice point $cp$ by the plan selector.

To illustrate our flexible backtracking strategy for exception handling and its related transactional execution framework, a travel management agent with the goal of preparing a holiday for the customers is represented.

**Example 4.1.** A group of execution plans, as shown in Table 4.1, is defined for the goal of holiday arrangement. The plan, in the form of event $\leftarrow$ preconditions $|$ action_sequence, can be transformed to and from the program specification of ARTS.

The organizational structure for goal and plan decomposition can be modelled as an execution tree, as illustrated in Figure 4.1. The multiple choice points of the plans are depicted as side-by-side rectangles.

In fact, atomic tasks, such as `FlightBooking()` and `HotelBooking()` in the example, can be expanded further when we need to consider how these tasks are processed. For instance, there are different ways to do a hotel booking, such as to call the hotel or to reserve online. Whether a task is viewed as atomic, or composed in a hierarchical structure, depends upon the abstraction level.
4.2 Motivation

Table 4.1: Plans for travel agent

<table>
<thead>
<tr>
<th>Plan</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrangeHoliday</td>
<td><code>true</code></td>
</tr>
<tr>
<td>chooseDestination()</td>
<td><code>true</code></td>
</tr>
<tr>
<td>chooseDestination()</td>
<td><code>true</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Hawaii</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Hongkong</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>California</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>chooseEntainment()</td>
<td><code>true</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sailing</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Disneyland</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

at which it is considered. If we just need to use the functionality of a task as a building block, we model it as being atomic; otherwise, its internal structure should be considered.

All alternatives at the choice points are eligible for selection. Let us assume the agent chooses California as the tour destination from the three possible locations. The agent then tries to book the flight and the hotel. After that, it chooses Sailing as the entertainment plan. Finally, it begins to rent a yacht. Branches not selected during the execution are not shown, in order to simplify the presentation.

If yacht renting fails, a traditional backtracking mechanism will go back to the choice point of Entertainment and try Disneyland. If tickets for Disneyland are sold out, the system needs to keep on backtracking by undoing hotel booking and flight booking to the choice point of Destination, and try either Hawaii or Hongkong as the new destination.

However, this rigid step-by-step backtracking approach is sometimes not appropriate in a dynamic and complicated environment.

- Its requirement of keeping the execution structure static for recovery purpose is hard to fulfill in a dynamic and non-deterministic environment, and recovery in the reverse order of the execution history will probably become meaningless in many situations. For example, the closure of the hotel after the booking will invalidate any rollback or compensation attempts and result in a discontinuity in achieving the goal of holiday travel.
- After exceptions occur, step-by-step backtracking requires the complete compensation of
failed plans before trying any further efforts to achieve the goal. Because applying other plans to achieve the goal may not always conflict with the process of compensation, this requirement may result in inefficiency by forcing the sequential execution of the recovery and forward processes, and sometimes bring unnecessary costs. For instance, after the agent finds neither Disneyland nor Sailing in California is possible, and it also knows there is still enough money left for a holiday arrangement in another place, it can, instead of waiting for the cancellation and refund from the hotel and flight bookings in California, “jump” back directly to the choice point of \textit{Destination} to continue its execution by trying Hawaii or Hongkong.

Addressing these issues, we propose a multiple-step backtracking mechanism for recovering exceptions that occur in multi-agent systems running in dynamic and open environments. In our approach, when the system knows going back one level in the execution tree cannot deal with the exception, it can jump back several levels at once instead to continue its execution towards
the goal. We will also show how the open nested transaction model is integrated into our flexible backtracking approach, to provide systematic and automatic support for exception recovery and concurrency control.

4.3 Multi-step Backtracking Recovery for Exceptions

This section explains the details of the multi-step backtracking mechanism, which extends traditional step-by-step backtracking in two dimensions: allowing the backtrack of arbitrary steps in the plan tree, and allowing the concurrent run of forward execution towards the goal along with the backward recovery of the failed plan. We first analyse the features of plan organization and execution to reflect the flexibility of agent execution. Then the key concepts of the proposed backtracking mechanism, as well as the algorithm for recovery, are introduced. Finally, our model is compared with existing backtracking approaches.

4.3.1 Principles of Agent Execution

An agent in a system is typically programmed to accomplish a range of goals. When pursuing a certain goal, a BDI agent applies appropriate plans to break the goal into subgoals, performing some of them itself, and delegating some to other agents. The process towards a goal usually produces a goal-plan tree [150], which shares the same semantics as the task and plan composition in ARTS. Plans represent an agent’s knowledge about the real execution structure; therefore, we focus on analysing the features and roles of plans in the agent life cycle.

As we have discussed in Chapter 3, a plan may contain complex internal structure with respect to the goal decomposition, but it is its interfaces, not its implementation details, that are the concern of the user. The execution details of the plan are a black box for its caller.

To achieve a goal, a BDI agent organizes its plans in a hierarchical and nested structure. The goal is achieved once the execution of a sequence of plans terminates in a consistent state, satisfying the goal. However, if any plan results in an inconsistent state during the execution, the system also becomes inconsistent and requires a corresponding recovery plan to correct the inconsistency.

If a subplan fails to terminate in a consistent state that satisfies the invariants, even after the
compensation attempts, the work of the entire plan structure cannot progress further and needs to be recovered. This is because it is difficult, if not impossible, to prove the correctness or consistency of results derived from inconsistent beliefs. If the plan is continued on the flawed data, subsequent actions may even become harmful. For example, if a plan of the travel agent terminates in an inconsistent state in which the summation of money spent and remaining is not equal to the budget, no one can tell if the amount of remaining money is correct or not. Therefore, any further execution based on the false number becomes meaningless.

By modelling agent execution as a hierarchical tree structure of plan compositions, we can analyse various agent behaviours on participating plans, thus summarizing several principles underlying execution management.

**Principle 4.1.** The execution of a plan usually has side-effects which change the environment. If the plan terminates in a consistent state, that is, the resulting system state is also consistent, this state is acceptable, even if the plan actions are later shown to be futile in working towards a goal.

This rule allows plan execution of agents to be viewed in terms of small, independent plan fragments. When a plan does not satisfy its termination conditions as expected, for example, as a result of exceptions, the result can be tolerated by the system as long as it is consistent. In other words, recovery is not necessary for the plan to “undo” its side-effects. On the contrary, the system can continue execution towards the goal by accepting the result of the plan as facts.

**Example 4.2.** Referring to Figure 4.1, let us assume that the travel agent cannot arrange any entertainment in California after the successful completion of flight and hotel bookings. Since the system is still consistent, the existence of bookings does not prevent the agent from trying other options in Hawaii or Hongkong. Moreover, it is more important to achieve the goal of holiday arrangements for the agent than to undo its payment of the booking deposit.

**Principle 4.2.** Cleaning up a side effect of a failed selection \( p^e_i \) can have lower priority than trying its alternatives, or achieving the agent’s goal.
Compensation is usually used to release the consumed resources that are necessary for other attempts. However, it is the approach, not the purpose. All unrelated compensations can be processed in parallel with or even after the achievement of the goal, to improve system throughput and efficiency. The applicability of this principle is based on the fact that agent plans are dynamically composed when their events are triggered and their preconditions are satisfied. However, in resource-bounded situations, compensation may have a higher priority, in order to release the limited resources required for the execution of other plans.

**Example 4.3.** The travel arrangement for a traveller with abundant funds can try different options at the same time to achieve the goal more efficiently. Specifically, the traveller can still afford the cruise booking at Hawaii before getting the refund from the flight and hotel bookings in California. In this way, the time needed for money return is saved. By contrast, for a traveller with a tight budget, the selection of Hawaii would fail because of insufficient money, until the refund from existing bookings is received.

**Principle 4.3.** If an agent is consistent, any plan in its plan library can be launched, as long as the plan preconditions are met.

These rules allow us to continue agent execution at any legal choice point after current execution encounters interruption. In other words, it is not necessary to conservatively roll back before trying other paths. The execution flow just needs to “jump” back to an appropriate choice point and continue. We will give a more detailed description of this concept in the next section.

### 4.3.2 Key Concepts for “Jumping”

An agent can use knowledge of its set of choices at each choice point to decide which substitutable plan to choose if exceptions occur later.

**Definition 4.2** (Choice Point Stack). A choice point stack contains the choice points an agent has met chronologically. It is denoted as \( cp_{\text{stack}} = \{cp_n, cp_{n-1}, \ldots, cp_1\} \) where \( cp_i \) is the choice point occurred at time \( i \).
Example 4.4. Figure 4.2 shows the choice point stack of the travel example after the arrangement is made at California. It contains two choice points: Destination and Entertainment. The available plans at each choice point are represented as a linked list.

When there is a failure of execution, it must happen at the branch of $cp_n$, which is the newest entry in the stack and marks the node of current execution. The agent can search through its stack for another applicable branch of a past choice point to continue achieving its goal. The newly selected branch may potentially be identical to the failed one, initiating a retry. We call this process a “jump”.

**Definition 4.3 (Jump).** A jump is a continuation of agent execution flow at another selectable plan, implied by elements of its choice point stack. To enable the jump to the $j$th choice of $cp_i$, the preconditions of the plan $p_j^{cp_i}$ must be satisfied.

The “jump” method differentiates from traditional step-by-step backtracking in that it allows the continuation of execution at any eligible plan in the traversed part of a plan tree. Preconditions of a plan usually specify the required resources and execution context, which measure its applicability for the “jump” selection. For example, after the failure of the hotel booking, the travel agent may find there is not enough money left for Hong Kong or Hawaii because money has been paid for the air ticket to California.

In summary, the failed branch ($p_f^{cp_n}$) and the new selected branch ($p_i^{cp_i} | i \leq n$) may have three types of coordination with respect to their sharing of or competition for resources. Each different type of coordination leads to a different processing strategy, with respect to allowable execution.
4.3 Multi-step Backtracking Recovery for Exceptions

sequences between the forward execution of $p_s^{c_{ps}}$ towards the goal, and the backward recovery of $p_f^{c_{ps}}$.

1. $p_f^{c_{ps}}$ does not consume any resources which will be used by $p_s^{c_{ps}}$. That is, $p_f^{c_{ps}}$ has no or read-only access to the resources which must be guaranteed as preconditions of $p_s^{c_{ps}}$. In this case, $p_s^{c_{ps}}$ can be started directly, prior to addressing the failure. However, there could be a background thread recycling resources consumed by $p_f^{c_{ps}}$.

2. $p_s^{c_{ps}}$ shares some resources with $p_f^{c_{ps}}$. However, there are still enough resources left for $p_s^{c_{ps}}$ after the consumption by $p_f^{c_{ps}}$. The handling method of this case is the same as the first type. As shown in Figure 4.3 (a) and (b), the execution can jump directly to arrange travel to a different destination, because the flight and hotel bookings have spent nothing, or only a small amount of money. The refund process can be ignored, processed in parallel, or even started after the goal of the agent has been achieved.

3. $p_s^{c_{ps}}$ shares some resources with $p_f^{c_{ps}}$, of which too much has been used for $p_s^{c_{ps}}$ to remain applicable. In this case, compensation must be applied first, before execution can continue at $p_s^{c_{ps}}$. As shown in Figure 4.3 (c), because the bookings of flight and hotel has required too much money, the agent must obtain the refund before continuing towards the goal.

Based on these different types of coordination, we have designed an innovative failure recovery algorithm to achieve non-stop execution towards the agent’s goal. It is embedded in an open nested transaction structure tailored for BDI agents, thus providing an agent system with higher efficiency and throughput when dealing with exceptions, and architecture-level support of concurrency.
Algorithm 4.1: Execution flow of “jump”

Function buildStack()
1 if number of eligible plans > 1 then
2 save plan choices into choice point stack;
3 end

Function evalAndTry()
4 foreach choice point cp_i (i from n to 1) do
5 foreach plan p_j \( cp_i \) at choice point cp_i do
6 if isApplicable(\( p_j^{cp_i} \)) then
7 continue at \( p_j^{cp_i} \);
8 compensate previous work after \( cp_i \);
9 remove \( cp_{n-1}^{cp_{i+1}} \) from stack;
10 break;
11 end
12 end
13 end
14

Function backOneStep()
15 randomly select \( p_r^{cp_n} \);
16 add preconditions of \( p_r^{cp_n} \) as new subgoal;
17

Function jump()
18 if agent execution fails then
19 if evalAndTry() fails then
20 backOneStep();
21 end
22 end
23

control.

4.3.3 Recovery Mechanism and Algorithm

Algorithm 4.1 provides details about the overall execution flow of the “jump” method. It essentially consists of three parts, which operate on the choice point stack data structure.

The choice points (including their sets of choices) encountered during execution are stored into a stack as part of the agent’s beliefs. The stack is built and updated during agent execution in accordance with the traversal along its execution tree structure. As the agent traverses down
the execution tree, it pushes each choice point it meets onto the stack. When there is a failure preventing forward execution, the execution jumps back to a previous choice point. In the process, all choice points after the selected one are removed from the stack altogether. Note that the selected choice point becomes the top element of the stack, indicating the current execution flow.

**Theorem 4.1.** If a choice point $cp_i$ appears in the stack, all its ancestor choice points are present in the stack at the same time. If $cp_i$ is the top element of the stack, only its ancestor choice points are in the stack.

*Proof:* Let $cp_a$ be the ancestor choice point of $cp_i$. There are only two situations in which $cp_a$ is not present in the stack: $cp_a$ has not been visited at all, or $cp_a$ has been jumped over. In the first case, $cp_i$ will also not have been visited, and in the latter case $cp_i$ will also have been jumped over. However, this is in conflict with the fact that $cp_i$ is in the stack. Therefore, all ancestor choice points of $cp_i$ are in the stack.

Let $cp_j$ be a choice point in the stack which is not an ancestor of $cp_i$; then $cp_j$ must either be a descendant of $cp_i$, or have no relationship with $cp_i$ at all. In the first case, where $cp_j$ is the descendant of $cp_i$, $cp_i$ cannot be the top element of the stack because $cp_j$ should be visited later than $cp_i$. In the second case, $cp_j$ is not on the path from the root to $cp_i$. In other words, $cp_j$ cannot have been visited, since it is not reachable in the execution towards $cp_i$; thus it cannot be on the stack. Therefore, only ancestor choice points of the top element are on the stack. ▲

The `buildStack()` function is invoked by the plan selector in Figure 1.3 to build up the choice point stack for the “jump” process.

The `evalAndTry()` function is the core method, which searches through the stack to find a qualified plan to launch in case of failure. This method utilizes the event-driven feature of BDI agents. During this stage, compensation of a failed branch may be carried out in parallel with or after the execution of a new selected branch. In the worst case, `evalAndTry()` needs to search through the whole choice point stack. However, the complexity of search depends on the application, and in general the impact of the search process on the overall performance can be ignored with respect to the task execution. Moreover, the search process can be performed in parallel by another machine as compensation in many situations can be decoupled from the application itself.
The `backOneStep()` function randomly selects a branch at the last choice point and creates new sub-goals to achieve its preconditions if “jump” cannot find an appropriate substitutable plan. This step relies on the means-end reasoning ability of BDI agents, and pauses the agent until compensation or other measures are taken.

The compensation process generally comes from two methods. One is to compensate back step by step like sagas [56]; another is to generate a compensation plan for all the backtracked steps together [46].

With the choice point stack, the method `jump()` can be added into the agent deliberation cycle as a general exception handling mechanism, as shown in Figure 4.4.

For the failed branch, its compensation may be carried out in parallel with, or after the execution of, its alternatives. Thus, the result of the compensation may falsify the validity of the plan chosen in the “jump” process. However, if the compensation does not over-correct, but only partially or completely recovers existing side-effects, the order of execution will not affect the final result. Non-over-correction is denoted as $\text{comp}(r) \leq \text{invl}(r)$, where $\text{invl}(r)$ represents the amount of involved resource $r$ in normal execution and $\text{comp}(r)$ is the reversal of $r$ in the corresponding compensation. For example, if there is a refund for a flight booking, the agent will not pay more cancellation fines than the airfare, nor the air company refund more than the airfare.

**Theorem 4.2.** The result of compensation of the failed branch ($p_{cp}^f$) will not affect the validity of the execution of the new selected branch ($p_{cs}^c$) if the compensation has the feature of non-over-correction.

**Proof:** If the compensation occurs before any other progress is made, it cannot invalidate the later
4.3 Multi-step Backtracking Recovery for Exceptions

“jump” process.

If a compensation was to occur after the “jump”, the initial resources \( r \) and the minimum requirement \( r_{\text{min}} \) for \( p_{j}^{cp} \) and \( p_{s}^{cp} \) satisfy \( r \geq r_{\text{min}} \). Otherwise, they cannot be saved as choice point options in the stack. Therefore,

- If \( p_{j}^{cp} \) consumes resources \( \text{cons}(r) \), the selection of \( p_{s}^{cp} \) means \( r - \text{cons}(r) \geq r_{\text{min}} \). Thus, the compensation is unable to invalidate the execution of \( p_{s}^{cp} \) after releasing some resources back.
- Conversely, if \( p_{j}^{cp} \) produces resources \( \text{prod}(r) \), we get \( r + \text{prod}(r) - \text{comp}(r) \geq r \geq r_{\text{min}} \) under the feature of non-over-correction after the compensation. Then the precondition of \( p_{s}^{cp} \) is still guaranteed.

In all cases, the compensation of \( p_{j}^{cp} \) cannot affect \( p_{s}^{cp} \), as long as the compensation is non-over-correction.

4.3.4 Relation to Existing Approaches

In stateful exception recovery mechanisms (Section 2.2), rigid backtracking, which goes back step by step in a defensive route of a recovery-then-try pattern until one substitutable path is found, is commonly applied, since it can be carried out systematically and protects existing work as much as possible. Moreover, it can be mapped to nested transaction structures and realized by compensation. However, rigid recovery along the reverse chronological order of execution history will in many instances become meaningless and inappropriate because of the dynamic and nondeterministic features of complex environments.

Our model is essentially a variant of backtracking strategy supporting execution backtracking. However, we do not follow the standard backtracking path of \( \text{compensate}(cp_{n}); \text{try}() \), where \( \text{compensate}(cp_{n}) \) compensates the work done after the latest choice point \( cp_{n} \), and \( \text{try}() \) selects a different branch to run. Here, “;” denotes sequential execution while “||” denotes parallel.

Instead, we argue that achieving the goal has higher priority than performing compensation. Thus, we introduce the operation \( \text{evalAndTry}() \), which evaluates each choice point to find if there is a directly applicable plan. If one is found, it is launched without waiting for the completion of compensation. So, the backtracking in our approach follows the path of
Multiple-step Backtracking in BDI Transactional Agent Execution

evalAndTry() || compensate(). If this procedure fails to find an applicable plan from any branch of the choice point stack, the execution pattern is converted to try(); compensate(cp_n). The compensation must complete before try() if and only if the previous execution has consumed too much of a resource to allow continuation. Our approach will have higher throughput because compensation and the substitutable plan can execute in parallel, and goal achievement becomes more efficient.

The approach implicitly makes the assumption that certain domain knowledge, such as goal preconditions and action effects, can be modelled with sufficient accuracy for such decisions. If this were not the case in some contexts, compensation can make subsequent actions more robust, since it helps avoid interactions that are not well modelled. Thus, doing the compensation first can be adopted as the default handler to make the system more robust.

In fact, we can make some modifications to the last two functions of Algorithm 4.1 to simulate different backtracking strategies. For example, if evalAndTry() is constrained so that it will not return a directly executable plan, our approach becomes very similar to the standard one.

We also integrate the open nested transaction model into our multiple-step backtracking model to provide automatic and system-level support for concurrency control and exception recovery. In contrast to existing approaches that apply transaction models directly, we integrate essential concepts of transactions, such as failure atomicity, concurrency control, nested structure, compensation, and forward recovery, into a BDI architecture, to design a flexible and well-organized agent recovery structure from a programming and software engineering perspective. Our approach maximizes the sound features of the BDI model, so as to handle exceptions and to support robust agent execution.

In previous sections, we discussed plan execution in the context of single agents. However, each part of a plan, or subgoals, can be delegated to other agents, making our model applicable in a multi-agent environment as well.

4.4 Transactional Execution of BDI agents

Traditional transactions presume that a transaction has full control of, and exclusive access to, its resources. However, the assumption no longer holds in agent environments, thus making the
direct application of a transaction model to agent management usually not feasible. In contrast to database applications where transactions can lock records to get exclusive access, and can restore any data from history, agents usually work in an open environment and operate on physical objects where actions “always commit” and it is impractical or even impossible to satisfy either requirement. For instance, a flight reservation agent cannot lock the schedule to avoid flight changes, or restore a bank account to the original amount by itself when cancelling a booking.

Although the basic presumption of transaction models is no longer held in an agent environment, transactions still provide invaluable concepts and features (e.g., failure atomicity, concurrency control, nested structure, compensation, and forward recovery) which can be used in components that form a part of an agent system, especially for the problem of error handling and recovery.

In fact, the inapplicability of the transactional model is caused by the dynamic and non-deterministic execution environment of agents, and should be resolved in the agents’ framework, which provides facilities to deal with this kind of complexity and difficulty. We can benefit more from the features of event-driven and means-end reasoning for the purpose of robust execution in agent systems. Therefore, open nested transactions are combined with a multiple-step backtracking recovery strategy to manage agent execution, thus providing flexible, robust, and automatic exception-handling support.

### 4.4.1 Transactional Encapsulation of Plan Execution

Open nested transactions [99, 170] support flexible interaction between parent and child transactions to achieve a higher degree of concurrency. Specifically, subtransactions are allowed to abort or commit independently of their parent. In the case that a parent transaction aborts, its effect, including the result of committed children, is undone by executing a compensation transaction.

The goal decomposition and plan execution of agents can be organized and managed by the open nested transaction model, in which each plan is treated as a transaction, and sub-plans are treated as sub-transactions. This integrated model can provide the agent execution platform with system level support for automatic recovery and concurrency control:

- The integrated model guarantees the modularity and failure atomicity of plans. Thus, programmers only need to deal with two situations: the plan succeeds correctly, or
Multiple-step Backtracking in BDI Transactional Agent Execution

terminates in an incorrect, but consistent, state. For example, if renting a yacht fails, the agent will return back to the choice point of Entertainment, with the details about the incomplete deal with the yacht company abstracted out.

- The integrated model enforces the automation of recovery and resumption. For example, after returning back to the choice point of Entertainment automatically, the agent can retry renting again, or choose to visit Disneyland instead.
- The nested structure of transactions also elegantly matches the tree structure of plan composition.
- The integrated model provides inter- and intra-plan level support for distribution and concurrency control.

If plan execution results in an inconsistent state, recovery is required to deal with the inconsistency, or to release the resources it has held. In this case, as we have discussed in Chapter 3, each plan can be accompanied with a set of corresponding recovering plans for different exceptional situations, to enforce and support plan failure atomicity [79, 146]. Consequently, inconsistent results of plan execution will be recovered by automatically executing the recovering plan. However, if there is no recovering plan defined for a specified exceptional situation, it will cause the system to terminate in an inconsistent state. Even if a plan commits successfully, the changing environment may require it to cancel its results later. In this case, the “undoing” is carried out by a separate compensating plan.

The topics of recovering/compensating plans are out of scope of this thesis; we refer to [46, 145, 153, 154, 168] for details.

The internal structure of a plan may contain hierarchically organized sub-plans. Section 3.3.4 has discussed the dependency between the state transition of the parent plan and the execution of its children. In addition, the execution state of the parent also depends on the backtracking decisions made by the agent. When a sub-plan \( p_i \) of the plan \( p \) fails, \( p_i \) will enter the state of Compensating, and eventually Compensated or Aborted. The Aborted state of \( p_i \) causes the state of \( p \) to become Aborted as well, because of the inconsistency of the agent state. However, the Compensated state of \( p_i \) will trigger the function \( jump() \), to decide which prior choice point is selected to continue. If the chosen choice point is still a descendent of \( p \), which means that another sub-plan of \( p \) is applied to achieve the goal, then the state of \( p \) remains Active. Otherwise, \( p \) needs
4.4.2 Transaction Management

To increase the efficiency of goal achievement, the multiple-step backtracking model allows the compensation of futile execution in the failed path and of the attempts of alternatives to run in parallel. As discussed in Section 4.3.2, concurrently executing plans may access the same resource and cause conflict. For example, as shown in Figure 4.5, after the failure of yacht renting, the execution flow jumps back to Hawaii for holiday arrangement. Consequently, the compensating of hotel booking for California and new cruise booking for Hawaii both need to access the customer’s bank account concurrently.

To avoid conflicts caused by plans’ concurrent access of shared resources, such as bank accounts or available hotel rooms, traditional lock mechanisms are applied. However, it is a requirement that the lock is obtained only when the resource is visited, and released as soon as possible after the visit. Otherwise, it is likely to cause the problem of long-running transactions, which results in low efficiency and even deadlock. As long as the system is consistent, cascading rollback is avoided. In the example, cruise booking in Hawaii and hotel cancellation in California both need to hold the lock when they modify the amount of the money. However, if there is not
enough money available, Hawaii, or other actions, cannot be selected to continue the execution. In this case, the forward execution flow is suspended until a refund is made by flight or hotel cancellation.

We have built a prototype for the travel management agent using 3APL-M [89] and JBoss Transaction Service (formerly Arjuna Transaction Service) [74] to simulate the travel arrangement example.

3APL-M is a lightweight version of 3APL [35], distributed under the GNU GPL (General Public License). Its source code retains simplicity by leaving out supplementary components of 3APL, such as the integrated development environment. It behaves like a programming library whose API (Application Programming Interface) allows a Java application to call 3APL logic and deliberation structures, and vice-versa. Because it is fully integrated with the Java platform and programming environments, 3APL-M is a good prototyping tool for cognitive agents. Further, programs written in 3APL-M can be easily migrated into 3APL, because they share similar underlying language concepts and programming constructs.

The JBoss Transaction Service is employed as the transactional execution manager, which guarantees the isolation of parallel plans. Its TxCore transaction engine supports both closed and open nested transactions, and presents programmers with a toolkit of Java classes which can be invoked by 3APL-M agents directly to obtain desired properties, such as persistence and concurrency control. Programmers only need to start the transaction and set appropriate locks; TxCore will take care of persistence, concurrency control, and recovery.

As shown in Figure 4.1, a root goal of the agent is decomposed into subgoals recursively, resulting in a tree structure. Each subgoal is achieved through a transaction-like function, as shown in Algorithm 4.2. The subgoals are then organized in an open nested transaction structure. Whenever an exception occurs, the execution of that transaction is taken over by the function $\text{jump}()$, shown in Algorithm 4.1. $\text{jump}()$ will fork two threads: one is used to continue the execution at an earlier choice point, and another is used to compensate the failed plan.

Because the rollback operation of traditional database transactions is usually not applicable in an agent environment, it is not invoked in the prototype. Instead, if $\text{jump}()$ succeeds, the side-effects of the failed plan will be undone by its counterpart compensating transaction.

The “jump” algorithm is opportunistic, as it makes the assumption that the failure of one path
Algorithm 4.2: Pseudo-code for the travel agent

Function transactionFlightBooking() {
    beginTrans();
    sequence of actions for flight booking;
    if action fails then
        agent.jump();
    end
    commitTrans();
}

Function transactionHotelBooking() {
    beginTrans();
    sequence of actions for hotel booking;
    if action fails then
        agent.jump();
    end
    commitTrans();
}

Function TravelAgent.run() {
    beginTrans();
    switch agent.select(Destination) do
        case California
            transactionFlightBooking();
            transactionHotelBooking();
            break;
        end
    . . . .
    end
    switch agent.select(Entertainment) do
        case Sailing
            transactionYachtRenting();
            break;
        end
        case Disneyland
            transactionTicketBuying();
            break;
        end
    end
    commitTrans();
}

of execution tree will not block others. In the worst case, where a failure state always holds resources required by other actions, the cost of maintaining and iterating the choice point stack increases. However, as different execution paths increase in independence from each other, the performance is improved.

4.5 Experiments and Evaluations

Simulation is used for evaluating our approach, as it allows performing a statistically significant number of experiments, and makes it possible to explore a wide range of configurations.

During agent execution, a directed acyclic graph or execution tree, in which each node is considered as a task, is generated randomly according to the simulation configurations. Each task in the tree can be executed by the agent itself, or be delegated to other agents for completion. There are two themes of execution during the simulation process:

- **Forward expansion.** After the current task finishes successfully, several feasible subsequent tasks are created as choices, and one of them is selected as the next step to continue the execution towards the goal.
- **Backward tracking.** If the current task fails, the execution flow will traverse back up the existing execution tree to try other options, which are the siblings of the failed task itself or its ancestors, to continue the execution.

The shape of the tree is defined by the parameters $maxChoices$ and $goalLevels$, both of which are used in the forward expansion. The $maxChoices$ parameter defines the maximum number of available continuation choices after a successful task execution; it is therefore related to the width of the tree. The $goalLevels$ parameter defines the successful termination condition for the overall system; that is, the depth of the execution tree.

For each simulation run, a set of resources with $resourceTypes$ elements is declared for participating agents to consume. To simplify the discussion and implementation, without loss of generality we assume that each resource is of a unique type. The amount of each resource is a randomly generated value ranging between $minAmount$ and $maxAmount$. During the forward expansion, a task randomly selects a resource and requests to consume an amount ranging from 0 to the total amount of that resource. If the requested amount is greater than the available one, the
task is not considered as a choice.

A complexity function, which is used to calculate the execution time, is assigned to each task randomly: $a \cdot N$, $a \cdot N \log N$, or $a \cdot N^{3/2}$, where $a$ is a factor randomly picked between $2^6$ and $2^9$ [113], and $N$ is its requested amount of resource.

The robustness of the system is defined by the parameters $P(TaskException)$, which defines the probability that a task encounters an exception and terminates, and $P(TaskRecovery)$, which defines the probability that a terminated task can be recovered or “undone” from its side effects, such as resource consumption. We distinguish exceptions from failures in the sense that if an exception cannot be recovered, the system will fail. The system may also fail when it cannot proceed further and has not satisfied the proposed target. Even in the presence of exceptions, the system should remain consistent, because tasks are capable of failure atomicity.

If $P(TaskException)$ is 0, which means no exceptions at all, then the recovery mechanism will not be invoked. The higher the possibility that the system encounters an exception, the more important the exception handling mechanisms become. Therefore, we will run the experiment under various settings of the following parameters:

- $P(TaskException)$: During the execution, each agent will stochastically decide whether it should signal an exception.
- $P(TaskRecovery)$: After the exception, the agent stochastically decides if it can be recovered.
- ResourceUsageDensity: The value, defined as $goalLevels / resourceTypes$, represents the average number of times that each resource is invoked in a no-exception system execution. The higher this value, the higher the possibility that the resources are exhausted. We let $goalLevels = 20$ in all the experiments.

For all application configurations described above we evaluate and compare the effectiveness of the following two recovery algorithms:

- **Rigid backtracking.** The failed task will be recovered first, and then the system will backtrack one step to continue the system execution. The process repeats until one feasible alternative task is found or the backtrack cannot continue.
- **Jump backtracking.** The agent tries to find a feasible task from its siblings or those of its ancestors to continue the achievement of the goal, and carries out the recovery at the same
Table 4.2: Key parameters in the experiments of multi-step backtracking

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td>maxChoices</td>
<td>maximum number of subtasks</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>goalLevels</td>
<td>the depth of execution tree to the goal</td>
<td>20</td>
</tr>
<tr>
<td>Parameters</td>
<td>resourceTypes</td>
<td>number of different type of resources</td>
<td>various</td>
</tr>
<tr>
<td></td>
<td>ResourceUsageDensity</td>
<td>average utility of each resource</td>
<td>various</td>
</tr>
<tr>
<td></td>
<td>P(TaskException)</td>
<td>probability a task has an exception</td>
<td>[0, 1]</td>
</tr>
<tr>
<td></td>
<td>P(TaskRecovery)</td>
<td>probability a task is recoverable</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>Evaluation</td>
<td>P(SystemFailure)</td>
<td>the probability that system fails</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>Parameters</td>
<td>timeSaveRatio</td>
<td>the time ratio of perfect to real exec</td>
<td>[0, 1]</td>
</tr>
</tbody>
</table>

To simplify the simulation, we assume that the time to recover a task is the same as its forward execution time. We also assume that all resources consumed by the task will be reclaimed after its recovery. The two recovery algorithms above are evaluated and compared with respect to the following two parameters:

- \( P(\text{SystemFailure}) \) which represents the probability that a system fails after applying the recovery algorithm. The lower, the better for this value.

- \( \text{timeSaveRatio} \) which represents the average value of \( \frac{\text{bestTime}}{\text{totalTime}} \) for all executions. \( \text{totalTime} \) is the total or real execution time of a successfully terminated system, while \( \text{bestTime} \) is the least possible time for the completed successful execution. \( \text{bestTime} \) is calculated by cutting all failed branches in the execution tree. Since \( \text{bestTime} \) remains the same for executions under different recovery strategies, the total execution time to complete the task indicates the efficiency of achieving the goal. Therefore, the higher, the better for the value of \( \text{timeSaveRatio} \). In the extreme situation that a run fails, \( \text{totalTime} \) grows to infinity and therefore the \( \text{timeSaveRatio} \) becomes 0.

Even though it is recognized that random graphs do not refer to a real class of application, the diversity they provide seems sufficient for these experiments. A summary of the parameters used in the experiments are listed in Table 4.2.
4.5 Experiments and Evaluations

4.5.1 Simulation Results

The experiments were performed in three categories, to test the effect of parameter $P(\text{TaskException})$, $P(\text{TaskRecovery})$, and ResourceUsageDensity. For each configuration, 10,000 simulation runs were carried out in a batch, and the results were accumulated and analysed. Since the value of goalLevels can affect the result of $P(\text{SystemFailure})$, all experiments in this section set goalLevels = 20. Therefore, the results are comparable with each other.

Effects of $P(\text{TaskException})$

The first group of the simulation uses a fixed $P(\text{TaskRecovery})$ to show the effects on system behaviour of changing $P(\text{TaskException})$. ResourceUsageDensity is set to 2, which means each resource will be applied twice on average in a no-exception execution. $P(\text{TaskRecovery})$ is sampled at 1.0, 0.9, 0.5 and 0.0 to show the different behaviours and results from changing $P(\text{TaskException})$.

The results of $P(\text{SystemFailure})$ and timeSaveRatio in various settings are shown in Figure 4.6 and Figure 4.7, respectively.

The first conclusion is that backtracking can gradually increase system robustness as the percentage of recoverable tasks increases. The trend is more remarkable for rigid backtracking, which has higher reliance on the recoverability of tasks.

Without backtracking, an exception in any task will cause the entire plan to fail. In other words, the success of a high-level plan requires that all participating plans succeed. Therefore, the probability that the system succeeds or fails can be computed as follows.

$$P(\text{SystemSuccess}) = (1 - P(\text{TaskException}))^{\text{goalLevels}} \quad (4.1)$$

$$P(\text{SystemFailure}) = 1 - P(\text{SystemSuccess}) \quad (4.2)$$

The rigid backtracking in Figure 4.6(d) presents the same situation, since no single task can be recovered and backtracked ($P(\text{TaskRecovery}) = 0$).

As shown in the definition of $P(\text{SystemFailure})$, the probability that the system fails is close correlated to the probability that each participating task encounters an exception. However, the increase of $P(\text{TaskRecovery})$ has the same effect as the reduction of $P(\text{TaskException})$ if
backtracking mechanisms are applied.

The second conclusion is that multiple-step backtracking is in general more robust and reliable than rigid backtracking under the same settings. The reason is that the former may jump over and bypass unrecoverable tasks during the backward traversal, while the latter will always fail when a task is unrecoverable.

Finally, multiple-step backtracking performs more efficiently with respect to achieving the goal of the system, because it allows forward execution and task recovery to run in parallel.

It is notable that when every task is recoverable \( P(TaskRecovery) = 1 \), the system failure rate of rigid backtracking is lower than that of multiple-step backtracking. In fact, it represents the best possible value of the system, since it can recover, then try each task in the execution tree, to enumerate every possible execution path. By contrast, some paths may be blocked and missed in multi-step backtracking, since it allows the resource recycling of failed tasks to be delayed.
4.5 Experiments and Evaluations

Effects of $P($TaskRecovery$)$

As a complement, the second group of simulation uses a fixed $P($TaskException$)$ to show the effects of $P($TaskRecovery$)$ on system behaviour. ResourceUsageDensity is also set to be 2. $P($TaskException$)$ is sampled at 0.1 and 0.5 to show the different behaviour under the changing $P($TaskRecovery$)$.

The results of systemFailureRate and timeSaveRatio at various settings are shown in Figure 4.8 and Figure 4.9 respectively.

The results show that $P($TaskException$)$, the probability of task exceptions, has stronger impact on system robustness than $P($TaskRecovery$)$, since recovery is only required after an exception occurs.

When $P($TaskException$)$ is low, there is a strong correlation between the system robustness of rigid backtracking systems and task recoverability. By contrast, multi-step backtracking is hardly affected by $P($TaskRecovery$)$, and outperforms rigid backtracking in maintaining a high system
successful rate and time save ratio, because it usually can find a substitutable task for the failed one, and does not need to recover the failed path beforehand.

However, as $P(\text{TaskException})$ grows higher, both kinds of backtracking strategy behave stably, but poorly. Even high $P(\text{TaskRecovery})$ cannot bring any improvement.

**Effects of resourceUsageDensity**

The last group of the simulation uses a fixed $P(\text{TaskException})$ and $P(\text{TaskRecovery})$ to show the effects of resourceUsageDensity on system behaviour. $P(\text{TaskException})$ is sampled at 0.1 and $P(\text{TaskRecovery})$ is sampled at 0.9 to show the different behaviour and result from changing resourceUsageDensity. Figure 4.10 shows the results of $P(\text{SystemFailure})$ and $\text{timeSaveRatio}$ at various settings.
Resource usage density, with the same goalLevels, has almost no impact on the backtracking mechanisms. It is trivial for rigid backtracking since it will release the consumed resources first, thus having no effect on continuous execution. However, for multi-step backtracking, substitutational tasks may require different types of resources, which helps to distribute the burden on a specified resource.

4.6 Discussion and Conclusion

We follow a common nested tree structure to construct agent execution, and then encapsulate it as an open nested transaction to inherit the benefits of concurrency control and distribution management of participating plans. However, our model does not follow a rigid backtracking strategy on the execution tree. We introduce the concept of choice points, and use a stack to maintain them. By iterating over the stack, the agent can find and launch a suitable plan from previously applicable ones to achieve its goal as soon as an exception occurs. This “jump” procedure strikes a balance between complete replanning and step-by-step backtracking after exceptions happen, by utilizing previous planning results in determining response to the failure. Because the substitutable path is allowed to start prior to or in parallel with the compensation process, an agent can achieve its goals more directly with higher efficiency.

[145] and [46] describe a similar method which recovers from execution problems by backtracking to a past nondeterministic choice point, from which the agent tries to “repair” the causes of failure and then continues. However, their aim is to generate a reverse plan to compensate

Figure 4.10: System behaviour with various resource usage densities

\[ P(TaskRecovery) = 0.9 \text{ and } P(TaskException) = 0.1 \]
to a previous point and retry from there, which also follows the compensate-then-retry pattern. Their approach can be adopted to generate a compensation plan for the failed execution path.

Our approach also frees agent programmers from considering low-level details of concurrency control and exception handling, because transactional execution automates these issues. Combining and utilizing several beneficial features of BDI agents, the open nested transaction model is tightly integrated into the BDI framework. Both BDI data structures and the deliberation cycle are leveraged to maximize the functionality of transaction management.

Our approach requires only that an agent maintain its current consistent state, as well as a choice point stack; in contrast, traditional methods require the agent to remember all its execution history, including choice points and the full execution paths. Thus, our model is comparably more space-efficient, which makes it more suitable for programming mobile agents. In the future, we will experiment using the “jump” approach to build robust mobile agents.
Chapter 5

Robust Scheduling and Runtime Adaptation of Multi-agent Plan Execution

Chapter 3 and Chapter 4 have presented an insight into the exception handling mechanisms in the ARTS model. Specifically, Chapter 3 has introduced the compositional structure of plans in ARTS, with a focus on the organization of user-specified exception handlers. Complementarily, Chapter 4 discussed the multi-step backtracking which serves as the default and automatic exception handling mechanism in ARTS.

Besides exception handling, exception prevention is also an important approach to improving the robustness and reliability of the system. Therefore, this chapter, as well as Chapter 6, will address this issue, with a focus on task and plan scheduling.

Robust scheduling plays a crucial role for the robustness of multi-agent systems, in which agent autonomy makes execution environments dynamic and non-deterministic. A good schedule can avoid improper task/agent assignments; such improper assignments may lead to exceptions in the subsequent execution, and require great effort to clean up.

In this chapter, we introduce a model to incorporate the concept of trust which indicates the probability that an agent will comply with its commitments to scheduling, thus improving the predictability and stability of the schedule. To deal with exceptions during execution, we adapt and
evolve the schedule at runtime by interleaving the processes of evaluation, scheduling, execution, and monitoring, in the life cycle of a plan. Experiments show that schedules maximizing participants’ trust are more likely to survive and succeed in open and dynamic environments. The results also prove that the proposed plan evaluation approach conforms to the results of simulation, thus being helpful for plan selection.

5.1 Introduction

Agents need to cooperate to achieve business goals, because individuals usually possess limited capabilities and resources. To produce correct and reliable results, execution plans that constrain and define the allowed business scenarios among agents act as a blueprint to guide and organize the autonomous participants.

A plan is commonly represented as a task network. For each task in a plan, there are usually several eligible agents to accomplish its requirements. A scheduler which assigns each task in the selected plan to an appropriate agent plays an important role in this framework. Much research has been carried out in scheduling, and one can assume that good schedules can be found using heuristic-based methods, as in general it is an intractable (NP-Hard) problem.

Current research on scheduling mainly concerns the trade-off between the execution cost and time of a task under the assumption that all participants are reliable and trustworthy [184]. However, in multi-agent systems, autonomous agents may no longer be 100% trustworthy. They may behave dishonestly and sometimes maliciously, especially when pursuing their own interest and benefit; this is the price of autonomy. The untrustworthy environment brings uncertainty which the scheduler has to deal with. For example, a company may lie about its capability, such as processing time and cost, to attract attention and contracts. Although a customer may avoid loss when dealing with a certain agent via a carefully designed legal contract, the failure may still ruin other parts of the plan and result in an unexpected completion time and/or explosion in cost [7].

Since trust provides a method to measure and minimise the uncertainty associated with interactions in an open distributed system [129], we propose to incorporate it as a third dimension into the scheduling process, in addition to cost and time, therefore improving the reliability and robustness of the schedule, in terms of securing the participation of agents that are more likely to
fulfil their commitments.

**Example 5.1.** Tom has been promised an iPod from his parents for his 18th birthday. He decides to buy it at eBay, in order to get a bargain price.

*He finds there are more than a hundred sellers with various prices for the same model of iPod. There is a rule of thumb that the sellers with higher trust values tend to have higher prices. Most sellers also provide the options of express (in 3 days) and normal delivery (in 10 days).*

*Being a rational shopper, instead of merely choosing cheapest price, Tom decides to choose the product from sellers with a high reputation, to avoid online fraud. It will cost more, but there is a higher probability that he can receive the item as advertised.*

*Because of his limited budget, he has to consider the factors of seller price and reputation, as well as delivery time and cost, to ensure he can get the gift before his birthday. That is, if the birthday is more than 10 days later, normal delivery can be chosen. Then he can choose a better seller and save money. Otherwise, he needs to take more risk on the choice of a seller.*

As discussed in the example, simply considering the cost and time advertised by an agent is no longer meaningful for the scheduler in a multi-agent environment where the advertisement of execution capacity is in doubt. Trust provides some degree of assurance about the advertisement, and therefore should be integrated into the scheduling process.

Moreover, a goal can usually be achieved by several alternative plans in multi-agent systems. The system needs to evaluate each eligible plan and then select the most suitable one (based on certain criteria) for execution. Usually, plan selection can be achieved by comparing the best possible schedule for each eligible plan. However, this approach becomes impractical in systems consisting of autonomous agents, since frequent changes in the environment will force repeated rescheduling. Consequently, although the cost of scheduling a plan once can usually be neglected with respect to the overall duration of the plan, it is both time and space consuming to explore and schedule all feasible plans repeatedly, thus degrading system performance. Therefore, we introduce a new evaluation mechanism prior to scheduling to discover the plan which is likely to result in the most robust schedule, thus avoiding (re-)scheduling on all alternative plans.

To avoid the need for rescheduling when an exception occurs, we adapt and evolve the existing
schedule at runtime by interleaving the processes of scheduling, execution, and monitoring, in the life cycle of a plan. A set of event-condition-action rules is applied to guide and manage the interaction between the schedule and the environment.

The remainder of this chapter is organized as follows. Section 5.2 gives the formal definition for the optimization criteria, and Section 5.3 describes details of incorporating trust into the life cycle of a plan. The design of experiment and results are presented in Section 5.4. Section 5.5 provides an overview of the state of the art. Finally, in Section 5.6, we present our summary and conclusions.

5.2 Trust Model and Problem Definition

In this chapter, a special Coordinator agent, which corresponds to a composite task in the ARTS model, is used to manage and execute the plan. It is in charge of plan selection, task scheduling, runtime monitoring, and schedule updating. The Coordinator is distinguished from other agents in that it is the only agent with a scheduling capability. An important point is that the Coordinator should not be regarded as a centralized controller because other agents (service providers) are autonomous and can decide whether to accept requests or not.

5.2.1 Trust-aware Platform for Agents

The concept and background on trust has been introduced in Section 1.3.4. Many models and mechanisms are proposed to model the trust value of an agent [78]. For the purpose of this thesis, a centralized trust management model like eBay [17] is assumed, to simplify the discussion. In this case, the trust value of each agent, ranged from 0 to 100, can be obtained by querying the trust manager, which models the trust value based on the feedback it receives from the service users. It should be noted that distributed trust models can also be applied in our model, to provide the trust value of an agent.
5.2 Trust Model and Problem Definition

5.2.2 Problem Definition

A plan, which specifies a task network, can be modelled as a directed acyclic graph (DAG) in which each node represents a task and each arch is a precedent relation between two tasks. The scheduling of the plan involves discovering resources or services and delegating tasks to suitable agents to meet users’ requirements and constraints. To simplify the discussion, and without loss generality, we assume that each agent is only designed for a specific task.

To achieve the goal of the system, the Coordinator is responsible for goal decomposition, plan selection, plan scheduling, and plan monitoring.

Definition 5.1 (Coordinator). The Coordinator can be represented as a tuple $\mathbf{CO} = < g, A, I, R >$.

- $g$ is the goal of the system, which can be decomposed into a set of plans, denoted as $\mathcal{P}$. Each plan $p \in \mathcal{P}$ is capable of achieving $g$. Consequently, $\mathcal{T}_p$, the set containing all tasks that appear in plan $p$, can be derived.
- $A$ is the set of service agents in the system. $A$ can be divided into subsets $\mathcal{A}_i$ whose elements are capable of achieving task $t_i$. Let $a_{ij} \in \mathcal{A}_i$ be the $j$th agent capable of achieving $t_i$.
- $I$ is the solution to achieve $g$. It contains two parts: a selected plan $p \in \mathcal{P}$; and a schedule $\omega_p$ which behaves as a mapping function: $t_i \rightarrow a_{ij}$ for each $t_i \in \mathcal{T}_p$.
- $R$ is the set of event-condition-action (ECA) rules to adapt the schedule to changes in the environment.

The ECA rules in $R$, which will be further discussed in the next section, are applied to enable the Coordinator to operate in a dynamic environment.

Since the Coordinator is only concerned with the outcome of an agent executing $t_i$, but not its internal implementation details, each $a_{ij}$ can be abstractly modelled as a tuple representing its processing capabilities, including time, cost, and trust.

Definition 5.2 (Agent Execution Capability). Agent $a_{ij}$ is a triplet $< \text{time}_{ij}, \text{cost}_{ij}, \text{trust}_{ij} >$, which means $a_{ij}$ can be trusted with $\text{trust}_{ij}$ to complete $t_i$ for the price of $\text{cost}_{ij}$ within time $\text{time}_{ij}$.
**Definition 5.3** (Task Execution Time). If a task $t_i$ is assigned to an agent $a_{ij}$, then the execution time required for $t_i$ equals to that of $a_{ij}$, denoted as $\text{time}(i) = \text{time}_{ij}$.

**Definition 5.4** (Task Execution Cost). If a task $t_i$ is assigned to an agent $a_{ij}$, then the execution cost required for $t_i$ equals to that of $a_{ij}$, denoted as $\text{cost}(i) = \text{cost}_{ij}$.

**Definition 5.5** (Task Trust). If a task $t_i$ is assigned to an agent $a_{ij}$, then the trust of $t_i$ equals to that of $a_{ij}$, denoted as $\text{trust}(i) = \text{trust}_{ij}$.

The aim of the Coordinator is to generate a solution $I$, which contains a selected plan $p$ from $\mathcal{P}$ and an assignment of every $t_i \in T_p$ onto a suitable $a_{ij}$, to achieve the multi-objective optimization criteria below. $\mathcal{CP}_p$ is used to denote the critical path [82] of the plan, which is the path with the longest overall duration, and $T_{p,\text{CP}}$ is the set containing all tasks on the critical path. $B$ is the cost constraint (budget) and $D$ is the time constraint (deadline) required by the user as part of the goal.

\[
\text{Cost}(I) = \sum_{t_i \in T_p} \text{cost}(i) \quad (5.1)
\]
\[
\text{Time}(I) = \sum_{t_i \in T_{p,\text{CP}}} \text{time}(i) \quad (5.2)
\]
\[
P(\text{Success}(I)) = \prod_{t_i \in T_p} \text{trust}(i) \quad (5.3)
\]

The objective is to maximize $P(\text{Success}(I))$ subject to:

\[
\text{Cost}(I) < B \quad (5.4)
\]
\[
\text{Time}(I) < D \quad (5.5)
\]

The cost of the solution $\text{Cost}(I)$, which is the summation of the cost of every task in the selected plan, should not exceed the user’s budget, while the total duration of the solution
Time($I$), which equals to the total execution time of tasks on the critical path, should be within the deadline.

$P(\text{Success}(I))$ indicates the probability that the solution will finally succeed. It is a product of trust($i$), which indicates the probability that each task will succeed, because the failure of any task will cause the whole schedule to fail. We assume that each agent’s trust or probability of success is independent. A robust schedule has to maximize the solution’s probability to succeed.

### 5.3 Trust-based Plan Management

In most practical environments, scheduling is an ongoing, reactive process, because of evolving and changing circumstances [143]. Therefore, Robustness is considered to be an important measurement for a good schedule, and has different definitions [3]. In this research, it is defined and measured as the probability that a schedule will complete successfully.

**Definition 5.6 (Schedule Robustness).** The robustness of a schedule $\omega$ indicates its effectiveness and liveness during the execution. It is measured by the probability that $\omega$ will complete successfully, and is denoted as $P(\omega) \in [0, 1]$.

The traditional solution for a changing environment is to completely regenerate a new schedule under the new settings, or to repair the previous schedule in some way. However, since scheduling of interdependent tasks in distributed, heterogeneous computing environments is well known to be an NP-hard problem [152], repeated rescheduling and reconsideration in a frequently changing environment becomes impractical, due to its cost and duration.

We propose to incorporate agent trust into plan management, thus improving the robustness and predictability of the system. Higher robustness indicates the schedule is more likely to remain valid in the dynamic, changing, and non-deterministic environment, thus reducing the possibility of subsequent rescheduling and making the outcome more predictable. The inspiration comes from the fact that in economic and social activities, the player with higher trust value is more likely to comply with its promises.

Figure 5.1 shows the behaviour of the Coordinator, which consists of three interleaved activities: plan evaluation, scheduling, and execution monitoring. The plan evaluation activity
Figure 5.1: Execution flow of Coordinator

is in charge of evaluating each eligible plan and selecting the most promising one from them for further scheduling. During the execution of the plan, the monitor will check its progress and environment changes, to decide whether rescheduling the current plan or selecting a new plan is necessary.

5.3.1 Plan Evaluation

To select the most suitable plan, some scheduling methods rely on scheduling every eligible plan, and then selecting the optimal one. However, since scheduling with cost and time constraints is NP-hard, this approach becomes too costly to deploy in practice. Other methods try scheduling plans one by one until a satisfactory result is found. However, this usually leads to a poor schedule.

We devise a heuristic plan evaluation method to select the most preferable plan, thus reducing the efforts needed by the scheduler. The core idea is to trade spare time and cost of tasks within the constraints to gain additional trust, thus checking the highest probability that a plan can succeed. The computation is based on the median values of cost, time and trust of each task, because
observation shows that median values are more stable and robust in depicting the sampled feature of the system [70].

Our approach first breaks down the goal into a set of plans, each of which is capable of achieving the coordinator’s goal. Then a utility function is applied to each plan to calculate its probability of succeeding within the budget and deadline constraints. Finally, the plan with the highest utility value is selected as solution $I$.

The decomposition of the goal may result in different plan sets if plans are allowed to contain selective composition. For example, a plan containing two alternative paths can also be treated as two separate plans. Therefore, to simplify the algorithms introduced in this chapter, we assume that the final form of a plan contains no selective composition. Consequently, each plan is equal to a directed acyclic graph in which every branch needs to be executed. For more information on goal decomposition, please refer to Chapter 3.

**Theorem 5.1.** A goal which is composed by sequential, selective, and parallel composition operators can be finally decomposed into a set of plans which contains no selective operator.

**Proof:** Algorithm 5.1 illustrates the process of decomposing a goal into a set of plans without the selective operator. Assume the goal has $n$ selective operators, each of which contains $m$ alternatives. After an iteration of the while loop, a plan $p \in \mathcal{P}^T$ will be replaced by $m$ plans, each of which contains one fewer selective operators than $p$. Therefore, the total number of iterations required to remove all selective operators is

$$
\text{iterations} = n + \sum_{i=1}^{n} (m^i \prod_{i=1}^{n-i} (n-i))
$$

Since both $n$ and $m$ are finite numbers, iterations is also finite. Therefore, the function $\text{goalToNonSelectivePlans}()$ will terminate and return a set of plans containing no selective operators.

**Theorem 5.2.** Let $\mathcal{P}_1$ and $\mathcal{P}_2$ both be the plan sets from the goal $g$ without the selective operator; then $\mathcal{P}_1 = \mathcal{P}_2$.

**Proof:** Assume $\mathcal{P}_1 \neq \mathcal{P}_2$, therefore, $\exists p (p \in \mathcal{P}_1 \land p \notin \mathcal{P}_2)$. Because $p \in \mathcal{P}_1$, there should be a path in $g$ matching $p$. However, because $p \notin \mathcal{P}_2$, no such path should be in $g$. The two inferences
Algorithm 5.1: goalToNonSelectivePlans()

Require: the goal $g$
Ensure: $\mathcal{P}$, a set of plans each of which can achieve $g$ and contains no selective composition

1. initialize a temporal plan set $\mathcal{P}^T = \{g\}$;
2. $\mathcal{P} = \emptyset$;
3. while $\mathcal{P}^T \neq \emptyset$ do
   4. get $p$ from $\mathcal{P}^T$;
   5. get a selective operator $so$ from $p$;
   6. if $so = \text{null}$ then
      7. move $p$ from $\mathcal{P}^T$ to $\mathcal{P}$;
   8. else
      9. decompose $p$ into $p_1, p_2, \ldots, p_n$, each contains an alternative option at $so$;
     10. replace $p$ with $p_1, p_2, \ldots, p_n$ in $\mathcal{P}^T$;
   11. end
4. end
13. return $\mathcal{P}$;

contradict with each other. Therefore, $\mathcal{P}_1 = \mathcal{P}_2$.

For a task $t_i$ in plan $p$, $\widetilde{\text{cost}}(i)$, the median cost for achieving $t_i$, and $\text{MAD}(\text{cost}(i))$, the median absolute deviation, can be calculated as follows. $\text{MAD}(\text{cost}(i))$ is the median of $\text{cost}_{ij}$’s absolute deviations from the median of $\text{cost}(i)$, where $\text{cost}_{ij}$ is assumed to be sorted such that $\text{cost}_{i1} \leq \text{cost}_{i2} \leq \cdots \leq \text{cost}_{im}$, and $m$ is the number of agents in $\mathcal{A}_i$.

$$\widetilde{\text{cost}}(i) = \begin{cases} \frac{1}{2} \left( \text{cost}_{i\frac{m}{2}} + \text{cost}_{i\frac{m}{2}+1} \right) & \text{if } m \text{ mod } 2 = 0 \\ \text{cost}_{i\frac{m}{2}} & \text{otherwise} \end{cases}$$ \hspace{1cm} (5.6)

$$\text{MAD}(\text{cost}(i)) = \text{median}(|\text{cost}_{ij} - \widetilde{\text{cost}}(i)|)$$ \hspace{1cm} (5.7)

In the same way, $\widetilde{\text{time}}(i)$, $\text{MAD}(\text{time}(i))$, $\widetilde{\text{trust}}(i)$ and $\text{MAD}(\text{trust}(i))$ can be calculated. Therefore, the median execution cost and time of $p$ are estimated as follows.

$$\widetilde{\text{Cost}}(p) = \sum_{t_i \in T_p} \text{cost}(i)$$ \hspace{1cm} (5.8)
\[ \text{Time}(p) = \sum_{t \in T^p} \text{time}(i) \] (5.9)

An estimation function \( \eta(i) \) for task \( t_i \) is introduced to predict the best possible trust that can be obtained by \( t_i \) within the constraints of budget \( B \) and deadline \( D \). Specifically, \( \eta(i) \) tries to trade the spare time and cost of \( t_i \) for additional trust.

\[ \text{timePrice}(i) = \frac{\text{MAD}(\text{time}(i))}{\text{MAD}(\text{cost}(i))} \] (5.10)

\[ \text{trustPrice}(i) = \frac{\text{MAD}(\text{trust}(i))}{\text{MAD}(\text{cost}(i))} \] (5.11)

\[ \text{trustFromSpareCost}(i) = \left( \frac{B \cdot \tilde{\text{cost}}(i)}{\text{Cost}(p)} - \tilde{\text{cost}}(i) \right) \cdot \text{trustPrice}(i) \] (5.12)

\[ \text{trustFromSpareTime}(i) = \left( \frac{D \cdot \left( \text{time}(i) + \text{allocatedSlack}(i) \right)}{\text{Time}(p)} - \text{time}(i) \right) \cdot \frac{\text{trustPrice}(i)}{\text{timePrice}(i)} \] (5.13)

\[ \eta(i) = \text{trust}(i) + \text{trustFromSpareCost}(i) + \text{trustFromSpareTime}(i) \] (5.14)

The value \( \text{timePrice}(i) \) is the unit price of task execution time for deviating from \( \tilde{\text{time}}(i) \), while \( \text{trustPrice}(i) \) represents the unit price of the task trust deviating from \( \tilde{\text{trust}}(i) \). As defined in the function \( \text{trustFromSpareCost}(i) \), the allowed cost for \( t_i \) can be obtained by distributing \( B \) to each \( t_i \) according to its share in the total cost of the plan. Then the marginal cost for \( t_i \) is derived by deducting \( \tilde{\text{cost}}(i) \) from the allowed cost. Finally, the cost margin is converted into the measure of trust according to the unit price of changing \( t_i \)'s trust.

The trade of \( t_i \)'s time for trust follows the same way, except that the allocated slack time for \( t_i \), denoted as \( \text{allocatedSlack}(i) \), should also be considered as part of its allowed time. The value of \( \text{allocatedSlack}(i) \) is derived from \( t_i \)'s slack time [82], denoted as \( \text{slack}(i) \), which represents the maximum amount of time that \( t_i \) is allowed to be delayed without affecting the whole schedule, and is computed while searching for the critical path. Algorithm 5.2 illustrates the procedures to compute the critical path, as well as the slack time.

However, \( \text{slack}(i) \) is in fact the total allowed delay for a non-critical path and is usually shared
Algorithm 5.2: findCriticalPath(p)

Require: p, the plan to be evaluated. It should contain no selective composition. \(time(i)\),
the execution time of task \(t_i \in p\), comes with \(p\).

Ensure: \(T_p^{\text{CP}}\), the set containing all tasks on the critical path of \(p\)

1. initialize \(T_p^{\text{CP}} = \emptyset\);
2. foreach task \(t_i \in p\) do
   /* initialize the array of earliest finish time for each task */
3. \hspace{1em} EFT\[t_i]\] = 0;
4. end
5. foreach task \(t_i \in p\) do
   foreach child task \(t_j\) of \(t_i\) do
7. \hspace{1em} EFT\[t_j]\] = max(EFT\[t_j]\], EFT\[t_i]\] + \(time(j)\);
8. end
9. end
10. totalTime = 0;
11. foreach EFT\[t_i]\] do
12. \hspace{1em} totalTime = max(totalTime, EFT\[t_i]\]);
13. end
14. foreach task \(t_i \in p\) do
   /* initialize the array of latest finish time for each task */
15. \hspace{1em} LFT\[t_i]\] = totalTime;
16. end
17. foreach task \(t_i \in p\) do
   foreach parent task \(t_j\) of \(t_i\) do
19. \hspace{1em} LFT\[t_j]\] = min(LFT\[t_j]\], EFT\[t_i]\] - \(time(i)\);
20. end
21. end
22. foreach task \(t_i \in p\) do
   /* get the slack time for each task */
23. \hspace{1em} slack\[t_i]\] = LFT\[t_i]\] - EFT\[t_i]\];
24. /* tasks on the critical path have the slack time of 0 */
25. if slack\[t_i]\] is 0 then
26. \hspace{1em} put \(t_i\) into \(T_p^{\text{CP}}\);
27. end
28. end
29. return \(T_p^{\text{CP}}\);

by several continuous tasks. Therefore, we introduce \(\text{allocatedSlack}(i)\), which is the allocated delay for \(t_i\) on a per-task basis, to record the result of splitting and distributing \(\text{slack}(i)\) among
involved tasks.

**Definition 5.7** (Allocated Slack Time). A *period of allocated slack time for a task* \( t_i \), denoted as \( \text{allocatedSlack}(i) \), is a fraction of \( t_i \)’s slack time \( \text{slack}(i) \), and \( \text{allocatedSlack}(i) \in [0, \text{slack}(i)] \). \( t_i \) is allowed to be delayed for a maximum duration of \( \text{allocatedSlack}(i) \).

The slack time of a task on the critical path is 0. Therefore, the allocated slack time should also be 0.

**Example 5.2.** Figure 5.2 illustrates the computation of the slack time for tasks in a plan. The earliest finish time of each task, denoted as \( \text{EFT} \), is obtained by a forward traversal of the graph, while the latest finish time \( \text{LFT} \) is obtained by a backward traversal. Thereafter, the slack time of each task (\( \text{slack}(i) \)) can be found from as \( \text{LFT} - \text{EFT} \). If \( \text{slack}(i) = 0 \), then \( t_i \) is on the critical path.

In the example, both \( t_2 \) and \( t_3 \) have the same slack time. That is, \( \text{slack}(2) = \text{slack}(3) = 6 \), which means either \( t_2 \) or \( t_3 \), but not both of them, can be at most delayed 6 minutes. Implicitly, they need to share the slack time. For instance, if \( t_2 \) is delayed by 2 minutes, then the start time of \( t_3 \) is also delayed by 2 minutes. Consequently, the allowed delay for \( t_3 \) becomes \( 6 - 2 = 4 \) minutes.

Allocated slack time is a distribution of the shared slack time amongst the involved tasks. Therefore, \( \text{allocatedSlack}(2) + \text{allocatedSlack}(3) = 6 \). Specifically, the slack time allocation algorithm in this thesis partitions the slack time in proportion to task duration. In this example, duration of \( t_2 \) to that of \( t_3 \) is 2:3. Therefore, \( \text{allocatedSlack}(2) \) should take \( 6 \times \frac{2}{5} = 2.4 \) minutes, and \( \text{allocatedSlack}(3) = 3.6 \) minutes.

The following special cases are not represented in the formula, but should be considered during the computation.

- If \( \text{MAD}(\text{time}(i)) = \text{MAD}(\text{cost}(i)) = 0 \), then the agent with the highest trust should be selected, since all agents have the same execution time and cost.
- If only \( \text{MAD}(\text{time}(i)) = 0 \), then \( \text{trustFromSpareTime}(i) = 0 \). In this case, although a task still has spare time, it cannot benefit from it, since all agents have the same execution time.
• If only $MAD(cost(i)) = 0$, then $trustFromSpareCost(i) = 0$, because all agents have the same execution cost.

• If $MAD(trust(i)) = 0$, then $trustFromSpareTime(i) = trustFromSpareCost(i) = 0$. In this case, there is no need to consider the conversion of spare time or cost, because all agents have the same trust.

• If $\eta(i) > 1$, then set $\eta(i) = 1$; or if $\eta(i) < 0$, then $\eta(i) = 0$. Since the result of $\eta(i)$ is measured in terms of trust, which is the probability that $t_i$ will be fulfilled, its value should be restricted to $[0, 1]$.

Finally, the utility function $\eta(p)$, which heuristically represents the highest probability that plan $p$ will succeed, is defined by multiplying the highest possible trust of each participating task. It comes from the fact that the failure of any task will cause the whole plan to fail.

\[
\eta(p) = \prod_{t \in T_p} \eta(i) \tag{5.15}
\]

$\eta(p)$ stands for the highest probability that every task of $p$ will succeed within the constraints of budget, deadline, and the actual execution environment. Therefore, a plan with higher $\eta(p)$ should be preferred since it is statistically more likely to succeed in an uncertain environment. It should be noted that the utility value $\eta(i)$ of each participating task $t_i$ is a statistical estimation by
5.3 Trust-based Plan Management

Algorithm 5.3: findMostPromisingPlan(\(\mathcal{P}\))

Require: \(\mathcal{P}\), a set plans which contains no selective composition
Ensure: a plan \(p\) which has the highest probability to succeed within the constraints

1. \(p = null;\)
2. \(\eta(p) = 0;\)
3. foreach \(p_i \in \mathcal{P}\) do
   4. compute \(\eta(p_i);\)
   5. if \(\eta(p_i) > \eta(p)\) then
      6. \(p = p_i;\)
      7. \(\eta(p) = \eta(p_i);\)
   8. end
4. end
10. return \(p;\)

evenly distributing budget and deadline into it, which is different from the precise distribution of resources into each task in plan scheduling and execution.

The process of plan evaluation and selection is summarized in Algorithm 5.3. Given a set of eligible plans, the utility function is applied to each of them, and the one with the highest value should be selected for scheduling and execution.

5.3.2 Plan Scheduling

After plan \(p\) is selected in the plan evaluation stage, the Coordinator refines the solution \(I\) to a multi-objective scheduling problem. To find an optimized task assignment for \(p\) that maximizes the possibility that the system will eventually succeed, we adopt a genetic algorithm (GA) [37] to design and implement the scheduling process.

GAs are a stochastic heuristic search method which is based upon simplifications of evolutionary processes observed in nature [8, 63]. Operating on more than one solution at once, they perform well at both the exploration and exploitation of large search spaces [162] to find high-quality solutions in polynomial time [91, 108, 185]. As they are a general-purpose but powerful and efficient scheduling mechanism, we propose to implement the scheduling component of our model within the framework of GAs.

Typically, a GA contains two main problem-dependent components: the encoding schema and the evaluation function. The encoding scheme breaks a potential solution into discrete parts, which
can vary independently. In GA, the discrete parts are called “genes” and the solution is called a chromosome. Since a plan can be modelled as a DAG, its corresponding chromosome can be naturally encoded as a vector of task-agent assignments, as shown in Figure 5.3. Correspondingly, $t_i$ is the $i$th gene of the chromosome, and each $a_{ij} \in \mathcal{A}_i$ is an allele of the gene. Task dependencies are not encoded into the chromosome. Instead, they are modelled into the calculation of the execution time and are dealt with by the evaluation function.

The evaluation function, denoted as $\Upsilon$, measures the quality of a particular solution according to the given optimization objectives. $\Upsilon$ contains two parts: the fitness function, denoted as $F(I)$, to encourage the higher robustness of the schedule, and the penalty function, denoted as $P(I)$, to enforce the constraints of budget $B$ and deadline $D$. In our definition, the higher the value of $\Upsilon$, the better the solution.

\begin{align*}
F_{cost}(I) &= 1 - \frac{Cost(I)}{B} \\
F_{time}(I) &= 1 - \frac{Time(I)}{D} \\
F(I) &= P(\text{Success}(I)) \cdot (F_{cost}(I) + F_{time}(I)) \\
P_{budget}(I) &= \begin{cases} 
F_{cost}(I) & \text{if } Cost(I) >= B \\
0 & \text{otherwise}
\end{cases} \\
P_{deadline}(I) &= \begin{cases} 
F_{time}(I) & \text{if } Time(I) >= D \\
0 & \text{otherwise}
\end{cases} \\
P(I) &= P_{budget}(I) + P_{deadline}(I) \\
\Upsilon &= F(I) + P(I)
\end{align*}

In $F(I)$, the multiplication of $P(\text{Success}(I))$ and $F_{cost}(I) + F_{time}(I)$ indicates that the
saving on execution time or cost is only valuable if the schedule succeeds. \( P(I) \) is designed to enforce the bias against ineligible schedules that exceed the budget or go beyond the deadline.

After the chromosome and the evaluation function are defined, standard GA processes, including selection, crossover, and mutation, can be applied to find the optimized schedule.

- **Selection.** The selection or reproduction process removes the poor solutions (as indicated by their low evaluation values) and replaces them by the duplicates of the best solutions. Elitism, the property of guaranteeing that the best solution remains in the population, is usually applied.

- **Crossover.** Crossovers are used to create new solutions by random swapping two portions of two arbitrarily selected solutions in the current population. The idea behind the crossover is that the fittest solution may result from the combination of two of the current fittest solutions.

- **Mutation.** Mutation is defined as randomly changing the agent assignment of a randomly selected task in an arbitrary chromosome to some other agent. Mutations occasionally occur, in order to allow a child to obtain features that are not possessed by either of its parents. It leads the search out of a local optimum, in the hope of getting an even better solution.

As summarized in Algorithm 5.4, these processes are repeated for a certain number of times or generations, and then the solution with the best evaluation value is chosen as the schedule. However, none of the evolutionary algorithms can guarantee that the resulting schedule is feasible (satisfying all the constraints) [162]. In practice, a feasible schedule can always be found if the constraints are relaxed. Since the purpose of this chapter is achieving the robustness and reliability of the schedule by integrating agent trust, experiments are not performed on extremely constrained cases; therefore, all solutions are feasible.

### 5.3.3 Execution Monitoring

Since the environment keeps on evolving, for instance because agents join and leave the system at will, it is a crucial requirement for the Coordinator to monitor the progress of schedule execution and to respond reactively in a timely fashion. To achieve this purpose, a set of ECA rules is devised for the Coordinator to manage and adapt the schedule. The rules, as shown in Table 5.1,
Algorithm 5.4: scheduleWithGA(p)

Require: p, the plan to be scheduled. It should contain no selective composition

Ensure: ω, a schedule of task assignments with the probability of success optimized within the constraints

/* n is a constant, representing the size of population */
1 pop = randomly generate a population of n chromosomes;

/* generations is a constant, representing the total number of generations to be evolved */
2 while generations > 0 do
3    perform the evaluation function Y on each member of pop;
4    repeat
5        /* chromosomes with higher utility have higher probability to be selected */
6           select two parent chromosomes from pop according to their fitness;
7           cross over the selected parents to form a new offspring;
8           mutate the new offspring at random locus;
9        until the size of pop' is n;
10       pop = pop';
11       generations = generations − 1;
12 end
13 ω = the member with the maximum Y in pop;
14 return ω;

are represented in the form of event ← conditions | actions, and can easily be translated into an existing agent programming language, such as 3APL or JADE [15]. To simplify the notation, the delegatee of ti in the schedule I is denoted as ai0, and tc is used to denote the current task being monitored by the Coordinator. The meanings of the functions are explained in the rest of the section.

The meaning of the rules can be better comprehended when combined with Figure 5.1. During the schedule execution, the Coordinator continuously monitors whether an exception has occurred;

<table>
<thead>
<tr>
<th>Function</th>
<th>Conditions/Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail(a0)</td>
<td>canRetry(c)</td>
</tr>
<tr>
<td>fail(a0)</td>
<td>cannotRetry(c)</td>
</tr>
<tr>
<td>unavailable(a0)</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>isSelectionPoint(tc) ∧ outOfControl</td>
</tr>
</tbody>
</table>
if so, the exception is then treated as an event to trigger further actions. There are two main types of exception the Coordinator needs to consider:

- \textit{fail}(a_{c0})$, representing that the execution of $a_{c0}$ failed, and $t_c$ has not been completed.
- \textit{unavailable}(a_{c0})$, representing that $a_{c0}$ is not available to start the execution of $t_c$. It also includes the situation that $a_{c0}$ breaks its promise about any of execution cost, time, or trust.

Dealing with task failure can be very complicated. If the failure leaves the system in an inconsistent state, it must be recovered to restore the consistency before any further actions can be taken. Even though the terminating state is consistent, some applications require the side effects of the failed execution to be cleaned up. The general approach is to generate a semantic compensation plan to “undo” the failure, and then restart the execution cycle shown in Figure 5.1, to reevaluate and reschedule the newly available execution graph. However, this issue is outside the scope of this chapter. For further details, we refer to Chapter 3 and Chapter 4.

If the failure can be resolved by reassigning the task to another agent for completion, the Coordinator can modify the schedule by finding a substitute agent, and then continue execution. Agent unavailability can also be dealt with by substitution. Therefore, the utility function $\zeta(j)$ is introduced to measure the soundness of using $A_{ij}$ as a substitute for achieving $t_i$.

$$
\zeta(j) = \text{trust}_{ij} \cdot \left( \left( 1 - \frac{\text{cost}_{ij} - \text{cost}_{i0}}{\text{costMargin}(I)} \right) + \left( 1 - \frac{\text{time}_{ij} - \text{time}_{i0}}{\text{timeMargin}(I) + \text{slack}(i)} \right) \right)
$$

(5.23)

The assessment is based on the cost and time margin between the schedule and system constraints. The value $\text{costMargin}(I)$ represents the surplus between budget $B$ and the schedule $I$, and $\text{timeMargin}(I)$ is the surplus of time from deadline $D$. The value of $\text{slack}(i)$, the slack time of $t_i$, is added, because tasks not on the critical path have an allowed delay without affecting the whole schedule. These three values need to be updated whenever there is a change to the existing schedule.

The function $\text{findSubstitution}(t_c)$ applies the utility function to find the most promising substitute agent $a_{cj}$ for $a_{c0}$ to complete $t_c$. Details are given in Algorithm 5.5.

In the algorithm, after $\zeta(j)$ is computed for each $a_{ij} \in A_i$, they are sorted in descending order. The Coordinator then iterates through and selects the first agent that does not violate the system
Algorithm 5.5: findSubstitution($I$, $t_i$)

Require: Schedule $I$, task $t_i$ which requires a substitute agent

Ensure: $a_{i0}$, the new selected agent for $t_i$

1. $a_{i0} = \text{null}$;
2. $\text{costMargin}(I) = B - \text{Cost}(I)$;
3. $\text{timeMargin}(I) = D - \text{Time}(I)$;
4. get $\text{slack}(i)$;
5. foreach $a_{ij} \in A_i$ do
6.     compute $\zeta(j)$;
7. end
8. sort $\zeta(j)$ in descending order;
9. foreach $\zeta(j)$ do
10.    if ($\text{cost}_{ij} - \text{cost}_{i0} < \text{costMargin}(I) \land (\text{time}_{ij} - \text{time}_{i0} < \text{timeMargin}(I) + \text{slack}(i))$
11.       then
12.           $a_{i0} = a_{ij}$;
13.           break;
14.       end
15. end
16. if $a_{i0} = \text{null}$ then
17.     $a_{i0} = a_{ij}$ with highest $\zeta(j)$;
18.     outOfControl = true;
19. end
20. $I = I^0$ which use $a_{i0}$ as a substitution;
21. return $a_{i0}$;

budget and deadline constraints. If none is found in the agent set, then the use of any agent as a substitution will breach system constraints. In this case, the Coordinator selects the agent with the highest utility, and marks the schedule as outOfControl, which means a plan reevaluation is necessary when arriving at the next point with more than one eligible subsequent plan. Finally, the solution $I$ is updated by applying the selected $a_{i0}$ to substitute the original agent for the execution of $t_i$. Consequently, $\text{costMargin}(I)$, $\text{timeMargin}(I)$ and $\text{slack}(i)$ are all updated accordingly.

In case of agent failure, an extra step update($I$) needs to be performed before the function findSubstitution(), since the failed agent has already consumed time and money. In this case, update($I$) first inserts a dummy task at the place of failure, and assigns the exact amount of consumed time and money to it. Then it updates $I$ to synchronize with all the changes.
The Coordinator can also provide support to control the computational cost of scheduling itself. It is generally assumed that the scheduling cost is comparatively low and can be ignored. However, repeated rescheduling, especially when performed at runtime, has a direct impact on the system performance. The solution is to define a threshold for the accumulated scheduling cost for the coordinator. If this value is exceeded, schedule \( I \), including \( \text{costMargin}(I) \) and \( \text{timeMargin}(I) \), should be updated to account for the coordinator’s cost.

It is possible that during the execution, a better service agent for \( t_i \) may appear. In this case, the Coordinator may simply assign \( t_i \) to it without considering the existing delegation relationship. However, this may reduce the Coordinator’s own reputation for breaching of contract.

### 5.4 Experiments and Evaluations

In this section, experiments on plan scheduling and plan evaluation are performed. To the best of our knowledge, applying trust into agent scheduling and execution management, to improve the predictability and stability of the schedule, has not been addressed in previous research. Therefore, we can only provide evaluation and comparison between scheduling strategies that do and do not utilize trust.

#### 5.4.1 Experiment design

The environment for the experiments was created by adopting the DAG dataset of the Resource-Constrained Project Scheduling Problem (RCPSP), provided in the Project Scheduling Problem Library (PSPLIB) [126]. The standard data sets j30, j60, and j90 consist of 480 DAGs with 30, 60 and 90 non-dummy activities, respectively. These graph instances are generated by the generator ProGen [126].

To transform the test sets into our experiments on trust-based scheduling, we have simulated a set of tasks with various cost, time, and trust levels, which are represented by mean values. In simulation, the cost level of each task is randomly selected between 100 and 2000, the time level is between 100 and 5000, and the trust level is set to 1.0. Each node in the graph is then randomly linked with a task.

Each task is supported by a random number (between 1 and 10) of agent providers with
Table 5.2: Parameter settings for the genetic algorithm

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>population size</td>
<td>the number of chromosomes in each generation</td>
<td>50</td>
</tr>
<tr>
<td>total generations</td>
<td>the iterations that the GA needs to evolve</td>
<td>1000</td>
</tr>
<tr>
<td>selection range</td>
<td>selection is made from top ranging percentages of the population</td>
<td>0.90</td>
</tr>
<tr>
<td>$P(crossover)$</td>
<td>the probability that a crossover occurs in the population</td>
<td>0.35</td>
</tr>
<tr>
<td>$P(mutation)$</td>
<td>the probability that each gene of an offspring mutates</td>
<td>0.08</td>
</tr>
</tbody>
</table>

varied processing capabilities. The exact cost, time, and trust value for $a_{ij}$ are obtained by randomly fluctuating about 20% around the cost, time, and trust level of $t_i$. To make the simulated environment more practical, the fluctuation follows the observation that in commerce, the execution time of an agent is typically inversely proportional to its cost, while its reputation is in direct proportion to the cost.

To simulate different levels of user constraints, the deadline $D$ and the budget $B$ are evaluated at a wide range of possible values between the maximum and minimum allowed time and cost for the plan. The values $k_1$ and $k_2$ in the formulas, ranging from 0 to 1, indicate the constraint levels of $D$ and $B$ respectively, the higher the value, the tighter the constraints.

$$B = maxCost - k_1(maxCost - minCost) \quad (5.24)$$
$$D = maxTime - k_2(maxTime - minTime) \quad (5.25)$$

In the experiments, $maxCost$ and $minCost$ are obtained by always selecting the service agent with the most or least processing cost for each task respectively, while $maxTime$ and $minTime$ are obtained by selecting the slowest/fastest agent for each task.

The implementation of the GA is based on JGAP (Java Genetic Algorithms Package) [104], which is an open source framework for genetic algorithms and genetic programming in Java. It provides basic genetic mechanisms and can be easily extended to solve more complicated problems.

The genetic parameters used in the experiments are listed in Table 5.2. The population for each generation is set to 50 and the total generation is 1000 for each experiment. All other parameters, such as crossover and mutation probability, are the system defaults.
5.4 Experiments and Evaluations

5.4.2 Result of plan scheduling

The experiment is carried out by setting the deadline constraint level $k_2$ at 0.2, 0.5, 0.8, and 1.0, to represent relaxed, medium, tight, and extreme constraints on $D$ respectively, and then increasing the budget constraint level $k_1$ from 0 to 1 by 0.1.

The simulation is performed on around 20% of the DAG definitions in the PSPLIB library. Altering the experiments by changing resource settings and DAG definitions yields similar results with respect to the correlation between $B$, $D$, and the robustness of the system. Therefore, the results of a randomly selected DAG with 32 nodes (defined by j3016_10.sm in the library) are illustrated in Figure 5.4 as a representative example. In Figure 5.4, NoTrust represents scheduling without considering agent trust, while MaxTrust tries to maximize participants’ trust value within the constraints.

The horizontal line at the top represents the ideal result of the most robust schedule, which is obtained by merely selecting the most trustworthy agent for each task without considering the constraints of deadline or budget. By contrast, the horizontal line at the bottom is the least robust
schedule, obtained by always assigning tasks to agents with the lowest trust value.

In general, the robustness of schedules, either MaxTrust or NoTrust, will drop as the constraint levels become tighter, since the scheduler needs first to consider cost and time constraints. In order to meet the deadline or budget, the scheduler sometimes has to choose less trustworthy agents for their cheaper price or shorter processing time. On the contrary, schedulers have better robustness when the constraint levels are low because the scheduler has abundant budget and time to exchange for services provided by more trustable agents.

The robustness of NoTrust schedules remains stable but low. It is only a little better than those using least trustworthy agents. By contrast, MaxTrust schedules result in more reliable schedules. When the constraint levels are extremely high, the scheduling result of both methods are approximately the same. However, as the constraints of budget and deadline become looser, MaxTrust scheduling outperforms NoTrust, and the difference between them becomes more outstanding. When the constraint levels approach 0, the result of MaxTrust scheduling approaches to the ideal result of the most robust schedule.

As shown in Figure 5.4(d), the result of NoTrust scheduling might be a little better than that of MaxTrust for extreme constraints (both $k_1$ and $k_2$ are greater than 0.9) because of the use of genetic algorithms. When constraint levels are extremely high, the proportion of chromosomes which fail to meet the constraints in the population is also high. This situation gets even worse in MaxTrust scheduling since its utility function devotes value to not only the budget and deadline constraints, but also trust. Therefore, it has higher probability to overadjust the population to use agents with low trust values. However, the variance only appears in extreme conditions, and it is small enough to be ignored.

### 5.4.3 Result of plan evaluation

Ten testing plans with the same task number are randomly selected from the DAG dataset, and each node in the graph is linked with a random selected task. These plans share the same budget and deadline, which are derived from another randomly selected plan. For each testing plan, the estimated and simulated outcome having the best success probability within the constraints are computed by plan evaluation (Equation 5.15) and plan scheduling (Algorithm 5.4) respectively.

Figure 5.5 represents the results achieved on 32 node DAGs under different constraint levels.
Since tasks differ from plan to plan, the resource requirements of each plan will also differ. As a result, the estimated outcome, as well as the simulated outcome, of each plan differs. Therefore, the simulation is concerned with difference between the estimated and simulated outcomes for each plan, but not with the comparison between plans.

In general, the estimation made by plan evaluation conforms to the simulated outcome computed by plan scheduling. Therefore, the proposed method for plan evaluation can be used as guidance to help find the most robust plan.

The estimation usually gives higher values than the simulated outcomes because the conversion of spare time and budget into trust value optimistically assumes that the estimated trust value will be provided by some agents. However, this assumption is usually not accurate and leads to the overestimation of plan robustness. By contrast, the scheduling process in the simulation is based on the actual service agents available in the system.

In the results, the estimation is closer to the simulated outcome in moderate-to-low constraint levels than in high levels. Referring to the definition of the estimation function $\eta(i)$ in...
Section 5.3.1, the conversion from spare cost to trust is computed as follow:

\[
trustFromSpareCost(i) = \text{cost}(i) \cdot \left( \frac{B}{\text{Cost}(p)} - 1 \right) \cdot trustPrice(i)
\]

As the distance between \( B \) and \( \text{Cost}(p) \) increases, the estimation relies more on the cost and trust distribution among agents. Similarly, the estimation also relies more on the time and trust distribution when \( D \) grows bigger or smaller than \( \text{Time}(p) \). However, it is the distribution of cost, time, and trust that builds up the implicit relationship between plan evaluation and scheduling. Therefore, the difference between the result of evaluation and scheduling is correlated to the value of constraint levels.

Moreover, the set of agents selectable by a scheduler depends highly on the constraint levels. With the increase of \( B \) and \( D \) in the system, the scheduler has more options during scheduling, thus producing results closer to those of the evaluation.

Although plan evaluation and scheduling produce similar outcomes, they are designed for different purposes. Plan evaluation can efficiently predict the best success probability of a plan; however, it does not provide the specific assignments of tasks to service agents. In contrast, plan scheduling can find out how to assign each task to a specific agent to achieve the best success probability. The two need to be combined together in multi-agent systems to provide an efficient and accurate solution for plan and task management.

### 5.5 Related Work

When scheduling is applied to environments composed of autonomous agents, the robustness of the schedule, which reflects the probability that the schedule will finally succeed, becomes a crucial requirement. Many researchers have applied redundancy-based approaches to deal with uncertainty in the environment [3, 36]. By contrast, we apply agent trust to improve the schedule robustness and reliability, thus reducing the likelihood of rescheduling.

Trust and reputation have been widely recognized as a crucial part of online applications and autonomous systems [78, 129], such as peer-to-peer networks, e-commerce, and online auctions, since they encourage individuals to behave as promised. [188] applies a Markov model to detect
fraudsters from amongst the participants, based on the reputation system. However, there is no existing work in systematically applying trust into plan management and plan scheduling in multi-agent environments.

The runtime execution monitoring of our Coordinator is similar to the concept of reactive scheduling [43, 143], which revises or reoptimizes the schedule when an unexpected event occurs in dynamic environments. However, our use of trust in plan scheduling and schedule updating can reduce the frequency of rescheduling and reoptimization.

### 5.6 Discussion and Conclusion

In this chapter, we incorporate agent trust to manage the life cycle of an agent plan, thus improving the robustness and predictability of the overall system. The approach can be classified into three main scenarios: plan evaluation, scheduling, and monitoring. During the evaluation process, the system goal is first decomposed into a set of plans, each of which is capable of achieving the goal. Then each plan is evaluated to find the highest probability it can succeed, by applying the median values of cost, time, and trust. Finally, the plan with the highest value is selected for scheduling.

The scheduling stage applies a genetic algorithm to find the optimized assignment of each task in the plan to an autonomous service agent, thereby maximizing the robustness of the schedule while meeting the constraints of deadline and budget. During schedule execution, the Coordinator monitors for any exceptions that occur. If one is detected, the Coordinator will actively respond by following the exception handling rules, which may result in task reassignment, plan rescheduling, or even reevaluation.

The experiments show that integrating trust into plan management can greatly improve system robustness and reliability, especially when the constraint level is not very tight, meaning that the resulting schedule is more likely to succeed. The results also verify that the proposed plan evaluation method conforms to the scheduling result, thus being helpful for plan selection.
Chapter 6

Reasoning Intra-Dependency in Commitments for System Robustness and Performance

Chapter 5 has discussed the integration of agent trust into the scheduling processes, to improve the probability that plans will be executed successfully as scheduled. Focusing on resource allocation, the approach relies on the Coordinator, a central management agent, to be the brain of the system. However, in order to maximize the benefits of autonomous and distributed execution in multi-agent systems, individual agents should also participate in the management of the system, performing runtime reasoning, and adaptation of the interaction and coordination among them. Addressing this issue, this chapter studies and enhances the individual’s ability to reason about and to manage its interaction relationship with others at runtime, based on commitments among them.

Commitment-modelled protocols enable flexible and robust interactions among agents. However, existing work has focused on features and capabilities of protocols without considering the active role of agents in them. Therefore, in this chapter, we propose to augment agents with the ability to reason about and manipulate their commitments, to maximize the system utility. We adopt a bottom-up approach by first investigating the intra-dependency between each commitment’s pre- and post- conditions, which leads to a novel classification of commitments as well as a formalism to express various types of complex commitment. Within this framework, we
provide a set of inference rules to benefit an agent by means of commitment refactoring, which enables composition and/or decomposition of its commitments to optimize runtime performance. We also discuss the pros and cons of an agent scheduling and executing its commitments in parallel. We propose a reasoning strategy and an algorithm to minimize possible loss when commitments are broken, and maximize the overall system robustness and performance. Experiments show that concurrent schedules based on the features of commitments can boost system performance significantly.

6.1 Introduction

Agents are autonomous and social entities which cooperate and compete with each other to achieve some goals beyond the limitation of each individual’s ability. To ensure the coherent and correct execution of distributed agents, the accurate representation of interaction protocols, which define the permissible sequence of message exchanges, among them is a crucial part of multi-agent system design and analysis. However, to specify every possible message sequence in multi-agent systems is impractical, even if it is possible, because of the dynamic and non-deterministic features of the execution environment. Consequently, a protocol should be flexible, to support and benefit from agent autonomy, and to tolerate possible exceptions. [181] argues that the interactions among agents should only constrain their actions to the extent necessary to carry out the given protocol, and no more.

To overcome the inflexibility of traditional approaches that focus on the permissible sequences of the messages in protocols, [39, 173] propose to design agent interactions with the high-level concept of social commitments [30, 75, 140], which represent responsibility from one agent to another. These approaches characterize the meanings of the messages between the participants in terms of commitments to provide flexible interaction protocols to participating agents. Specifically, different sequences of actions can be derived from the initial state to one of the acceptable termination states with all the constraints imposed by commitments satisfied. If the multi-agent system follows any one of these sequences, the system is considered to be consistent and correct.

However, the issue of how an agent can reason and manipulate its commitments, to control and
benefit from the provided flexibility, has not been addressed. In fact, commitments add an extra dimension of obligation and predictability to agents’ behaviours. Therefore, agents can perceive and reason with each other in a more predictive way by generating an optimized but reliable and robust execution plan that can be adjusted at the runtime. In certain situations, agents can convert some sequential execution schedules to be concurrent for better performance, while retaining the same correctness and consistency criteria.

To address the effects of commitments on an agent’s deliberation process, in this chapter, we first analyze and classify their inter- and intra-dependency relationships, and introduce a formalism to represent and manipulate these relationships. We then define a concurrent scheduling model based on the formalism, to enable agents to minimize their potential loss when they pursue parallel execution. The main contributions of the chapter are the following:

- A novel classification of commitments according to their intra-dependency between the pre- and post- conditions.
- A formalism based on the classification for expressing complex commitments.
- The concept and rules of commitment refactoring, to help agents compose and decompose their commitments for better runtime performance.
- A concurrent commitment-based scheduling strategy for agents, for minimizing possible loss when commitments are broken and thus maximizing the overall system performance.

The feature of commitment recoverability in case of breaches is also incorporated into the reasoning and scheduling processes of agents, thus helping to build concurrent and robust multi-agent systems.

The remainder of the chapter is organized as follows. Section 6.2 introduces the background about commitment protocols, along with a motivating example. Section 6.3 gives the formal definition for complex commitments with various intra-dependency types. Section 6.4 introduces commitment refactoring, reasoning, and manipulation strategies for agents to refine their runtime execution schedules. The design of experiments and results are presented in Section 6.5. Section 6.6 provides an overview of the state of the art. Finally, in Section 6.7, we present our conclusions.
6.2 Background and Motivation

Commitments among agents provide an abstraction to represent their obligatory or contractual relationship. They model the “content” of the interactions among agents, instead of the lower-level operational details of how messages are exchanged [183]. Therefore, commitment-based protocols enable flexible coordination among agents, with the specification of what to achieve instead of how to. By specifying the states that need to be reached in terms of commitments, the protocols allow various sequences of execution towards a state, thus becoming more flexible than traditional formalisms, such as finite state machines and Petri nets, which rely on exact message sequences [101, 159, 182].

Following [181], a conditional commitment $CC(x, y, \Phi, \Psi)$ denotes an obligation of a debtor $x$ to a creditor $y$ to bring about $\Psi$ if $\Phi$ holds. Here $\Phi$ is the precondition and $\Psi$ is the post-condition of the commitment. If $\Phi$ is true, it is considered as unconditional and can be denoted as $C(x, y, \Psi) \equiv CC(x, y, true, \Psi)$. After $\Psi$ becomes true, the commitment is said to be discharged.

The following Netbill protocol example, adapted from [101], shows a scenario represented in commitments.

Example 6.1. As shown in Figure 6.1, the interaction begins ($s_1$) with a customer requesting a quote for some desired goods ($s_2$), followed by the merchant sending the quote ($s_3$). If the customer accepts the quote ($s_4$), then the merchant delivers the goods ($s_5$) and waits for an electronic payment order (EPO). The goods delivered at this point are encrypted, that is, not usable. After receiving the EPO ($s_6$), the merchant sends the receipt to the customer ($s_7$), who can then successfully decrypt and use the goods.

As an addition to the original example, we append an attribute of processing time, represented as [number of hours], for completing each message-related activity. In this way, we can evaluate and compare the efficiency of different schedules. The execution time of the example is computed in a similar way to the estimation generally used in workflow graphs. It will take a total of 77 hours to accomplish the schedule. Table 6.1 lists the commitments within each state. For simplicity, the merchant is denoted as $m$ and the customer as $c$ in the table.

A wide range of interaction sequences can be generated and supported from the above
### 6.2 Background and Motivation

Table 6.1: Commitments in each state of Interaction Protocol

<table>
<thead>
<tr>
<th>State</th>
<th>Commitment-related Beliefs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>Null</td>
</tr>
<tr>
<td>$s_2$</td>
<td>request</td>
</tr>
</tbody>
</table>
| $s_3$ | $CC(m, c, \text{accept, goods}) \land CC(m, c, \text{payment, receipt})$  
  where accept = $CC(c, m, \text{goods, payment})$ |
| $s_4$ | $CC(c, m, \text{goods, payment}) \land C(m, c, \text{goods}) \land CC(m, c, \text{payment, receipt})$ |
| $s_5$ | $\text{goods} \land C(c, m, \text{payment}) \land CC(m, c, \text{payment, receipt})$ |
| $s_6$ | $\text{goods} \land \text{payment} \land C(c, m, \text{receipt})$ |
| $s_7$ | $\text{goods} \land \text{payment} \land \text{receipt}$ |

commitment protocol by combining various messages in different order when it is allowed [173]. However, existing work does not provide sufficient support for reasoning about the temporal relationship between the precondition $\Phi$ and post-condition $\Psi$ of a commitment $CC(x, y, \Phi, \Psi)$. Consequently, although the commitment protocol allows different interleaving of the execution of $\Phi$ and $\Psi$, player $x$ and $y$ have to sequentialize the execution of $\Phi$ before $\Psi$ to guarantee the completion of the commitment. Otherwise, agent $x$ may face the high risk of loss, in the case that $\Phi$ will not be satisfied thereafter.

**Example 6.2.** If the merchant in $CC(m, c, \text{accept, goods})$ commences to deliver the goods before the customer agrees with the payment, the merchant risks not receiving any money because the goods delivery completes and discharges the commitment and the customer has no further obligation. Thus, the merchant will tend to choose a defensive approach to wait until they receive the acceptance.
The lack of support for commitment reasoning greatly reduces the flexibility provided by the commitment concept. When every participant chooses to sequentialize their execution, the resulting state machine may even be condensed into one rigid sequence of messages, similar to the traditional methods. Moreover, the sequential execution of all commitments brings inefficiency and reduces system performance.

Addressing these drawbacks, we study and provide a classification for commitments according to the temporal relationship between their pre- and post-conditions. With this knowledge, we incorporate the ability of commitment reasoning into the agent’s deliberation process, to enable rational selection of a beneficial but robust and reliable execution path at run time. We will apply our model to the same example to demonstrate that an agent can actively reason about and manipulate its commitments to improve the system’s robustness, as well as performance.

6.3 Intra-Commitment Dependency

This chapter's main focus is on reasoning about and manipulating the dependency and temporal relationships between agents. We borrow the formalism of the commitment protocol from [39] to represent the commitment and dependency among agents. We also use many temporal logic notions directly to specify the semantics of our extensions to commitments. We note that all notions in the chapter can be translated into temporal logic [102] and applied to the general issues of task dependencies. However, we believe the commitment protocol is more natural and less prone to programming errors.

6.3.1 Commitment Classification

The pre- and post-conditions of a commitment have different types of temporal dependencies. Because the precondition is usually carried out by some other agents, the agent can reason about its partners and adjust its behaviour accordingly, and thus benefit from the knowledge of dependency. We formally define the classification of intra-dependency in commitments on top of the interaction protocol, which provides a semantic and execution framework for agents and their commitments.
6.3 Intra-Commitment Dependency

An interaction protocol defines a global state machine incorporating the allowed messages and their occurrence order. The individual agent represents its view of such a protocol as a transition system.

**Definition 6.1 (Agent Interaction Protocol).** An agent interaction protocol is a tuple \( P = (S, s_0, F, M, \Gamma) \) consisting of the following elements:

- \( S = \{s_0, s_1, \ldots, s_n\} \) is a finite set of interaction states in chronological order. If \( i < j \), then \( s_i \) happens before \( s_j \).
- \( s_0 \in S \) is the initial state.
- \( F \subseteq S \) is a set of final states.
- \( M = \{m_0, m_1, \ldots, m_n\} \) is a finite set of messages.
- \( \Gamma \) is a finite set of transitions (\( \Gamma \subseteq S \times M \times S \)). Each transition \( \tau = (s_i, m, s_{i+1}) \in \Gamma \) identifies a source state \( s_i \), a target state \( s_{i+1} \), and a message \( m \) that is either consumed or produced during this transition.

A computation, or path, of the interaction protocol is defined to be an infinite sequence of states, denoted as \( \sigma = < s_0, s_1, \ldots > \). The commitment types are defined by the properties of \( \sigma \).

**Definition 6.2 (Commitment Types).** A commitment, denoted as \( CC(x, y, \Phi, \Psi) \), can be classified according to the temporal constraints between its conditional part \( \Phi \) and result part \( \Psi \). The creditor and debtor of a commitment are omitted when they are clear.

- The commitment is **ordered** if \( \Phi \) has to be achieved before or at the same time as \( \Psi \) is satisfied; formally, \( \sigma \) satisfies \( \forall i \leq n \land s_i \models \Phi \land s_n \models \Psi \). The set of ordered commitments, denoted as \( OC \), can be further categorized:
  - **strictly ordered**, denoted as \( CC(\Phi \cap \Psi) \), if \( \Phi \) has to be satisfied before or at the same time as \( \Psi \) and must remain true until \( \Psi \), as illustrated in Figure 6.2(a). Formally, \( \sigma \) satisfies \( \forall i \leq n \land \exists j (i \leq j \leq n \land s_j \models \Phi) \land s_n \models \Psi \). The strictly ordered commitment set is denoted as \( SOC \).
  - **weakly ordered**, denoted as \( CC(\Phi \sqcup \Psi) \), if \( \neg \Phi \) is required between the first occurrence of \( \Phi \) and the completion of the commitment, as illustrated in Figure 6.2(b). Formally, \( \sigma \) satisfies \( \exists i \leq n \land s_i \models \Phi \land s_n \models \Psi \land \exists j (i < j \leq n \land \forall k (j \leq k \leq n \land s_k \models \neg \Phi)) \),
Reasoning Intra-Dependency in Commitments for System Robustness and Performance

(a) Strictly ordered

(b) Weakly ordered

(c) Strictly unordered

(d) Weakly unordered

Figure 6.2: Temporal relationship for the pre- and post- conditions of a commitment

\[ n \wedge s_n \models \neg (\Phi) \]. The set of weakly ordered commitments is denoted as WOC, where 
\[ WOC = OC - SOC. \]

- The commitment is unordered if the occurrence order of \( \Phi \) and \( \Psi \) does not affect the execution or completion of the commitment. The set of unordered commitments is denoted as UC, which can be further categorized into
  - strictly unordered, denoted as \( CC(\Phi \parallel \Psi) \), if its success requires both \( \Phi \) and \( \Psi \) are satisfied but in any order, as illustrated in Figure 6.2(b). Formally, \( \sigma \) satisfies 
    \[ \exists n (s_n \models \Phi \wedge s_n \models \Psi). \] The set of strictly unordered commitments is denoted as SUC.
  - weakly unordered, denoted as \( CC(\Phi\diamond\Psi) \), if regardless of whether \( \Phi \) occurs or not, \( \Phi \) is not satisfied at the time of completion, as illustrated in Figure 6.2(d). Formally, \( \sigma \) satisfies 
    \[ \exists n (s_n \models \neg \Phi \wedge s_n \models \Psi). \] The set of weakly unordered commitments is denoted as WUC, where \( WUC = UC - SUC. \)

Example 6.3. One example per commitment category is provided as follows.

\( CC(payment \cap refund) \equiv CC(m, c, payment, refund) \) is a strictly ordered commitment because merchants have to receive money from customers before they can return it.

\( CC(member \cup no\_join\_fee) \equiv CC(club, customer, member, no\_join\_fee) \) is a weakly ordered commitment which indicates that the joining fee will be waived for a past member of the club.

\( CC(payment \parallel goods) \equiv CC(m, c, payment, goods) \) can be treated as a strictly unordered commitment to process the payment and goods delivery at the same time.
6.3 Intra-Commitment Dependency

\[ CC(\text{threat\&boarding}) \equiv CC(\text{airline, passenger, has\_prohibited\_goods, boarding}) \] is a weakly unordered commitment which means that a passenger who is not carrying prohibited goods at the gate can board the plane.

For some companies, \( CC(m, c, \text{payment, receipt}) \) can be treated as a strictly unordered commitment because they trust customers and allow the payment and receipt issuance to be carried out concurrently. By contrast, others may have a policy of requiring it to be strictly ordered to reduce their risk of money loss. Therefore, the dependency relationship is application-dependent and needs to be defined by users by adding an extra attribute of commitment type to the commitment expression.

The containing relationships among different sets of commitment types are summarized in Theorem 6.1.

**Theorem 6.1.** The relations between commitment categories are

\[
\begin{align*}
\text{SOC (Strictly Ordered Commitment)} & \subset \text{SUC (Strictly Unordered Commitment)} \\
\text{WOC (Weakly Ordered Commitment)} & \subset \text{WUC (Weakly Unordered Commitment)} \\
\text{OC (Ordered Commitment)} & \subset \text{UC (Unordered Commitment)}
\end{align*}
\]

**Proof:**

- **SOC \subset SUC**

  In Definition 6.2, elements of SOC satisfy

  \[
  \exists i \exists n (i \leq n \land \forall j (i \leq j \land \phi_{ij} \land s_n \models \psi) \\
  \Leftrightarrow \exists i \exists n (i \leq n \land \forall j (i < j \land \phi_{ij} \land \phi_{ij} \land s_n \models \psi))
  \]

  \[
  \Leftrightarrow \exists i (i \leq n \land \forall j (i \leq j \land \phi_{ij} \land \phi_{ij} \land s_n \models \psi))
  \]

  By contrast, the elements of SUC satisfy

  \[
  \exists n (s_n \models \phi \land s_n \models \psi)
  \Leftrightarrow s_n \models \phi \land s_n \models \psi
  \]
Compared with $SUC$, $SOC$ is further constrained by $\exists i (i \leq n \land \forall j (i \leq j < n \land s_j \models \Phi)$.

Therefore, $SOC \subset SUC$.

- $WOC \subset WUC$

  In Definition 6.2, elements of $WOC$ satisfy

  $\exists i (i < n \land s_i \models \Phi \land s_n \models \Psi \land \exists j (i < j \leq n \land \forall k (j \leq k \leq n \land s_k \models \neg \Phi)))$

  $\iff \exists i (i < n \land s_i \models \Phi \land \exists j (i < j \leq n \land \forall k (j \leq k < n \land s_k \models \neg \Phi)) \land (s_n \models \neg \Phi \land s_n \models \Psi))$

  By contrast, the elements of $WUC$ satisfy

  $\exists n (s_n \models \neg \Phi \land s_n \models \Psi)$

  $\iff s_n \models \neg \Phi \land s_n \models \Psi$

  Compared with $WUC$, $WOC$ is further constrained by

  $\exists i (i < n \land s_i \models \Phi \land \exists j (i < j \leq n \land \forall k (j \leq k < n \land s_k \models \neg \Phi)))$. Therefore, $WOC \subset WUC$.

- $OC \subset UC$

  As in Definition 6.2, $OC = SOC \cup WOC$, $UC = SUC \cup WUC$. Because $SOC \subset SUC$ and $WOC \subset WUC$, $OC \subset UC$.

  Therefore, all three containing rules are proved.

\[\blacktriangle\]

As shown in Theorem 6.2, $WUC$ can be inferred from $SUC$. In addition, $WUC$ and $SUC$ can be represented by the composition of $WOC$ and $SOC$. However, all four categories are listed separately for the completeness of the definition.

\textbf{Theorem 6.2.}

\begin{align*}
CC(\Phi \diamond \Psi) & \equiv CC(\neg \Phi \parallel \Psi) \\
CC(\Phi \diamond \Psi) & \equiv CC(\neg \Phi \sqcap \Psi) \lor CC(\Phi \sqcup \Psi) \\
CC(\Phi \parallel \Psi) & \equiv CC(\Phi \sqcap \Psi) \lor CC(\neg \Phi \sqcup \Psi)
\end{align*}

\textit{Proof:}
6.3 Intra-Commitment Dependency

- $CC(\Phi \diamond \Psi) \equiv CC(\neg \Phi \parallel \Psi)$

  The proof of this equation can be straightforwardly derived from the definition.

  
  
  \[
  CC(\Phi \diamond \Psi) \iff \exists n(s_n \models \neg \Phi \land s_n \models \Psi) \\
  \iff CC(\neg \Phi \parallel \Psi)
  \]

- $CC(\Phi \diamond \Psi) \equiv CC(\neg \Phi \cap \Psi) \lor CC(\Phi \sqcup \Psi)$

  $CC(\Phi \diamond \Psi)$ represents that at the final state, $s_n \models \neg \Phi \land s_n \models \Psi$. There are only two possibilities that result in $s_n \models \neg \Phi$:

  1. $\Phi$ is always false. The definition of $CC(\neg \Phi \cap \Psi)$ represents this case.
  2. $\Phi$ was true first, and became false later. The definition of $CC(\Phi \sqcup \Psi)$ represents this case.

  Therefore, $CC(\Phi \diamond \Psi) \equiv CC(\neg \Phi \cap \Psi) \lor CC(\Phi \sqcup \Psi)$.

- $CC(\Phi \parallel \Psi) \equiv CC(\neg \Phi \cap \Psi) \lor CC(\neg \Phi \sqcup \Psi)$

  According to the first two equations in this theorem:

  
  
  \[
  CC(\Phi \parallel \Psi) \iff CC(\neg \Phi \diamond \Psi) \\
  \iff CC(\Phi \cap \Psi) \lor CC(\neg \Phi \sqcup \Psi)
  \]

  Therefore, all three equations are proved.

6.3.2 Syntax of Commitments

Commitment can be simple, in that it can contain only atomic propositions as pre- and post-conditions. But it is more general to have complex commitments in practice, where both precondition and post-condition can be complex logical formulas with nested commitments.

We extend the existing commitment representation to express the complexity caused by various combinations. The operators include the standard connectives, such as $\land$, $\lor$ and $\lnot$ from propositional logic, and the commitment type operators from Definition 6.2.

**Definition 6.3 (State-Formula).** Let \( \dot{p} \) be an atomic proposition defined by a state variable, $\Phi$ and
Ψ be state formulas. State-Formulas in a commitment are defined as:

1. \( \dot{p} \) is a state-formula.
2. \( \neg \Phi, \Phi \land \Psi \) or \( \Phi \lor \Psi \) is a state-formula.

\[\square\]

**Definition 6.4 (Commitment-Formula).** Let \( \Phi \) and \( \Psi \) be state-formulas, Commitment-Formulas are defined as:

1. \( CC(\Phi \cap \Psi), CC(\Phi \cup \Psi), CC(\Phi \parallel \Psi) \) or \( CC(\Phi \spadesuit \Psi) \) is a commitment-formula.
2. if \( \phi, \phi_1 \) and \( \phi_2 \) are commitment-formulas, \( \phi_1 \land \phi_2, \phi_1 \lor \phi_2 \) or \( \neg \phi \) is a commitment-formula.

\[\square\]

If we consider a commitment formula \( \phi = CC(x, y, c, r) \) as a proposition indicating whether the commitment is held, then \( \phi \) becomes a state formula and nested commitments are supported.

### Semantics of Commitments

The semantics of a commitment are defined by two satisfaction relations (both denoted by \( \models \)): one for state-formulas and one for commitment-formulas. For the state-formulas, \( \models \) is a relation between a protocol \( \mathcal{P} \), one of its states \( s \), and a state-formula \( \Phi \), denoted as \( \mathcal{P}, s \models \Phi \). For the commitment-formulas, \( \models \) is a relation between a protocol \( \mathcal{P} \), one of its paths \( \sigma \), and a commitment formula \( \phi \), denoted as \( \mathcal{P}, \sigma \models \Phi \). For convenience, we omit \( \mathcal{P} \) if it is clear from the context.

**Definition 6.5 (Commitment Semantics).** Let \( \dot{p} \) be an atomic proposition, \( \mathcal{P} = (\Pi, S, s_0, \mathcal{F}, M, \Gamma) \) be an interaction protocol, and \( s \in S, \Phi, \Psi \) be formulas. The satisfaction relation \( \models \) for state-formulas is defined as:

- \( s \models \dot{p} \) if and only if \( \dot{p} \) is true in \( s \)
- \( s \models \neg \Phi \) if and only if \( s \not\models \Phi \)
- \( s \models \Phi \land \Psi \) if and only if \( s \models \Phi \) and \( s \models \Psi \)
- \( s \models \Phi \lor \Psi \) if and only if \( s \models \Phi \) or \( s \models \Psi \)

Let \( \phi, \phi_1 \) and \( \phi_2 \) be commitment-formulas. For path \( \sigma \) the satisfaction relation \( \models \) for commitment-formulas is defined by:
• $\sigma \models (\phi \equiv CC(\Phi \cap \Psi))$ if and only if $\phi \in SOC$
• $\sigma \models (\phi \equiv CC(\Phi \cup \Psi))$ if and only if $\phi \in WOC$
• $\sigma \models (\phi \equiv CC(\Phi \parallel \Psi))$ if and only if $\phi \in SUC$
• $\sigma \models (\phi \equiv CC(\Phi \diamond \Psi))$ if and only if $\phi \in WUC$
• $\sigma \models \phi_1 \land \phi_2$ if and only if $\sigma \models \phi_1$ and $\sigma \models \phi_2$
• $\sigma \models \phi_1 \lor \phi_2$ if and only if $\sigma \models \phi_1$ or $\sigma \models \phi_2$
• $\sigma \models \neg \phi$ if and only if $\sigma \not\models \phi$

$s_0 \models \neg \Phi \land s_0 \models \neg \Psi$ is assumed, in order to simplify the discussion.

The same state-formulas combined by different commitment types may exhibit different properties. In certain conditions, complex formulas can be decomposed into smaller and simpler ones, thus helping to reduce system complexity and improve system concurrency. The following theorem shows the applicable inference rules for commitments.

**Theorem 6.3.** Let $\Phi, \Psi$ be state-formulas. The inference rules for commitments are defined as follow:

- **Distribution over strictly ordered:**
  1. $CC(\Phi \cap (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \cap \Psi_1) \land CC(\Phi \cap \Psi_2)$
  2. $CC(\Phi \cap (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \cap \Psi_1) \lor CC(\Phi \cap \Psi_2)$
  3. $CC((\Phi_1 \land \Phi_2) \cap \Psi) \Rightarrow CC(\Phi_1 \cap \Psi) \land CC(\Phi_2 \cap \Psi)$
  4. $CC((\Phi_1 \lor \Phi_2) \cap \Psi) \Leftarrow CC(\Phi_1 \cap \Psi) \lor CC(\Phi_2 \cap \Psi)$

- **Distribution over weakly ordered:**
  1. $CC(\Phi \cup (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \cup \Psi_1) \land CC(\Phi \cup \Psi_2)$
  2. $CC(\Phi \cup (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \cup \Psi_1) \lor CC(\Phi \cup \Psi_2)$
  3. $CC((\Phi_1 \land \Phi_2) \cup \Psi) \Leftarrow CC(\Phi_1 \cup \Psi) \lor CC(\Phi_2 \cup \Psi)$
  4. $CC((\Phi_1 \lor \Phi_2) \cup \Psi) \Rightarrow CC(\Phi_1 \cup \Psi) \land CC(\Phi_2 \cup \Psi)$

- **Distribution over strictly unordered:**
  1. $CC(\Phi \parallel (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \parallel \Psi_1) \land CC(\Phi \parallel \Psi_2)$
  2. $CC(\Phi \parallel (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \parallel \Psi_1) \lor CC(\Phi \parallel \Psi_2)$
  3. $CC((\Phi_1 \land \Phi_2) \parallel \Psi) \Rightarrow CC(\Phi_1 \parallel \Psi) \land CC(\Phi_2 \parallel \Psi)$
4. \( CC((\Phi_1 \lor \Phi_2) \parallel \Psi) \equiv CC(\Phi_1 \parallel \Psi) \lor CC(\Phi_2 \parallel \Psi) \)

- \textbf{Distribution over weakly unordered:}

1. \( CC(\Phi \Diamond (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \Diamond \Psi_1) \land CC(\Phi \Diamond \Psi_2) \)
2. \( CC(\Phi \Diamond (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \Diamond \Psi_1) \lor CC(\Phi \Diamond \Psi_2) \)
3. \( CC((\Phi_1 \land \Phi_2) \Diamond \Psi) \equiv CC(\Phi_1 \Diamond \Psi) \lor CC(\Phi_2 \Diamond \Psi) \)
4. \( CC((\Phi_1 \lor \Phi_2) \Diamond \Psi) \Rightarrow CC(\Phi_1 \Diamond \Psi) \land CC(\Phi_2 \Diamond \Psi) \)

\textbf{Proof:} We only present the proof of first rule of each commitment category. The complete proof can be found in Appendix A.

- \( CC(\Phi \sqcap (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \sqcap \Psi_1) \land CC(\Phi \sqcap \Psi_2) \)

From Definition 6.5, the state-formulas \( s_n \models (\Psi_1 \land \Psi_2) \equiv s_n \models \Psi_1 \land s_n \models \Psi_2 \). Therefore, the commitment-formulas

\[
CC(\Phi \sqcap (\Psi_1 \land \Psi_2))
\]

\[
\Leftrightarrow \exists i \exists n (i \leq n \land \forall j (i \leq j \leq n \land s_j \models \Phi) \land (s_n \models \Psi_1 \land s_n \models \Psi_2))
\]

\[
\Rightarrow \exists i \exists n (i \leq n \land \forall j (i \leq j \leq n \land s_j \models \Phi) \land s_n \models \Psi_1)
\]

\[
\land \exists i \exists n (i \leq n \land \forall j (i \leq j \leq n \land s_j \models \Phi) \land s_n \models \Psi_2)
\]

\[
\Leftrightarrow CC(\Phi \sqcap \Psi_1) \land CC(\Phi \sqcap \Psi_2)
\]

- \( CC(\Phi \uplus (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \uplus \Psi_1) \land CC(\Phi \uplus \Psi_2) \)

\[
CC(\Phi \uplus (\Psi_1 \land \Psi_2))
\]

\[
\Leftrightarrow \exists i \exists n (i < n \land s_i \models \Phi \land (s_n \models \Psi_1 \land s_n \models \Psi_2)
\]

\[
\land \exists j (i < j \leq n \land \forall k (j \leq k \leq n \land s_k \models \neg \Phi)))
\]

\[
\Rightarrow \exists i \exists n (i < n \land s_i \models \Phi \land \exists s_n \models \Psi_1 \land \exists j (i < j \leq n \land \forall k (j \leq k \leq n \land s_k \models \neg \Phi)))
\]

\[
\land \exists \exists n (i < n \land s_i \models \Phi \land s_n \models \Psi_2 \land \exists j (i < j \leq n \land \forall k (j \leq k \leq n \land s_k \models \neg \Phi)))
\]

\[
\Leftrightarrow CC(\Phi \uplus \Psi_1) \land CC(\Phi \uplus \Psi_2)
\]
6.4 Reasoning and Manipulation of Commitments

- \( CC(\Phi \parallel (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \parallel \Psi_1) \land CC(\Phi \parallel \Psi_2) \)

\[
CC(\Phi \parallel (\Psi_1 \land \Psi_2)) \\
\Leftrightarrow \exists n(s_n \not\models \Phi \land s_n \models \Psi_1 \land s_n \not\models \Psi_2) \\
\Rightarrow \exists n(s_n \not\models \Phi \land s_n \models \Psi_1) \land \exists n(s_n \not\models \Phi \land s_n \not\models \Psi_2) \\
\Leftrightarrow CC(\Phi \parallel \Psi_1) \land CC(\Phi \parallel \Psi_2)
\]

- \( CC(\Phi \diamond (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \diamond \Psi_1) \land CC(\Phi \diamond \Psi_2) \)

In Theorem 6.2, \( CC(\Phi \diamond \Psi) \equiv CC(\lnot \Phi \parallel \Psi) \)

\[
CC(\Phi \diamond (\Psi_1 \land \Psi_2)) \\
\Leftrightarrow CC(\lnot \Phi \parallel (\Psi_1 \land \Psi_2)) \\
\Rightarrow CC(\lnot \Phi \parallel \Psi_1) \land CC(\lnot \Phi \parallel \Psi_2) \\
\Leftrightarrow CC(\Phi \diamond \Psi_1) \land CC(\Phi \diamond \Psi_2)
\]

The proof for other rules has the similar procedures as shown above.

6.4 Reasoning and Manipulation of Commitments

The knowledge of the compositional structure of a commitment, as well as its dependency type, helps agents to reason about its run-time properties. Therefore, agents can decide to compose, decompose, or run in parallel their commitments to find an optimal execution schedule.

6.4.1 Commitment Refactoring

An agent usually has more than one simultaneous commitment. It can reorganize them for its own benefits as long as it retains the same obligations to others. In fact, other agents cannot even notice any difference from the reorganization, since they only perceive the contractual interface of commitments, but not their internal implementation details. For example, merchants can decompose their commitment \( CC(m, c, \text{payment}, (\text{goods} \land \text{receipt})) \) into two
parallel commitments, $CC(m, c, payment, goods)$ and $CC(m, c, payment, receipt)$, or vice versa. However, from the customer’s point of view, there is no change to their contract as long as both the goods and receipt are delivered.

**Definition 6.6 (Commitment Refactoring).** Commitment refactoring modifies the commitments of an agent to improve its performance and simplify its structure without changing its external behaviour.

There are two main approaches to accomplishing commitment refactoring:

- *composing*, which merges several similar commitments into a complex one.
- *decomposing*, which splits one complex commitment into several smaller independent ones.

Refactoring by composing helps agents to aggregate several related requests together and process them in a batch. This approach benefits agents in batch processing, such as through the reduction of the cost related to the allocation of resources and the switch of execution context. It also helps to reduce the expense of thread interactions, including locking and unlocking operations. In contrast, refactoring by decomposing helps agents to disassemble a batch processing of commitments into several independent processes, which can then be executed in parallel, thus improving system efficiency and performance.

An agent carries out its commitment refactoring with the inference rules provided in Theorem 6.3. For equivalent rules $\phi_1 \equiv \phi_2$, either side of the commitment formula can be substituted by the other one. However, for rules in the form of $\phi_1 \Rightarrow \phi_2$, the commitment $\phi_1$ is used to substitute $\phi_2$ because $\phi_1$ has more restrictive constraints on permissible executions. For example, the agent with a commitment $CC(\Phi_1 \lor \Phi_2 \land \Psi)$ can decide to execute either $CC(\Phi_1 \land \Psi)$ or $CC(\Phi_2 \land \Psi)$ by inference from Rule 4 of the strictly ordered commitments.

### 6.4.2 Robust Schedule of Single Commitments

Commitment machines can support a wide range of interaction sequences [173]. However, existing research on the commitment protocol has only focused on finding all possible execution sequences, without considering the effect of preconditions on the post-condition. In fact, ordered commitments require sequential satisfaction of their conditional and result part to enact the
business logic. Although unordered commitments, especially strictly unordered ones, put no constraints on allowed orders, an agent may still choose to retain an order to minimize its possible loss.

If $x$ in $CC(x, y, \Phi \parallel \Psi)$ commences to bring about the result of $\Psi$ before the success of $\Phi$, $x$ may later encounter a loss because satisfaction of $\Psi$ completes and discharges the commitment and $y$ has no further obligation to satisfy $\Phi$. Thus, $x$ will tend to choose a defensive approach to wait until $\Phi$ becomes true. Consequently, the flexibility provided by the commitment concept is greatly reduced.

Not only the flexibility of the system, but also the possible performance boost, is reduced if agents cannot benefit from carrying out the execution of the conditional and result parts of a commitment concurrently. For example, during the execution of the Netbill protocol, the customer will reach a state holding $CC(c, m, goods \parallel payment)$ in which it faces two choices to continue its execution:

1. waiting until receiving the goods from the merchant before payment, or
2. making the payment directly while still waiting for the goods.

The first choice will guarantee that the system always terminates in a proper state in which both goods and payment are true, but will result in a longer processing time (the summation of both the goods delivery time and the payment time). Conversely, the second choice can reduce the execution time greatly (to the maximum value between the goods delivery time and payment time) but leave the customer in risk of losing money, because the merchant may choose not to deliver the goods.

We propose to incorporate the ability of commitment reasoning into the agent’s deliberation, leading to a new scheduling approach which keeps a balance between the two extremes. Before choosing to execute a commitment sequentially or concurrently, the agent first needs to evaluate the risk of following the concurrent schedule, considering the probability that the precondition will finally be achieved, and the recovery cost in case of an invalid precondition. This work focuses on strictly unordered commitments, because they allow the parallel execution of their precondition and post-condition.

As the first step, we introduce the concept of the commitment safety level for the debtor, to measure the risk and recoverability of achieving the post-condition before the precondition of a
commitment.

**Definition 6.7** (Guarded Commitment). A commitment \( \phi \equiv CC(x, y, \Phi, \Psi) \) in state \( s \) has the property of Guardianship if \( x \) perceives that there exists an unconditional commitment from any agent \( A \) to fulfill the precondition \( \Phi \). Formally, \( s \models \phi \land s \models (C(A, x, \Phi)) \).

\[\square\]

**Definition 6.8** (Recoverable Commitment). A commitment \( \phi \equiv CC(x, y, \Phi, \Psi) \) in state \( s_i \) has the property of Recoverability if \( x \) perceives that there exists a compensating commitment \( \rho \equiv CC(y, x, \neg\Phi, \Psi') \) in case of commitment breach during the execution which terminates in a state \( s_{n'} \models \Psi' \). Formally, \( \forall j (i < j < n \land s_j \models \rho) \).

\[\square\]

**Definition 6.9** (Commitment Safety Level). The Commitment Safety Level controls the robustness and reliability of a running commitment. According to the properties of Guardianship and Recoverability, each commitment \( \phi \equiv CC(x, y, \Phi, \Psi) \) contained in state \( s \) can be classified into four different levels as depicted in Table 6.2:

0 : Bare if neither Guardianship nor Compensability is true.

1 : Guarded if only Guardianship is true.

2 : Recoverable if only Compensability is true.

3 : Guaranteed if both properties are true.

**Table 6.2: Properties of commitment safety levels**

<table>
<thead>
<tr>
<th>Property</th>
<th>Level</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guardianship</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Recoverability</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
Example 6.4. In state $s_4$ of the NetBill example, the commitment $CC(c, m, \text{goods} \parallel \text{payment})$ has the feature of Guardianship because of the existence of an unconditional commitment $C(m, c, \text{goods})$ for goods delivery. Therefore, it is a Guarded commitment.

Moreover, if the customer has a method to claim back the payment from the merchant who fails to deliver the goods, the commitment $CC(c, m, \text{goods} \parallel \text{payment})$ has the feature of Recoverability. Therefore, it is a Recoverable commitment.

Since $CC(c, m, \text{goods} \parallel \text{payment})$ has the features both of Guardianship and of Recoverability, it becomes a Guaranteed commitment.

The commitment safety level is especially useful in stating the degree of system consistency in case of commitment breach if the agent decides to execute both the conditional and the result part of a strictly-unordered commitment concurrently. The higher the level of the commitment, the less possible loss in case of exceptions for the agent which schedules its execution in parallel.

Based on the safety level of the commitment $CC(x, y, \Phi \parallel \Psi)$, the debtor will reason and plan its next step accordingly:

- If it is bare, it would better choose not to act, otherwise it faces great possible losses
- If it is guarded, it may start to achieve $\Psi$ even though the conditional part has not been satisfied. This decision largely depends on the trustworthiness [129] of the agent who promises to realize $\Phi$.
- If it is recoverable, it can start to achieve $\Psi$ immediately, as long as the benefit of efficiency outperforms the cost of recovery process.
- If it is guaranteed, it should decide to achieve $\Psi$ in parallel with the agent $A$ for $\Phi$, unless efficiency is not an issue of the system.

If the debtor decides to run the commitment which is guarded or guaranteed in parallel mode, it needs to manipulate its beliefs as shown in Algorithm 6.1. The functions $create()$ and $discharge()$ represent creation and completion of a commitment respectively and $cancel()$ indicates a cancelation [101], while $add()$ appends a fact into the beliefbase of the agent. For interaction, “m?” is receiving and “m!” is sending a message. The rules in the algorithm are in the form $head? | guard \rightarrow body$ which means that when receiving the message $head$, if $guard$ is true, then $body$ will be executed.
 Algorithm 6.1: Debtor’s commitment manipulation for concurrent execution

/* x is the debtor agent, y is the creditor agent, and A is an arbitrary agent */
1 require $s_0 \models (\phi \equiv CC(x, y, \Phi, \Psi)), s_0 \models \neg\Phi, s_0 \models \neg\Psi, s_0 \models (\psi \equiv C(A, x, \Phi))$

/* To ensure $\Psi$ is achieved */
2 while $\neg\Psi$
3    /* receiving a message confirming the satisfaction of $\Phi$ */
4    if $\Phi$ then
5        discharge($\phi$);
6    else
7        /* track progress of $\Phi$ with an additional commitment */
8            create($\theta \equiv CC(x, y, \neg\Phi \land \neg\psi \land \neg\Psi, \neg\Psi)$, $\theta$);
9        while $\neg\Phi$
10           /* canceling $\psi$ means no $\Phi$ available, start recovery */
11              cancel($\psi$)? | $\neg\Phi \rightarrow cancel(\phi)$!, $\neg\Psi$!, discharge($\theta$), exit;
12              $\Phi$? | true $\rightarrow add(\Phi)$, discharge($\psi$), discharge($\theta$);
13        end
14    end
15 end

If $\theta$ is triggered, the agent needs to achieve a compensating commitment for $\neg\Psi$. For guaranteed commitment, it is already defined as $\rho$. In other cases, the agent needs to either negotiate with others or seek help from human operators to find such a way.

In the Netbill example, state $s_4$ contains a guarded commitment $CC(c, m, goods, payment)$. If the customer trusts the merchant’s promise of delivering goods, the customer makes the payment directly without waiting for the goods. As a result, the overall interaction time can be reduced from 77 hours in Figure 6.1 to 53 hours in Figure 6.3.

As shown in the example, our model provides a flexible approach to generating concurrent schedules reactively, according to the runtime situation, by reasoning about the trustworthiness of and promises held by agents to each other. Therefore, even if the program is specified sequentially as in Figure 6.1, it may be executed in parallel by the participating agents as in Figure 6.3. If only recoverable commitments are allowed to be executed in parallel, the concurrent execution can be mapped back into the sequential one for exception recovery.
6.4 Reasoning and Manipulation of Commitments

6.4.3 Robust Schedule of Unordered Commitments Sequence

In the interaction protocol of a multi-agent system, the pre-condition of one commitment may depend on the success of another commitment, thus forming the dependency relationship among them.

**Definition 6.10 (Commitment Dependency).** Let $\phi_1 = CC(\Phi_1, \Psi_1)$ and $\phi_2 = CC(\Phi_2, \Psi_2)$. If $\Phi_2 = \Psi_1$, then $\phi_2$ depends on the successful completion of $\phi_1$, denoted as $\phi_1 \triangleright \phi_2$.

---

**Definition 6.11 (Commitment Sequence).** Let $\phi_1, \phi_2, \ldots, \phi_n$ be a group of commitments. If they satisfy the dependency relationship $\phi_1 \triangleright \phi_2 \triangleright \ldots \triangleright \phi_n$, then they form a sequence of commitments. Correspondingly, the tasks achieving their pre- and post-conditions have the same sequential relationship.

---

In general, given a sequence of commitments $\phi_1 \triangleright \phi_2 \triangleright \ldots \triangleright \phi_n$, the execution of commitments, as well as their corresponding tasks, must follow the order specified by the sequence, that is, $\phi_i$ must precede $\phi_{i+1}$.

**Example 6.5.** In the NetBill example, the commitment $\phi_3 = CC(m, c, payment, receipt)$ depends on the completion of $\phi_2 = CC(c, m, goods, payment)$, which in turn depends on $\phi_1 = CC(m, c, accept, goods)$. They form a sequence of $\phi_1 \triangleright \phi_2 \triangleright \phi_3$. In general, tasks must be executed in the prescribed order to satisfy the steps of accept, goods, payment, and receipt one after the...
For a sequence of tasks specified by the commitment dependency, any error will cause the whole sequence to fail. Therefore, we introduce Theorem 6.4 to compute the cost of error for the sequence, which helps developers to evaluate the risk and consequences of task failure.

**Lemma 6.1.** For a single task \( t \) which has the probability \( p \) to be completed for the price of \( \text{cost} \), the cost of its error is \( \text{cost}_{err}^{t} = (1 - p)\text{cost} \).

**Proof:** \( t \) has the probability of \( 1 - p \) to fail. Therefore, its failure will statistically result in the loss of \( (1 - p)\text{cost} \), and denoted as \( \text{cost}_{err}^{t} = (1 - p)\text{cost} \).

**Lemma 6.2.** Let \( t_1, t_2, \ldots, t_n \) be a sequence of tasks, and \( t_i(1 \leq i \leq n) \) has the probability \( p(i) \) to be completed for the price of \( \text{cost}(i) \); then the error of the \( k \)th task will cost

\[
\text{cost}_{err}^{t_k} = \prod_{i=1}^{k-1} p(i) \left[ \sum_{j=1}^{k} \text{cost}(j)(1 - p(k)) \right]
\]

**Proof:** The commence of \( t_k \) relies on the success of all previous tasks. Therefore, \( t_k \) has the probability of \( \prod_{i=1}^{k-1} p(i) \) to be executed.

When \( t_k \) is executed, the total cost involved is \( \sum_{j=1}^{k} \text{cost}(j) \), which has the probability of \( 1 - p(k) \) to fail. Therefore, the cost of error for \( t_k \) is \( \prod_{i=1}^{k-1} p(i) \left[ \sum_{j=1}^{k} \text{cost}(j)(1 - p(k)) \right] \).

**Theorem 6.4.** Let \( t_1, t_2, \ldots, t_n \) be a sequence of tasks, where \( t_i(1 \leq i \leq n) \) has the probability \( p(i) \) to be completed for the price of \( \text{cost}(i) \); then the cost of error for the sequence is

\[
\text{cost}_{err}(n) = \prod_{k=1}^{n} \left[ \sum_{i=1}^{k-1} p(i) \left[ \sum_{j=1}^{k} \text{cost}(j)(1 - p(k)) \right] \right]
\]

**Proof:** When \( n = 1 \), according to Lemma 6.1, the cost of error should be \( \text{cost}_{err}(1) = (1 - p(1))\text{cost}(1) \) which is the same as the result from the theorem.

Let us assume \( \text{cost}_{err}(n - 1) \), which represents the cost of error for the sequence of first \( n - 1 \) tasks, is correct. When \( t_n \) is added into the sequence, its error will result in the cost of \( \text{cost}_{err}^{t_n} \).
which can be computed by Lemma 6.2. Therefore, the cost error for the sequence of all \( n \) tasks is the summation of \( \text{cost}_{err}(n-1) \) and \( \text{cost}_{err}^n \).

\[
\text{cost}_{err}(n) = \text{cost}_{err}(n-1) + \text{cost}_{err}^n
\]

\[
= \sum_{k=1}^{n-1} \left( \prod_{i=1}^{k-1} p(i) \left[ \sum_{j=1}^{k} \text{cost}(j)(1 - p(k)) \right] \right) + \prod_{i=1}^{n-1} p(i) \left[ \sum_{j=1}^{n} \text{cost}(j)(1 - p(n)) \right]
\]

\[
= \sum_{k=1}^{n} \left( \prod_{i=1}^{k-1} p(i) \left[ \sum_{j=1}^{k} \text{cost}(j)(1 - p(k)) \right] \right)
\]

Since both the basis and the inductive step have been proved, it has now been proved by mathematical induction that \( \text{cost}_{err}(n) \) holds for all \( n \).

\[\square\]

**Definition 6.12 (Unordered Commitment Dependency).** Let \( \phi_1 \triangleright \phi_2 \). If both \( \phi_1 \) and \( \phi_2 \) are strictly unordered commitments, then they have unordered dependency, denoted as \( \phi_1 \triangleright \phi_2 \), which means their pre- and post- conditions can be executed in any order or in parallel.

\[\square\]

**Definition 6.13 (Unordered Commitment Sequence).** Let \( \phi_1, \phi_2, \ldots, \phi_n \) be a group of strictly unordered commitments which satisfy \( \phi_1 \triangleright \phi_2 \triangleright \ldots \triangleright \phi_n \); then they form an unordered sequence of commitments.

\[\square\]

For a sequence of unordered commitments \( \phi_1 \triangleright \phi_2 \triangleright \ldots \triangleright \phi_n \), the execution order of any \( \phi_i (1 \leq i \leq n) \) and \( \phi_j (1 \leq i \leq n) \) are swappable, because their pre- and post- conditions cast no constraints on the execution order. Compared with a group of independent commitments that also allow arbitrary execution order, the unordered sequence of commitments poses the extra constraint that the failure of any commitment will cause the whole sequence to fail.

**Example 6.6.** In the NetBill example, if commitments \( \phi_1 = \text{CC}(m,c, \text{accept, goods}), \phi_2 = \)
Reasoning Intra-Dependency in Commitments for System Robustness and Performance

CC(c, m, goods, payment), and $\phi_3 = CC(m, c, \text{payment}, \text{receipt})$ are all strictly unordered, they form a sequence of $\phi_1 \geq \phi_2 \geq \phi_3$. That is, agents can choose to execute the actions towards the goals of accept, goods, payment, and receipt in any order. However, the failure of any action will make the whole sequence fail.

Since the corresponding tasks for an unordered commitment sequence can be executed in any order, Theorem 6.4 can be applied to find a better task execution order to reduce the cost of error. However, since $n$ tasks have $n!$ possible execution sequences, brute force evaluation is expensive to perform. Therefore, some heuristic methods need to be applied. For example, the task with low $p$ and high cost should be placed at the end of the sequence, while the one with high $p$ and low cost should be at the beginning. In the future, we will apply genetic algorithms to find better solutions.

When the system decides to execute a sequence of strictly unordered commitments in parallel, the sequence can be treated as an arbitrary order of tasks. Theorem 6.5 provides the method to compute the cost of error for concurrent task execution. Therefore, the potential loss from concurrent execution becomes quantified and comparable to its efficiency gain, helping to make better decisions between concurrent and sequential scheduling. Specifically, let $\text{cost}_{\text{err}}^\parallel(n)$ stand for the cost of error for concurrent task execution, and $\min(\text{cost}_{\text{err}}(n))$ for the minimum cost of error for sequential executions; then $\text{cost}_{\text{err}}^\parallel(n) - \min(\text{cost}_{\text{err}}(n))$ represents the risk brought by concurrent execution. If the efficiency gain outweighs this difference, concurrent execution should be followed. The discussion is especially helpful for risk evaluation when scheduling a commitment to run concurrently, as in Section 6.4.2.

**Theorem 6.5.** Let $t_1, t_2, \ldots, t_n$ be a group of tasks, where $t_i(1 \leq i \leq n)$ has the probability $p(i)$ to be completed for the price of $\text{cost}(i)$; then the cost of error for the concurrent execution of tasks is

$$\text{cost}_{\text{err}}^\parallel(n) = \sum_{i=1}^{n} \text{cost}(i)(1 - \prod_{i=0}^{n} p(i))$$

**Proof:** When tasks are executed in parallel, the error of any task will cause the whole system to fail. Therefore, the probability that the system succeeds is $\prod_{i=0}^{n} p(i)$.

During the execution, the involved cost is the summation of all tasks. Therefore, the cost of error for parallel execution is $\sum_{i=1}^{n} \text{cost}(i)(1 - \prod_{i=0}^{n} p(i))$. 
Table 6.3: Cost and probability of success for NetBill example

<table>
<thead>
<tr>
<th>Task</th>
<th>Cost</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>goods</td>
<td>5</td>
<td>0.9</td>
</tr>
<tr>
<td>payment</td>
<td>20</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Example 6.7. In the NetBill example, if commitments $\phi_2 = CC(c, m, \text{goods}, \text{payment})$ and $\phi_3 = CC(m, c, \text{payment}, \text{receipt})$ form a sequence of $\phi_2 \succeq \phi_3$, then the task execution sequence can be obtained from $\text{goods} \parallel \text{payment} \parallel \text{receipt}$. For simplicity, we only focus on the first two tasks. The cost and probability of success for each task are shown in Table 6.3.

The cost of error for the execution order of $\text{goods} - \text{payment}$ is $5 \times (1 - 0.9) + 0.9 \times (5 + 20) \times (1 - 0.8) = 5$, while that for $\text{payment} - \text{goods}$ is $20 \times (1 - 0.8) + 0.8 \times (20 + 5) \times (1 - 0.9) = 6$. Therefore, the sequence with lower cost of error should be chosen.

The cost of error for the concurrent execution of the two tasks is $(5 + 20) \times (1 - 0.9 \times 0.8) = 7$, which is only 2 more than the value of the safest sequential execution. Therefore, if the efficiency gain from concurrent execution is greater than 2, the system should follow concurrent execution.

6.4.4 Robust Scheduling of Combined Commitments

We have discussed the safety level for single commitments in Definition 6.9. However, the ability of inferring the level of combined commitments is also important for agents to work and operate in more complicated environments. The general way to reason about the safety property of a complex commitment is to decompose it into several child commitments and then aggregate their safety properties together. The result of aggregation affects scheduling and synchronization among the commitment’s children. The safety level of the complex commitment $\phi_1 \land \phi_2 \land \ldots$ is the minimum safety level of its children, and that of $\phi_1 \lor \phi_2 \lor \ldots$ is the maximum one of its children.

For or-combined commitments, it is intuitive for the agent to select the child with the highest safety level, while for and-combined commitments, we define a new parameter to guide the scheduling. Two functions are used in the following definition, where $\text{min}(S)$ means the minimum
value of a set $S$ and $abs(\cdot)$ returns an absolute value. The value of $\min(S)$ is 0 if $S$ is empty.

**Definition 6.14 (Commitment Safety Indicator).** For the complex commitment $\phi_1 \land \phi_2 \land \ldots$, let $S_{rec}$ be the set containing the safety level value of its recoverable children (which are in level 2 or 3) and $S_{nrec}$ be the set containing that of its non-recoverable children (which are in level 0 or 1). The safety indicator of an and-combined commitment is equal to $\text{abs}(\min(S_{rec}) - \min(S_{nrec}))$.

By withholding an implicit goal, to keep the safety indicator of its and-combined commitments as high as possible, the agent can schedule its execution more robustly, while still providing high parallelism. To keep the safety indicator high, the schedule of the agent will prefer recoverable commitments to non-recoverable ones, and then prefer commitments with lower safety levels within each child group.

The preferences come from the fact that failure of any child leads to overall failure and recovery. Thus, if recoverable children are executed first, non-recoverable ones have a better chance not to get affected by them. And scheduling the commitment with the lowest level in each group first helps to distribute the possible loss among them, because it forces the commitments to execute at a similar rate. Therefore, it is less likely that some commitments will have completed while others have not started yet.

### 6.5 Experiments and Evaluations

We have discussed that a strictly ordered commitment (SOC) requires the debtor and creditor to execute strictly one after the other, while a strictly unordered commitment (SUC) allows them to run concurrently. The Netbill example shows that our method can schedule SUCs from a sequence of tasks to run in parallel, thus improving the efficiency of the system. In this section, experiments are performed on applications which can be modelled as directed acyclic graphs (DAGs). As DAG is a fundamental concept to model and represent procedural knowledge [93], our method can also be applied to a wide range of application domains, such as business process management systems and service-oriented computing [90]. To the best of our knowledge, applying commitment protocols to produce a concurrent schedule for a DAG-modelled task network has not been addressed in previous research. Therefore, we can only provide the evaluation and
comparison between schedules that do and do not execute strictly unordered commitments in the system concurrently. To simplify the discussion, we only consider the types of SOC and SUC.

The environment for the experiments was created by adopting the DAG dataset of the Resource-Constrained Project Scheduling Problem, provided in the Project Scheduling Problem Library [126]. Each DAG definition in the dataset specifies a set of task nodes, the precedence relations among them, and the duration of each task.

To transform the test sets for our experiments on commitment-based scheduling, we converted each edge in the DAG to be a commitment. For example, the edge from task $a$ to $b$ will be represented as $CC(b,a,\text{Successful}(a),\text{Successful}(b))$ which means that $b$ promises to complete if $a$ has completed. $\text{Successful}(x)$ is a predicate to check if task $x$ has been successfully executed. The execution time of the commitment comes from that of $b$.

After the conversion, a commitment set which specifies the same semantics as the DAG was obtained. Each element of the set was then randomly assigned to be either an SOC or an SUC. The parameter $P(\text{SUC})$ was introduced to indicate the probability that a commitment is strictly unordered.

When the feature that SUC allows concurrent execution is not considered, all commitments need to preserve the precedence order defined in the DAG as if all of them are SOCs. In this case, the total execution time of the commitment set, denoted as $t_{\text{ordered}}$, can be computed by finding the critical path [82] of the DAG, which is the path with the longest overall duration. In contrast, if SUC is exploited to boost the performance, the concurrent execution time of the commitment set is denoted as $t_{\text{unordered}}$. Theoretically, $t_{\text{unordered}} \leq t_{\text{ordered}}$, since the worst case is that no commitment can run concurrently.

To evaluate the effect of utilizing SUCs, we introduced the parameter $\text{TimeUsageRatio} = \frac{t_{\text{unordered}}}{t_{\text{ordered}}}$, which represents the ratio between the duration with and without executing SUCs in parallel. The lower the value, the better the performance.

In fact, $t_{\text{ordered}}$ should be a fixed value for a certain commitment set, while $t_{\text{unordered}}$ varies when either $P(\text{SUC})$, or the distribution of SUCs in the set, changes. Therefore, for each commitment set, $P(\text{SUC})$ was increased from 0 to 1 by 0.1 to show its effect on performance. In addition, for each $P(\text{SUC})$, the experiment was repeated 200 times to find the impact of different distributions of SUCs in the commitment set. A summary of the parameters used in the experiments
Table 6.4: Key parameters in the experiments of concurrent scheduling for strictly unordered commitments

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Parameters</td>
<td>$P(SUC)$</td>
<td>probability a commitment is strictly unordered</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>Evaluation Parameters</td>
<td>$TimeUsageRatio$</td>
<td>the time ratio of concurrent to sequential execution</td>
<td>[0, 1]</td>
</tr>
</tbody>
</table>

are listed in Table 6.4.

We have experimented on a large number of graph instances from 30-, 60-, 90-, and 120-node dataset groups and observed similar results. Figure 6.4 is the box plot showing the randomly selected instances from each group with the name of each sub-graph listed in the caption for clarification.

The results indicate that system performance has a steady improvement when the percentage of $SUC$ increases. The performance improvement is not only reflected in the median value, but also in the best case and the worst case values. The shrinkage of both the value range and the interquartile range shows that the performance improvement is more stable and obvious at higher $P(SUC)$ value.

The results also show that the value range and the interquartile range are wider in 30-node cases. Since there are fewer nodes in the graph, the length (in term of the number of edges) of the critical path is shorter. Therefore, the distribution of $SUC$s, especially when $P(SUC)$ is low, has a stronger impact on the result. In other words, a longer path has a lower chance of getting all its edges being $SUC$s.

Our approach is shown to be scalable. The increase of graph size does not cause the performance improvement to drop. On the contrary, the boost of performance becomes more accessible for graph instances since the interquartile range keeps on decreasing. Therefore, users can incrementally extend the graph definition and tag eligible commitments to be $SUC$s, thus progressively developing and tuning their systems.
6.6 Related Work

Commitment theory is one of the core parts of multi-agent research, because it helps to specify and constrain the relationship among agents [75, 140]. The formalization and application of commitment protocols have been carried out extensively [101, 158, 163, 173]. Previous work focuses on using commitments to specify the semantics of communication among agents and to reason about all possible execution paths. However, the work assumes that each commitment, as well as each part of it, is independent and atomic. This assumption greatly reduces the complexity of system modeling, but ignores agent’s important features of autonomy and activeness and the issue of concurrency coordination. Therefore, it does not cover the issues of how individual agents can deliberatively reason and manipulate their commitments to make the best selection from the available execution paths or concurrent schedules at run time, especially in the case of possible commitment breaches, which often occur in real applications.

[86] also discusses the over-committed problem in existing research, where agents intend to achieve the condition before the goal of the commitment. They do not provide a balanced
reasoning strategy for agents to choose a schedule, with the requirements of robustness and concurrency considered, from various available execution paths at run time.

[57] formalizes groundedness within an extended BDI (Belief, Desire, Intention) logic, thereby enabling agents to reason about their commitments. However, their focus is on bridging the gap between commitments and the underlying agent framework, without considering the detailed features of commitments. Thus, the issues of robustness and concurrency related to commitment fulfilment are not addressed.

[28, 150] model agent execution as goal-plan trees and apply summary information to avoid resource conflicts while pursuing multiple goals in parallel. The work is similar to ours in the sense that agents are active in choosing available execution paths. However, it resides in the abstract level of plan definitions, while ours is in social commitments. Therefore, both the purpose and the methods applied in the two research approaches are different.

Leveled-commitment contracting [135], which is similar to the concept of recoverable commitments in our model, is proposed as a backtracking instrument to deal with the uncontrollability of the committing and decommitting behaviour of agents, in order to build robust interactions. In fact, it can be utilized in our model to help the design of recovery procedures.

Concurrent scheduling has been heavily studied in high performance computing [48]. However, it remains hard to decompose tasks into parallel parts while preserving complex inter- and intra-dependencies. Commitment-based modeling provides a natural approach to master this complexity by splitting the system into finer components at the semantic level, and utilizing the autonomous features of agents to manage concurrent execution. To our knowledge, applying commitments to accompany runtime concurrent scheduling has not been addressed in previous research.

Moreover, the traditional concurrent scheduling approaches become impractical in dynamic multi-agent environments because they require concurrent schedules to be pre-programmed. In contrast, our proposal enables agents to identify and control the opportunities of concurrent execution at runtime by reasoning about their commitment-based dependencies. Our approach is capable of not only improving system performance, but also ensuring the robustness of the concurrent schedule.
6.7 Discussion and Conclusion

In this chapter we have addressed the intra-dependency between the pre- and post- conditions of a commitment and adopted a bottom-up approach to discuss how agents can benefit from this dependency knowledge. A novel classification of commitments based on the temporal dependency of commitment’s pre- and post- conditions is proposed first. Thereafter, a set of refactoring rules, which enable composing or decomposing commitments to suit the runtime situation, is defined and proved for each commitment category. An agent can benefit from refactoring by executing its commitments in a batch or in parallel, thus improving system performance. With the provided reasoning strategy and corresponding commitment manipulation operations, the agent can evaluate the execution context to make optimized decisions at runtime, thus pursuing a robust and reliable schedule. At the same time, our reasoning and scheduling model enables the agent to follow concurrent commitment execution to boost system performance with a minimized possibility of loss, even in the case of commitment breaches.

Assume there are \( n \) commitments. The complexity of refactoring a single commitment is \( O(n) \) because it requires iterating through each commitment. The decision process of reasoning is relatively simple, with \( O(1) \) for single commitment scheduling and \( O(n) \) for combined commitments. In fact, the complexity of reasoning lies in evaluating the trustworthiness and recovery cost of an agent. The complexity highly depends on the model used for acquiring these values. If a central reputation server like eBay is available in the system, and the recovery costs are predefined, the complexity is reduced to \( O(1) \), as there is only one query process needed. We are currently considering applying research results from trust [21, 64, 78, 80] and process mining [32, 156] to calculate the information automatically.

The experiments performed on DAGs showed that system performance can be improved by executing strictly unordered commitments (\( SUC \)) in parallel. In addition, the increase of the percentage of \( SUCs \) in the commitment set will bring steady improvement. Therefore, users can adopt an incremental approach to specify and tune their task network by progressively tagging commitments as \( SUCs \).

To our knowledge, this is the first proposal of this kind and has significant practical importance. For example, after dependencies between the pre- and post- conditions of commitments are identified, possible concurrent executions can be inferred automatically in the form of branching
trees. At certain points of the tree, the agent can use inference rules, along with the notion of trust, compensability, and recoverability, to select the most beneficial branch. Since this formalism allows agents to make run time decisions, it helps to build robust and recoverable agents.
Chapter 7

Applications of ARTS

In previous chapters, we have introduced the ARTS model for building robust multi-agent systems while relieving developers from being concerned with low-level details of concurrency control and failure recovery. The ultimate goal of our work is to benefit software engineering in distributed and dynamic environments. Therefore, we propose to apply both the ARTS model and its exception handling and prevention mechanisms to the design and development of real applications. Specifically, we use the ARTS model and the multi-step backtracking recovery mechanism to improve the adaptiveness and flexibility of business process management systems, and we use trust-based plan management to enhance the reliability of scheduling and execution in scientific workflow management systems.

7.1 Adaptive Business Process Management System

7.1.1 Background

A business process specifies the composition and coordination of a set of business activities or tasks to achieve certain business objectives. Along with the globalization of economy, the scale and complexity of business processes, which usually need to span multiple organizations and nations and operate in dynamic and open environments, are ever-increasing. Therefore, the effectiveness of process management becomes a critical factor in business success.

To improve the efficiency and reduce the cost of management, a business process management
(BPM) system is proposed, which provides methods, techniques, and software to design, enact, control, and analyze operational business processes involving humans, organizations, applications, documents, and other sources of information [142, 157]. The system automates related business processes to coordinate and streamline business activities, thus reducing operational costs while boosting system performance and organization productivity [62, 142].

One important trend of BPM research and products is integration with the technology of web services [22, 94]. On one side, a business process can be modularized and encapsulated as a service to support Internet-wide interoperability via open standards and light-weight protocols, including SOAP (Simple Object Access Protocol) [161], WSDL (Web Services Description Language) [24], and UDDI (Universal Description, Discovery, and Integration) [114]. On the other side, BPM systems provide a framework of service management including composition, enactment, monitoring, and tracking. Specifically, BPEL4WS (Business Process Execution Language for Web Services) [4] extends the Web Services interaction model and enables it to support business transactions by providing a language for the formal specification of business processes and business interaction protocols.

A critical challenge in building practical BPM systems is to allow users to maintain system robustness and reliability with respect to correct execution even in the presence of abnormalities. However, existing BPM systems, which largely rely on workflow technology as the underling process management engine [85, 157, 171], require developers to predefine all business processes, together with possible exception handling processes, as static and structured workflows. Therefore, although they may be well suited to applications with standard inputs, processes, and outputs as in traditional closed business environment, they cannot satisfy the challenges brought by modern business development.

As the operating environment becomes increasingly complex, dynamic, and error-prone, it is extremely challenging for designers and programmers to find out all possible combinations of exceptions, as well as to design corresponding handling methods. Therefore, a flexible, systematic, and autonomic approach for exception handling is essential for the success of applying complex BPM systems into wider fields of application.

Addressing these issues, we propose to use the ARTS platform to construct business processes and use multiple-step backtracking to handle possible exceptions. The work has been published
in [165] and will be briefly introduced in the following section.

### 7.1.2 Multiple-Step Backtracking of Exception Handling in Autonomous Business Process Management

Multi-agent systems have been extensively studied as a powerful high-level decomposition and abstraction tool for analyzing, designing, and implementing complex software systems [77]. Many researchers and practitioners have noticed the fundamental relationship between agents and workflow systems [45, 76], and have proposed various approaches to building dynamic and adaptive workflow systems with the concept of agents, and vice versa.

Providing a higher level of abstraction, agents help to simplify the modeling and construction of flexible BPM systems operating in open and distributed environments. To deal with exceptions, most multi-agent systems apply the mechanism of backtracking to recover and retry the failed execution. However, they back track in a rigid step-by-step manner until one alternative execution path is found. Moreover, they follow the defensive route of the recovery-then-try pattern. Due to the dynamic and nondeterministic features of complex environments, rigid recovery on the reverse chronological order of execution history will in many instances be meaningless and inappropriate. Some systems attempt to consider starting a fresh plan after the exception. However, this approach is too computationally expensive and rarely practical to deploy because it requires a complete knowledge about the running environment and needs to consider all previous actions. As a result, developers are forced to consider low-level details of disturbances, failure, or uncontrolled interactions between workflow actors for the requirement of robustness and reliability.

To manage the complexity arising from the dynamic and complex running environment, we propose to apply the ARTS model and the multi-step backtracking exception handling mechanism to design and construct BPM systems for the advantages of robustness, flexibility, and adaptability. The main features of the proposed approach can be summarized as follows. The details on the ARTS model and multi-step backtracking have been discussed in Chapter 3 and Chapter 4 respectively; therefore, they are not restated in this section.

- Business activities and tasks are composed by the ARTS programming model to hierarchically and recursively specify the normal and exception handling processes.
- The execution of business tasks and processes is delegated to BDI agents, to utilize their
beneficial features of event-driven and means-end reasoning. Therefore, the execution tree of a business process towards a certain business goal is generated at run-time reactively to its surroundings.

- Multiple-step backtracking is applied as the default exception handling and recovery mechanism for the BPM system.
- The open nested transaction model is used to encapsulate process execution and backtracking, to gain system-level support for concurrency control and automatic recovery.

The nested tree structure is applied to depict the BPM system execution. During the construction of the tree structure, alongside system execution, the choice points are stored and maintained in a stack. By iterating over the stack, the system can find and execute a suitable plan from previously applicable ones, to achieve its goal as soon as an exception occurs. This “jump” procedure strikes a balance between complete replanning and rigid step-by-step backtracking after exceptions occur, by utilizing previous planning results in determining the response to failure. Because the substitutable path is allowed to start prior to or in parallel with the compensation process, the system can achieve its goals more directly and with higher efficiency.

Compared with conventional workflow systems, the proposed BPM execution model is more dynamic and adaptive to the environment, because the execution tree is constructed at run time, in accordance with the real situation.

To provide automatic and system-level support for concurrency control and exception recovery, open nested transactions [170] are integrated into the multiple-step backtracking model. Compared with other approaches, which apply transaction models [66] into workflow [58] or multi-agent systems [110, 127], our method unites the beneficial features of event-driven and means-end reasoning from BDI agent systems and utilizes a flexible backtracking approach to allow the execution to “jump” back several levels at once to continue its execution towards the goal, in the case that backtracking to one level in the execution tree does not solve the problem.

Our approach also frees system programmers from considering low-level details of concurrency control and exception handling, because transactional execution automates these issues. Combining and utilizing several beneficial features of BDI agents, the open nested transaction model is tightly integrated into the BPM system. Both BDI data structures and the deliberation cycle are leveraged to maximize the functionality of transaction management.
7.2 Trust-based Scientific Workflow Management

7.2.1 Background

Scientific workflow is an important unifying mechanism to support the modelling, management, and execution of large-scale scientific computing in many complex e-science applications such as climate modelling, astrophysics, medical surgery, and disaster recovery [2, 95, 98]. To achieve their goal, domain users define a scientific workflow by composing a large number of computing or data intensive tasks, as well as specifying dependencies between them [179]. For example, large scale scientific computing can usually be achieved by several alternative workflows. In addition, for the same task in a workflow, there are several eligible service providers or agents to accomplish its requirements. During execution, the system needs to evaluate eligible workflows and then select the most suitable one (based on certain criteria) for execution. A scheduler which assigns each task in the selected workflow to an appropriate service agent is also required in this framework. Much research has been carried out in scheduling, and one can assume that good schedules can be found using heuristic-based methods, as in general it is an intractable (NP-Hard) problem [152].

A workflow specifies a task network to compose and organize a group of tasks to achieve a well-defined goal, such as weather forecasting or gene alignment in e-science applications [38, 189, 190]. The scheduling of the workflow involves discovering resources or services and delegating tasks to suitable service agents to meet users’ requirements and constraints. Since the terms of service provider and agent are both commonly used for the executor of a task, we will use them interchangeably in this chapter.

The actual execution flow, or plan, is a sub-graph of the workflow definition (when alternative execution paths exist), and can be modelled as a directed acyclic graph (DAG) in which each node represents a task and each arch is a precedent relation between two tasks. Tasks in the graph are delegated to service agents for completion. If the participating agents’ execution capacities and costs are guaranteed to remain the same throughout the scheduling and execution of the workflow, the resources of the system are considered static; otherwise, they are dynamic. Correspondingly, problems of static scheduling and dynamic scheduling are recognized respectively.

Research in high performance computing has developed various static scheduling algorithms to help systems optimize execution time and cost to satisfy users’ requirements on quality of
Applications of ARTS [113, 138, 184, 186]. However, these methods become restrictive in scientific workflows that are expected to operate in open, dynamic and internetworked information environments [189], such as Grid [11, 51, 52, 184], Cloud [20, 68, 69, 137], and Service-Oriented computing [34, 44, 60, 72].

Addressing this realistic application domain, including the execution of scientific workflow in service-oriented or grid environments, this section focuses on the issue of dynamic scheduling. In fact, there are two fundamental issues making static scheduling infeasible.

- Existing work in workflow scheduling assumes that all participants will adhere to their promises. However, the participating agents may no longer be 100% trustworthy. They may break their commitment when they feel that doing so is profitable. In other words, service agents may behave dishonestly and sometimes maliciously, especially when pursuing their own interest and benefit [23, 53].

- There is usually more than one feasible execution flow with different needs of resources to achieve the same goal. The selection process described in existing work, which compares the scheduling result of each alternative flow, is both time-consuming and expensive. As resources become dynamic, frequent changes of the environment will force repeated rescheduling on all possible flows, which is expensive and impractical.

Since a service agent may behave differently from its promises, the scheduler cannot merely choose the one with the least cost and/or shortest execution time, because the agent may deny the request at run time, thus making the whole schedule obsolete.

To address these issues, we propose to apply the trust-based scheduling and execution management mechanisms introduced in Chapter 5 to the scientific workflow management system, thus increasing the reliability and robustness of the schedule, in terms of the participation of agents that are more likely to fulfil their commitments. The result has been published in [166] and will be briefly described in the following section.

### 7.2.2 Trust-Based Robust Scheduling and Runtime Adaptation of Scientific Workflow

A scientific workflow can be modelled as an AND/OR graph, which is commonly applied to represent the execution structure of applications. We refine and extend the existing representation
of workflows to suit our purpose of dynamic scheduling. Moreover, we use the name Coordinator for the functional part that is in charge of flow selection, task scheduling, runtime monitoring, and schedule updating in the workflow management system.

Workflows are defined by composing individual tasks, each of which is an atomic unit of work from the Coordinator’s point of view, according to the rules shown in Definition 7.1, which are in EBNF notation. The three composition operators share the same meaning as in Section 3.3.1: “;” for sequential composition, in which each elements is executed one after the other; “∥” for parallel composition, in which all elements are launched together; and “;” for selective composition, in which one of the alternatives is chosen for execution.

Definition 7.1 (Workflow Composition). Task composition serves to organize a group of atomic tasks, according to their functionalities, to build a workflow for a certain goal. Let \( \mathcal{T} \) be the set of tasks and \( t_i \in \mathcal{T} \).

\[
< \text{atomicTask} > ::= \epsilon \mid t_i \mid \text{start} \mid \text{end}
\]

\[
< \text{Workflow} > ::= \text{start} \; ; \; < \text{Branch} > \; ; \; \text{end}
\]

\[
< \text{Branch} > ::= < \text{atomicTask} > \mid < \text{sequence} > \mid < \text{parallel} > \mid < \text{selection} >
\]

\[
< \text{sequence} > ::= < \text{Branch} > \; ; \; < \text{Branch} >
\]

\[
< \text{parallel} > ::= < \text{Branch} > \; || \; < \text{Branch} >
\]

\[
< \text{selection} > ::= < \text{Branch} > \; , \; < \text{Branch} >
\]

The task \( \epsilon \) is a dummy task in the system, which does nothing and always succeeds. It acts as a bridge to connect branches in task composition. The \( \text{start} \) and \( \text{end} \) tasks are special \( \epsilon \) tasks serving as the starting and ending points of the workflow graph respectively.

The selection of a task for execution must enforce the precedent relationship specified by the graph to preserve the application logic. For sequential composition and parallel composition, the Coordinator simply selects both branches together for execution. However, for selection composition, the Coordinator needs to decide which branch to follow, thus making the actual execution path non-deterministic and only known at run time.
The Coordinator can decompose the workflow into a set of execution flows or plans, denoted as $\mathcal{P}$, containing all possible execution paths from start to end, by combining different branches at each selective composition. Each plan, denoted as $p \in \mathcal{P}$, is capable of achieving the goal of the workflow, and can be represented as a DAG that specifies the runtime execution flow. Since both task numbers and task types may differ from plan to plan, the resource requirements of each plan will also differ. Let $\mathcal{T}_p$ be the set containing all tasks in $p$.

Usually, there are several agents with different processing capacities eligible to complete each $t_i$ in the system. Let the set of candidate agents for $t_i$ be denoted as $\mathcal{A}_i$ and $A_{ij} \in \mathcal{A}_i$ be the $j$th element. For every $t_i \in \mathcal{T}_p$, the scheduler will choose an agent from $\mathcal{A}_i$ and delegate the work to it.

To operate in a dynamic environment, the Coordinator should also be able to detect changes and exceptions and then act accordingly. A set of reactive rules is applied to make the Coordinator environment-aware.

**Definition 7.2 (Workflow Coordinator).** The Coordinator can be represented as a tuple $\text{CO} = \langle \mathcal{W}, \mathcal{A}, I, R \rangle$.

- $\mathcal{W}$ represents the graph definition of the workflow, which can be decomposed into a set of plans $\mathcal{P}$. As a result, $\mathcal{T}_p$, the set containing all tasks that appear in plan $p \in \mathcal{P}$, can be derived.
- $\mathcal{A}$ is the set of service agents in the system. $\mathcal{A}$ can be further divided into subsets $\mathcal{A}_i$ whose elements are capable of achieving $T_i$.
- $I$ is the solution to achieve $\mathcal{W}$. It contains two parts: a selected plan $p \in \mathcal{P}$; and a mapping function $\delta : t_i \rightarrow A_{ij}$, which assigns each task $t_i \in \mathcal{T}_p$ to an eligible agent $A_{ij} \in \mathcal{A}_i$.
- $R$ is the set of event-condition-action (ECA) rules to monitor system changes and adapt the schedule accordingly.

**Example 7.1.** The AND/OR graph in Fig. 7.1 represents a simplified travel workflow $\mathcal{W}$. $\mathcal{W}$ can be decomposed into the plan set $\mathcal{P}$ containing four elements:

- $p_1 : $ start $\rightarrow$ Train $\rightarrow$ Credit card $\rightarrow$ end,
- $p_2 : $ start $\rightarrow$ Train $\rightarrow$ Check $\rightarrow$ end,
• $p_3 : \text{start} \rightarrow \text{Flight} \rightarrow \text{Credit card} \rightarrow \text{end},$
• $p_4 : \text{start} \rightarrow \text{Flight} \rightarrow \text{Check} \rightarrow \text{end}.$

The solution $I$ of $\mathcal{W}$ is a plan $p_i$ from $\mathcal{P}$ as well as the service assignment for each task in $p_i$. Taking $p_3$ as the solution, $I$ also indicates the selection of the airline and credit card company.

After a workflow is mapped to the terms of plans, tasks, and service agents, the methods discussed in Chapter 5 can be straightforwardly applied to achieve better schedules and execution management. The trust-based management for the life cycle of a scientific workflow can also be classified into three main scenarios: plan evaluation, scheduling, and monitoring. The algorithms for each scenario have been presented and discussed in Chapter 5. Interested readers can refer to it for details.

### 7.3 Discussion and Conclusion

As the prevalence of the Internet increases, applications are continuously becoming distributed and networked. In general, they involve the coordinated execution of large numbers of processes in open and dynamic environments, which makes system development and management complicated and difficult, especially with respect to handling possible exceptions. To address these issues, the ARTS model has been introduced on top of multi-agent platforms in previous chapters to increase system robustness while relieving developers from considering every detail of exceptions.

To make our approach more concrete, in this chapter, we have discussed the application of the ARTS model, as well as its exception handling and prevention mechanisms, to the design and development of robust and flexible business process management systems and scientific workflow management systems. The results prove that the ARTS model is practical and beneficial in resolving the complexity brought by the challenging execution environments. Following
similar methods, our model can also be applied to extend other distributed applications, such as web services and service-oriented computing [117, 147], for improved reliability and reduced programming complexity.
Chapter 8

Conclusion

Multi-agent programming models allow the realization of complex applications in distributed, open, and dynamic environments. However, their lack of systematic support for handling exceptions makes them impractical in real applications. Therefore, this thesis proposes a reliable multi-agent platform, covering the life cycle of program specification, scheduling, and execution, by integrating the concept of transactions to provide intrinsic support for concurrency control, recoverability, and robustness, thus relieving developers from dealing with the low level complexity of exception handling. In this chapter, we first put our work into a historical perspective by comparing transactions, workflows, and multi-agent systems. Then, we summarize and conclude the thesis, and discuss some interesting future research directions.

8.1 Transactions, Workflows, and Multi-agent Systems

The ACID transaction model resides at the core of relational database systems, for concurrency control and program correctness. However, ACID transactions require the structure of the program to be flat, which is too restrictive and usually not suitable for domain applications. Therefore, various advanced transaction models (ATMs) have been proposed to support hierarchical organization and relaxed ACID features with the help of domain knowledge. Many ATMs, such as sagas [56] and flexible transactions [47, 187], are implemented using workflow systems, which play a fundamental role in service-oriented applications for service composition and coordination.

However, workflows require the program structure to be predefined and remain static during
the execution, which is usually not satisfiable in open and non-deterministic environments. In contrast, multi-agent systems support the composition of services at runtime reactively, therefore being more suitable in modelling applications in Internet computing.

Figure 8.1 illustrates the evolutionary relationship among transactions, workflows, and multi-agent systems, as the complexity of the execution environment varies. A workflow system evolves into a multi-agent system when its operating environment becomes more and more complex and dynamic, and condenses into a database transaction when it obtains full control to the running environment.

The shared theme of transactions, workflows, and multi-agent systems, is to ensure system correctness by executing concurrent tasks cooperatively towards their goals during normal execution, and to guarantee system reliability by repairing affected participating tasks coordinately to reach a semantically correct state when exceptions occur. With this background, we propose and design the ARTS platform by absorbing valuable concepts and features, such as failure atomicity, concurrency control, and compensation, from transactions and workflows. These features are integrated with the merits of autonomy, reactivity, pro-activeness, and social ability brought by multi-agent systems, to construct a flexible and well-organized exception minimization and handling platform from a programming and software engineering perspective. The ARTS platform contributes to both the research and the practice of distributed computation in the Internet era.
8.2 Thesis Summary

In this dissertation, we have investigated and developed mechanisms to integrate intrinsic support for concurrency control, recoverability, and robustness into multi-agent systems. The research covers agent specification, planning and scheduling, execution, and coordination together, to reduce the impact of environmental uncertainty.

A thorough survey, along with a taxonomy of existing proposals and approaches to building robust multi-agent systems, was first provided (in Chapter 2). In addition, the merits and limitations of each category were also highlighted.

Then, we introduced the ARTS platform (in Chapter 3), in which agents are abstractly considered as executors of encapsulated task entities that comply with a set of execution constraints on both normative execution and compensation semantics. The ARTS platform allows agent developers to compose recursively-defined, atomically-handled tasks to specify scoped and hierarchically-organized exception-handling plans for a given goal. It also provides a straightforward and easy-to-use programming structure to specify the exception handling knowledge in case of task failure, and supports the automatic selection, execution, and monitoring of appropriate plans in a systematic way for both normative and recovery execution.

In addition to explicitly-specified exception handlers for certain events, we have proposed multiple-step backtracking (in Chapter 4), which maintains a tradeoff between replanning and step-by-step plan reversal, to serve as the default exception handling and recovery mechanism in ARTS. Multi-step backtracking utilizes previous planning results in determining the response to failure, and allows the substitutable plan to start prior to or in parallel with the compensation process, thus allowing an agent to achieve its goals more directly and efficiently in the presence of exceptions. We have also discussed the utilization and integration of several beneficial features of the BDI agent model, such as its data structure and deliberation cycle, together with an open nested transaction system, to inherit the benefits of architectural-level concurrency control and the distributed management of participating plans. Experiments show that these systems, even though they are composed of unreliable agents, are more reliable and robust in achieving the design goal with higher efficiency when the multiple-step backtracking model is applied.

Besides exception handling, we have also proposed two exception minimization mechanisms. Trust-based plan management (in Chapter 5) incorporates trust, which indicates the probability
that an agent will comply with its commitments, along with the constraints of system budget and deadline, to improve the predictability and stability of the schedule. To deal with exceptions during execution, it adapts and evolves the schedule at runtime by interleaving the processes of evaluation, scheduling, execution, and monitoring in the life cycle of a plan. Experiments show that schedules maximizing participants’ trust are more likely to survive and succeed in open and dynamic environments. The results also prove that the proposed plan evaluation approach conforms to the simulation result, thus being helpful for plan selection.

Commitment-based agent interaction management (in Chapter 6), which focuses on the interaction and coordination protocol among agents, augments agents with the ability of reasoning about and manipulating their commitments to maximize system utility. We have studied the temporal dependency between each commitment’s pre- and post- conditions, and provided a novel classification of commitments accordingly. We have then proved a set of inference rules to benefit an agent by means of commitment refactoring, which enables the composition and/or decomposition of its commitments to optimize runtime performance. Finally, we have discussed the pros and cons of an agent scheduling and executing its commitments in parallel, and proposed a reasoning strategy and an algorithm to minimize possible loss when the commitment is broken, and maximize overall system robustness and performance. Experiments show that concurrent schedules based on the features of commitments can boost system performance significantly.

Finally, we have applied the ARTS platform (in Chapter 7), especially its exception handling and minimization mechanisms, to build and extend real applications running in open and dynamic environments, to achieve flexibility and robustness. Specifically, we have used the ARTS model and the multi-step backtracking recovery mechanism to improve the adaptiveness and flexibility of business process management systems, and we have used trust-based plan management to enhance the reliability of scheduling and execution in scientific workflow management systems.

8.3 Future Research Directions

There are many new research directions that can be carried out in the future to make agent systems more robust and reliable. They include:

- In the thesis, we have mainly discussed the situation that one exception occurs at a time.
However, in distributed and concurrent execution, several exceptions can occur concurrently in different processes. In general, we cannot handle them separately since there might be dependencies among them [107, 132, 178]. Therefore, we need to design a resolution algorithm, and corresponding recovery mechanisms, for multiple exception handling.

- During the research, we found that Teleo-Reactive Programs (TRPs), proposed by Nils J. Nilsson [29, 68, 111, 112], provide an elegant agent control formalism, which can enhance the ARTS model with the feature of goal persistence in case of exceptions. Thus, we plan to extend our work by assimilating the core concepts from TRPs.
- Market-based approaches, especially the auction model, have been widely studied in research on resource allocations [27, 31, 169, 180]. Since they allow more flexible pricing strategies for agents, and usually produce more efficient outcomes, we intend to investigate the relationship between the auction, trust, and scheduling. Moreover, we will apply the auction model as the resource discovery and pricing mechanism.
- In the thesis, we have discussed the use of trust to find trustworthy agents for task execution. In contrast, much work has been carried out on fraud detection to filter out malicious agents, using technologies from data mining and machine learning [26, 40, 115, 188]. We plan to integrate both approaches to improve the management of agents.
- Some researchers have proposed to add redundancy recourses, such as duplicated execution node or spare execution time, into the schedule, trading for extra reliability and robustness [3, 12, 36, 176]. In contrast, we have studied the use of agent trust and commitments for the same purpose. We believe that the two approaches can be combined together to achieve a better result than either of them, and will investigate and realize this idea.
- We have discussed agents' capability of reasoning about and manipulating their commitments, to achieve improved efficiency and robustness. However, a quantified risk evaluation for running commitments in parallel is still incomplete. Therefore, we expect to study and apply knowledge and achievements in research on decision making and game theory [54, 105, 118, 136, 164] to help agents make more precise decisions.
- The Phoenix project [9, 10, 96] from Microsoft Research provides a transparently persistent stateful programming model for Internet applications, to improve their dependability and reliability. We plan to apply the model and techniques of Phoenix to enhance the
recoverability and reliability of agent interaction and coordination.

- Due to the time constraint of PhD study, Chapter 7 only provides the rules and guidelines to apply our methods in building real applications. As well, the system is only evaluated in a simulated and experimental environment. We plan to collaborate with industry companies to build real and operational applications.
Appendix A

Proof for Commitment Refactoring Rules

Theorem 6.3 has listed all the refactoring rules for commitments in Chapter 6. In this section, we will provide the complete proof for the theorem.

Let us first review the definitions for each type of commitment (Definition 6.2):

\[ CC(\Phi \sqcap \Psi) \equiv \exists i \exists n (i \leq n \land \forall j (i \leq j \leq n \land s_j \not\models \Phi) \land s_n \models \Psi) \]

\[ CC(\Phi \sqcup \Psi) \equiv \exists i \exists n (i < n \land s_i \not\models \Phi \land s_n \models \Psi \land \exists j (i < j \leq n \land \forall k (j \leq k \leq n \land s_k \not\models \neg \Phi))) \]

\[ CC(\Phi || \Psi) \equiv \exists n (s_n \not\models \Phi \land s_n \models \Psi) \]

\[ CC(\Phi \triangleright \Psi) \equiv \exists n (s_n \not\models \neg \Phi \land s_n \models \Psi) \]

The following inference rules for quantifiers will be used in the process of proof:

\[ \exists x (f_x \lor g_x) \iff \exists x f_x \lor \exists x g_x \]

\[ \forall x (f_x \land g_x) \iff \forall x f_x \land \forall x g_x \]

\[ \exists x (f_x \land g_x) \Rightarrow \exists x f_x \land \exists x g_x \]

\[ \forall x f_x \land \forall x g_x \Rightarrow \forall x (f_x \lor g_x) \]
• Distribution over strictly ordered:

1. \( CC(\Phi \cap (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \cap \Psi_1) \land CC(\Phi \cap \Psi_2) \)

From Definition 6.5, the state-formulas \( s_n \models (\Psi_1 \land \Psi_2) \equiv s_n \models \Psi_1 \land s_n \models \Psi_2. \)

Therefore, the commitment-formulas

\[
CC(\Phi \cap (\Psi_1 \land \Psi_2))
\equiv \exists i \exists n (i \leq n \land \forall j(i \leq j \leq n \land s_j \models \Phi) \land (s_n \models \Psi_1 \land s_n \models \Psi_2))
\]

\[
\Rightarrow CC(\Phi \cap \Psi_1) \land CC(\Phi \cap \Psi_2)
\]

2. \( CC(\Phi \cap (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \cap \Psi_1) \lor CC(\Phi \cap \Psi_2) \)

\[
CC(\Phi \cap (\Psi_1 \lor \Psi_2))
\equiv \exists i \exists n (i \leq n \land \forall j(i \leq j \leq n \land s_j \models \Phi) \land (s_n \models \Psi_1 \lor s_n \models \Psi_2))
\]

\[
\Rightarrow CC(\Phi \cap \Psi_1) \lor CC(\Phi \cap \Psi_2)
\]

3. \( CC((\Phi_1 \land \Phi_2) \cap \Psi) \Rightarrow CC(\Phi_1 \cap \Psi) \land CC(\Phi_2 \cap \Psi) \)

\[
CC((\Phi_1 \land \Phi_2) \cap \Psi)
\equiv \exists i \exists n (i \leq n \land \forall j(i \leq j \leq n \land s_j \models \Phi_1 \land s_j \models \Phi_2) \land s_n \models \Psi)
\]

\[
\Rightarrow \exists i \exists n (i \leq n \land \forall j(i \leq j \leq n \land s_j \models \Phi_1)
\land \forall j(i \leq j \leq n \land s_j \models \Phi_2) \land s_n \models \Psi)
\]

\[
\Rightarrow CC(\Phi_1 \cap \Psi) \land CC(\Phi_2 \cap \Psi)
\]
4. \( CC((\Phi_1 \lor \Phi_2) \cap \Psi) \equiv CC(\Phi_1 \cap \Psi) \lor CC(\Phi_2 \cap \Psi) \)

\[
CC(\Phi_1 \cap \Psi) \lor CC(\Phi_2 \cap \Psi)
\]

\[
\Leftrightarrow \exists \forall i (i \leq n \land \forall j(i \leq j \leq n \land s_j \neq \Phi_1) \land s_n \neq \Psi)
\]

\[
\lor \exists \forall i (i \leq n \land \forall j(i \leq j \leq n \land s_j \neq \Phi_2) \land s_n \neq \Psi)
\]

\[
\Leftrightarrow \exists \forall i ((i \leq n \land \forall j(i \leq j \leq n \land s_j \neq \Phi_1) \land s_n \neq \Psi)
\]

\[
\lor (i \leq n \land \forall j(i \leq j \leq n \land s_j \neq \Phi_2) \land s_n \neq \Psi)
\]

\[
\Leftrightarrow \exists \forall i ((i \leq n \land s_n \neq \Psi)
\]

\[
\land (\forall j(i \leq j \leq n \land s_j \neq \Phi_1) \lor \forall j(i \leq j \leq n \land s_j \neq \Phi_2))
\]

\[
\Rightarrow \exists \forall i (i \leq n \land \forall j(i \leq j \leq n \land s_j \neq \Phi_1) \land s_n \neq \Psi)
\]

\[
\Rightarrow CC(((\Phi_1 \lor \Phi_2) \cap \Psi)
\]

• Distribution over weakly ordered:

1. \( CC(\Phi \cup (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \cup \Psi_1) \land CC(\Phi \cup \Psi_2) \)

\[
CC(\Phi \cup (\Psi_1 \land \Psi_2))
\]

\[
\Leftrightarrow \exists \forall i (i < n \land s_i \neq \Phi \land (s_n \neq \Psi_1 \land s_n \neq \Psi_2)
\]

\[
\land \exists j(i < j \leq n \land \forall k(j \leq k \leq n \land s_k \neq \neg \Phi))
\]

\[
\Rightarrow \exists \forall i (i < n \land s_i \neq \Phi \land s_n \neq \Psi_1 \land \exists j(i < j \leq n \land \forall k(j \leq k \leq n \land s_k \neq \neg \Phi))
\]

\[
\land \exists \forall i (i < n \land s_i \neq \Phi \land s_n \neq \Psi_2 \land \exists j(i < j \leq n \land \forall k(j \leq k \leq n \land s_k \neq \neg \Phi))
\]

\[
\Rightarrow CC(\Phi \cup \Psi_1) \land CC(\Phi \cup \Psi_2)
\]

2. \( CC(\Phi \cup (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \cup \Psi_1) \lor CC(\Phi \cup \Psi_2) \)

\[
CC(\Phi \cup (\Psi_1 \lor \Psi_2))
\]

\[
\Leftrightarrow \exists \forall i (i < n \land s_i \neq \Phi \land (s_n \neq \Psi_1 \lor s_n \neq \Psi_2)
\]

\[
\land \exists j(i < j \leq n \land \forall k(j \leq k \leq n \land s_k \neq \neg \Phi))
\]

\[
\Rightarrow \exists \forall i (i < n \land s_i \neq \Phi \land s_n \neq \Psi_1 \land \exists j(i < j \leq n \land \forall k(j \leq k \leq n \land s_k \neq \neg \Phi))
\]

\[
\lor \exists \forall i (i < n \land s_i \neq \Phi \land s_n \neq \Psi_2 \land \exists j(i < j \leq n \land \forall k(j \leq k \leq n \land s_k \neq \neg \Phi))
\]
172 Proof for Commitment Refactoring Rules

3. \( CC((\Phi_1 \land \Phi_2) \sqcup \Psi) \iff CC(\Phi_1 \sqcup \Psi) \lor CC(\Phi_2 \sqcup \Psi) \)

\[
CC((\Phi_1 \land \Phi_2) \sqcup \Psi)
\equiv \exists \exists (i < n \land s_j \equiv \Phi_1 \land s_i \equiv \Phi_2 \land s_n \equiv \Psi \\
\land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg((\Phi_1 \land \Phi_2))))
\equiv \exists \exists (i < n \land s_j \equiv \Phi_1 \land s_i \equiv \Phi_2 \land s_n \equiv \Psi \\
\land (\exists j(i < j \leq n \land (\forall k(j < k \leq n \land s_k \equiv \neg(\Phi_1)) \lor \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_2)))))
\equiv \exists \exists (i < n \land s_j \equiv \Phi_1 \land s_i \equiv \Phi_2 \land s_n \equiv \Psi \\
\land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_1)))
\lor \exists \exists (i < n \land s_j \equiv \Phi_2 \land s_i \equiv \Psi \land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_2))))
\equiv CC(\Phi_1 \sqcup \Psi) \lor CC(\Phi_2 \sqcup \Psi)
\]

4. \( CC((\Phi_1 \lor \Phi_2) \sqcup \Psi) \implies CC(\Phi_1 \sqcup \Psi) \land CC(\Phi_2 \sqcup \Psi) \)

\[
CC((\Phi_1 \lor \Phi_2) \sqcup \Psi)
\equiv \exists \exists (i < n \land s_j \equiv \Phi_1 \lor s_i \equiv \Phi_2 \land s_n \equiv \Psi \\
\land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg((\Phi_1 \lor \Phi_2))))
\equiv \exists \exists (i < n \land s_j \equiv \Phi_1 \lor s_i \equiv \Phi_2 \land s_n \equiv \Psi \\
\land (\exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_1)))
\land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_2))))
\equiv \exists \exists (i < n \land s_j \equiv \Phi_1 \land s_i \equiv \Phi_2 \land s_n \equiv \Psi \\
\land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_1)))
\lor \exists \exists (i < n \land s_j \equiv \Phi_2 \land s_i \equiv \Psi \land \exists j(i < j \leq n \land \forall k(j < k \leq n \land s_k \equiv \neg(\Phi_2))))
\equiv CC(\Phi_1 \sqcup \Psi) \land CC(\Phi_2 \sqcup \Psi)
\]

• Distribution over strictly unordered:

1. \( CC(\Phi \parallel (\Psi_1 \land \Psi_2)) \implies CC(\Phi \parallel \Psi_1) \land CC(\Phi \parallel \Psi_2) \)

\[
CC(\Phi \parallel (\Psi_1 \land \Psi_2))
\]
\[ \Leftrightarrow \exists n (s_n \vDash \Phi \land s_n \vDash \Psi_1 \land s_n \vDash \Psi_2) \]
\[ \Rightarrow \exists n (s_n \vDash \Phi \land s_n \vDash \Psi_1) \land \exists n (s_n \vDash \Phi \land s_n \vDash \Psi_2) \]
\[ \Leftrightarrow CC(\Phi \parallel \Psi_1) \land CC(\Phi \parallel \Psi_2) \]

2. \( CC(\Phi \parallel (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \parallel \Psi_1) \lor CC(\Phi \parallel \Psi_2) \)

\[ CC(\Phi \parallel (\Psi_1 \lor \Psi_2)) \]
\[ \Leftrightarrow \exists n (s_n \vDash \Phi \land (s_n \vDash \Psi_1 \lor s_n \vDash \Psi_2)) \]
\[ \Rightarrow \exists n (s_n \vDash \Phi \land s_n \vDash \Psi_1) \lor \exists n (s_n \vDash \Phi \land s_n \vDash \Psi_2) \]
\[ \Leftrightarrow CC(\Phi \parallel \Psi_1) \lor CC(\Phi \parallel \Psi_2) \]

3. \( CC((\Phi_1 \land \Phi_2) \parallel \Psi) \Rightarrow CC(\Phi_1 \parallel \Psi) \land CC(\Phi_2 \parallel \Psi) \)

\[ CC((\Phi_1 \land \Phi_2) \parallel \Psi) \]
\[ \Leftrightarrow \exists n (s_n \vDash \Phi_1 \land s_n \vDash \Phi_2 \land s_n \vDash \Psi) \]
\[ \Rightarrow \exists n (s_n \vDash \Phi_1 \land s_n \vDash \Psi) \land \exists n (s_n \vDash \Phi_2 \land s_n \vDash \Psi) \]
\[ \Rightarrow CC(\Phi_1 \parallel \Psi) \land CC(\Phi_2 \parallel \Psi) \]

4. \( CC((\Phi_1 \lor \Phi_2) \parallel \Psi) \equiv CC(\Phi_1 \parallel \Psi) \lor CC(\Phi_2 \parallel \Psi) \)

\[ CC((\Phi_1 \lor \Phi_2) \parallel \Psi) \]
\[ \Leftrightarrow \exists n ((s_n \vDash \Phi_1 \lor s_n \vDash \Phi_2) \land s_n \vDash \Psi) \]
\[ \Rightarrow \exists n (s_n \vDash \Phi_1 \land s_n \vDash \Psi) \lor \exists n (s_n \vDash \Phi_2 \land s_n \vDash \Psi) \]
\[ \Leftrightarrow CC(\Phi_1 \parallel \Psi) \lor CC(\Phi_2 \parallel \Psi) \]

- Distribution over weakly unordered:

In Theorem 6.2, \( CC(\Phi \diamond \Psi) \equiv CC(\neg \Phi \parallel \Psi) \)

1. \( CC(\Phi \diamond (\Psi_1 \land \Psi_2)) \Rightarrow CC(\Phi \diamond \Psi_1) \land CC(\Phi \diamond \Psi_2) \)

\[ CC(\Phi \diamond (\Psi_1 \land \Psi_2)) \]
\(\Leftrightarrow CC(\neg \Phi \parallel (\Psi_1 \land \Psi_2))\)

\(\Rightarrow CC(\neg \Phi \parallel \Psi_1) \land CC(\neg \Phi \parallel \Psi_2)\)

\(\Leftrightarrow CC(\Phi \diamond \Psi_1) \land CC(\Phi \diamond \Psi_2)\)

2. \(CC(\Phi \diamond (\Psi_1 \lor \Psi_2)) \equiv CC(\Phi \diamond \Psi_1) \lor CC(\Phi \diamond \Psi_2)\)

\(CC(\Phi \diamond (\Psi_1 \lor \Psi_2))\)

\(\Leftrightarrow CC(\neg \Phi \parallel (\Psi_1 \lor \Psi_2))\)

\(\Leftrightarrow CC(\neg \Phi \parallel \Psi_1) \lor CC(\neg \Phi \parallel \Psi_2)\)

\(\Leftrightarrow CC(\Phi \diamond \Psi_1) \lor CC(\Phi \diamond \Psi_2)\)

3. \(CC((\Phi_1 \land \Phi_2) \diamond \Psi) \Leftrightarrow CC(\Phi_1 \diamond \Psi) \lor CC(\Phi_2 \diamond \Psi)\)

\(CC((\Phi_1 \land \Phi_2) \diamond \Psi)\)

\(\Leftrightarrow CC(\neg (\Phi_1 \land \Phi_2) \parallel \Psi)\)

\(\Leftrightarrow CC((\neg \Phi_1 \lor \neg \Phi_2) \parallel \Psi)\)

\(\Leftrightarrow CC(\neg \Phi_1 \parallel \Psi) \lor CC(\neg \Phi_2 \parallel \Psi)\)

\(\Leftrightarrow CC(\Phi_1 \diamond \Psi) \lor CC(\Phi_2 \diamond \Psi)\)

4. \(CC((\Phi_1 \lor \Phi_2) \diamond \Psi) \Rightarrow CC(\Phi_1 \diamond \Psi) \land CC(\Phi_2 \diamond \Psi)\)

\(CC((\Phi_1 \lor \Phi_2) \diamond \Psi)\)

\(\Leftrightarrow CC(\neg (\Phi_1 \lor \Phi_2) \parallel \Psi)\)

\(\Leftrightarrow CC((\neg \Phi_1 \land \neg \Phi_2) \parallel \Psi)\)

\(\Rightarrow CC(\neg \Phi_1 \parallel \Psi) \land CC(\neg \Phi_2 \parallel \Psi)\)

\(\Leftrightarrow CC(\Phi_1 \diamond \Psi) \land CC(\Phi_2 \diamond \Psi)\)
Appendix B

Abbreviations and Glossary of terms

B.1 Abbreviations

[ ACID ] Atomicity, Consistency, Isolation and Durability

[ API ] Application Programming Interface

[ APL ] Agent Programming Language

[ ARTS ] Agent-oriented Robust Transactional System

[ ATM ] Advanced Transaction Model

[ BDI ] Belief, Desire, Intention

[ BPM ] Business Process Management

[ DAG ] Directed Acyclic Graph

[ EBNF ] Extended Backus-Naur Form

[ ECA ] Event, Condition, Action

[ EH ] Exception Handling

[ GA ] Genetic Algorithm

[ GPL ] GNU General Public License
**B.2 Glossary of Terms**

[**Exception**] An unexpected occurrences that are not accounted for in a system’s normal operation.

[**Failure**] The delivered service deviates from the system specification.

[**Consistency**] A state is consistent if all system invariants are satisfied.

[**Failure Atomicity**] A task either finishes successfully or terminates in an acceptable consistent state in case of failure.

[**Plan**] A composition of a group of tasks to react to an event.

[**Choice Point**] For a certain goal of the system, if there are more than one eligible plan, the corresponding node in the execution tree is marked as a choice point.
[**Choice Point Stack**] A choice point stack contains the choice points the system has met chronologically.

[**Multi-Step Backtracking**] In case of exception, the execution flow of the system continues at another selectable plan implied by elements of its choice point stack.

[**Critical Path**] A sequence of tasks which add up to the longest overall duration for the completion of the task network.

[**Slack Time**] The maximum amount of time that a task is allowed to be delayed without affecting the whole schedule for a task network.

[**Agent Interaction Protocol**] A specification which defines a global state machine incorporating the allowed messages and their occurrence order among agents.

[**Commitment**] An abstraction to the obligatory and contractual relationship between agents.

[**Commitment Refactoring**] The composition or decomposition of an agent’s commitments, resulting in improved performance or simplified structure without changing its external behaviour.
Bibliography


Index

agent, 1–3, 6, 94
  BDI, 6
    data structure, 6
    event driven, 7
    execution principles, 69
    means-end reasoning, 7
    execution capability, 97
  feature, 6
  interaction protocol, 127
  proactive, 31
  programming language, 8, 20
  reactive, 31
ARTS, 33, 36
  application, 153
  dependency set, 51
    dependency path, 55
  event, 47
  exception handling, 63
plan, 48
  execution rules, 59
    inheritance, 50
procedure, 49
  inheritance, 50
program, 51
scheduling, 93, 121
task composition, 48
task model, 37
  atomic task, 38, 41
  compensation task, 42
  composite task, 43, 44
  reconciliation, 46
  task inheritance, 43
autonomic computing, 2
autonomous agent, 3, 4, 44, 96, 122
autonomous system, 3
autonomy, 1, 3, 93, 94
BPM system, 154
  challenge, 154
    multi-step backtracking, 155
  trend, 154
business process, 153
  commitment, 122, 124, 149
    conditional, 124
  dependency, 141
    guarded, 138
  inference rules, 133
  recoverable, 138
  refactoring, 136
  semantics, 132
sequence, 141
strictly ordered, 127
strictly unordered, 128
syntax
  commitment-formula, 132
  state-formula, 131
unconditional, 124
unordered commitment dependency, 143
unordered commitment sequence, 143
weakly ordered, 127
weakly unordered, 128
commitment protocol, 124
commitment scheduling, 139
  commitment safety indicator, 146
  safety level, 138
coordination, 4
exception, 4, 17
exception handling, 18
  backtracking, 64, 67, 77
  multi-step backtracking, see multi-step backtracking
  replanning, 64
failure, 18
fault tolerant, 8
  redundancy, 9
  transaction, 9
genetic algorithm, 107
interaction protocol, 122
Internet computing, 2–5, 11, 164
leveled-commitment contracting, 150
multi-agent system, 2, 9, 18, 94, 122, 155, 164
multi-step backtracking, 78
  choice point, 66
  choice point stack, 71
  jump, 72
  algorithm, 74
  transaction management, 81
  transactional execution, 79
plan management
  coordinator, 97
  DAG, 97
execution monitoring
  agent substitution, 111
  rules, 109
  genetic plan scheduling, 107
  chromosome, 108
  evaluation function, 108
plan evaluation, 100
  allocated slack time, 105
  plan utility function, 106
  task estimation function, 103
plan selection, 95
planning, 4
reliability, 1
robust multi-agent system
  exception avoidance, 20
  exception recovery
  Citizens, 29
commitment protocols, 26
goal-based semantic compensation, 23
Guardian, 28
Phoenix, 24
SaGE, 25
stateless, 21
transactional, 21
taxonomy, 18, 30
robustness, 1, 8
scheduling, 4, 94
  concurrent scheduling, 150
  probability of success, 99
  schedule robustness, 99
slack time, 103
task
  cost, 98
  execution
    aborted state, 39
    compensated state, 39
    consistent state, 38
    failure atomicity, 40
    successful state, 39
group
  cost of error, 144
sequence
  cost of error, 142
time, 97
trust, 98
transaction, 9, 78

ACID, 9, 36, 163
advanced transaction model, 10, 163
cascade rollback, 10
compensating transactions, 10
long-running transaction, 10
open nested transaction, 78, 79
trust, 11, 94–96
model, 11
trustworthy, 94
workflow, 157, 163
  composition, 159
  scheduling, 157
  dynamic, 159
  static, 157
  scientific, 157