Hermes: Goal-Oriented Interactions for Intelligent Agents

A thesis submitted for the degree of
Doctor of Philosophy

Dean Christopher Ho Mok Cheong
B.App.Sc. Computer Science,
B.App.Sc. Computer Science (Honours)

School of Computer Science and Information Technology,
Science, Engineering, and Technology Portfolio,
RMIT University,
Melbourne, Victoria, Australia.

September 2008
Declaration

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

Dean Christopher Ho Mok Cheong
School of Computer Science and Information Technology
RMIT University
September 2008
Acknowledgments

I was very fortunate to be under the supervision of Associate Professor Michael Winikoff and Professor Lin Padgham during my PhD candidature. I am immensely grateful for their help, guidance, and support. They have been instrumental in my journey of becoming a researcher and an academic by the standards they set and the way they conduct themselves. They embody the advice they give to their students and I could not have asked for better supervisors.

I must also thank my family for their never-ending support. My dad, France, who has always led by example. My mum, Roseline, who is always supportive and encouraging. My two younger brothers, Jason and Warren, always keeping me grounded in the real world and providing me with welcomed distractions from the PhD.

The RMIT Intelligent Agents Group must also be thanked. I have benefited from each member’s experience and advice.

I’d also like to express my gratitude to Simon Duff, David Poutakidis, and Uy Dung (Dennis) Nguyen, fellow PhD candidates from the Intelligent Agents Group, who have made my candidature an even more enjoyable experience. We have shared some great (and dangerous!) times in the “PhD room”.

I must especially thank Simon Duff. We have been through some excellent times and fun adventures together since our Honours year.

Special thanks must also be made to Jason Khallouf and Kevin Leung.

I am also grateful to RMIT University and the School of Computer Science and Information Technology for financial support through an Australian Postgraduate Award.

I must also thank Professor Brian Corbitt, Head of School, School of Business Information Technology, RMIT University, for allowing me a semester of teaching leave in order to complete and finalize the last stages of my thesis.
Dedication

To my grandfather, Marcel Ho Mok Cheong (19/11/1919 – 24/08/2000). A loving man who had the insight to give his son an education. An education which his son passed down to his sons. An education which I in turn hope to pass on. You may no longer be amongst us, but your legacy is.
Credits

Portions of the material in this thesis have previously appeared in the following publications:


Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.
# Contents

Abstract 1

1 Introduction 3
  1.1 Intelligent Agents and their Interactions 3
  1.2 Aims and Scope of Thesis 7
  1.3 Contributions 8
  1.4 Thesis Overview 8

2 Background 10
  2.1 Agents and Agent Models 10
    2.1.1 Reactive Agents 12
      Reactive Agents and Social Ability 13
    2.1.2 Reasoning Agents 13
      BDI Agents 14
      Reasoning Agents and Social Ability 17
    2.1.3 Hybrid Agents 17
      Hybrid Agents and Social Ability 19
  2.2 Social Ability 19
    2.2.1 Agent Communication Languages 20
    2.2.2 The Speech Act Theory 22
    2.2.3 Interacting to Reach Agreements 23
    2.2.4 Interacting to Work Together 26
    2.2.5 Agent Societies 29
  2.3 Interaction Design 30
    2.3.1 Traditional Message-Centric Approaches to Agent Interactions 31
    2.3.2 Alternative Approaches to Traditional Agent Interactions 33
      Commitment Machines 34
      Social Commitments 38
      Landmark-Based Approach 39
## CONTENTS

4.3 Integrated Process ............................................. 108  
4.4 Interaction Goal Hierarchy ........................................ 111  
4.5 Action Maps .......................................................... 114  
  4.5.1 Develop Initial Action Maps ................................. 114  
  4.5.2 Adding Data to Action Maps ................................. 116  
  4.5.3 Generalizing Action Maps ................................... 117  
  4.5.4 Adding Failure Handling to Action Maps .................. 118  
4.6 Software Tool Support .............................................. 120  
4.7 Summary ............................................................... 123  

5 Implementing Goal-Oriented Interactions .......................... 124  
  5.1 Interaction Goal State Representation and Beliefs ............ 126  
  5.2 Interface Plans ..................................................... 128  
  5.3 Coordination Plans ................................................ 129  
  5.4 Achievement Plans ............................................... 132  
  5.5 Sample Execution .................................................. 135  
  5.6 Implementing Parallelism ........................................ 139  
    5.6.1 Implementing Parallel Interaction Goals ................. 139  
    5.6.2 Implementing Parallel Actions .............................. 140  
    5.6.3 Implementing Homogeneous Instance Parallelism ........ 144  
  5.7 Implementation Software Support Tool ........................ 145  
    5.7.1 Syntactic Representation of Hermes Goal-Oriented Interaction 145  
    5.7.2 Skeleton Code Generation Tool ............................. 148  
  5.8 Summary ............................................................. 151  

6 Evaluation ............................................................. 153  
  6.1 Evaluation Objectives and Metrics .............................. 153  
  6.2 Experimental Procedure ......................................... 155  
  6.3 Results .............................................................. 157  
    6.3.1 Post-Evaluation Questionnaire Analysis .................... 158  
    6.3.2 Scenario Coverage .......................................... 162  
    6.3.3 Flexibility .................................................... 162  
    6.3.4 Robustness .................................................... 166  
    6.3.5 Design Time .................................................. 168  
  6.4 Summary ............................................................. 168  

7 Conclusion and Future Work ......................................... 170  
  7.1 Research Aims and Contributions Revisited .................... 172  
  7.2 Limitations and Future Work .................................... 172
List of Figures

1.1 Prometheus Protocol Example .............................................. 5
2.1 Goal-Plan Tree ............................................................. 16
2.2 Layered Hybrid Architectures ............................................. 18
2.3 The Request and Inform Speech Acts ................................ 23
2.4 AUML Protocol diagram for Negotiate Details Interaction ........ 32
2.5 NetBill Protocol Interaction Diagram with Commitments ......... 36
2.6 Finite State Machine Representation of Commitment Machine for the NetBill Protocol 37
2.7 Partially Ordered Landmarks ............................................. 39
2.8 Prometheus Overview Diagram ........................................... 46
2.9 A Prometheus Analysis Overview Diagram ......................... 47
2.10 Negotiate Price Scenario ................................................ 48
2.11 A Prometheus Goal Overview Diagram ............................ 49
2.12 Negotiate Price Interaction Diagram ............................... 50
2.13 Negotiate Price Interaction Protocol ............................... 52
2.14 A System Overview Diagram ........................................... 53
2.15 Process Diagram for Vendor Agent in the Negotiate Price Interaction Protocol ... 54
2.16 Plan Selection Process ................................................... 56
3.1 Simplified NetBill Protocol ............................................... 61
3.2 Hermes Methodology Overview Diagram ............................ 62
3.3 Intermediate Interaction Goal Hierarchy ............................ 66
3.4 Final Interaction Goal Hierarchy ......................................... 67
3.5 Initial Action Map ......................................................... 71
3.6 Action Map with Data Stores ............................................ 75
3.7 Action Map with Generalizations ....................................... 77
3.8 Modified Interaction Goal Hierarchy .................................. 78
3.9 Action Map with an Action Retry ...................................... 81
3.10 Action Map with Rollback .............................................. 83
LIST OF FIGURES

3.11 Action Sequence Diagram ........................................... 86
3.12 Action Message Diagram ............................................ 89
3.13 Interaction Parallelism Taxonomy .................................. 90
3.14 Interaction Goal Hierarchy with Parallel Interaction Goals .... 92
3.15 Interaction Goal Hierarchy with a Mixture of Sequential and Parallel Interaction Goals ............................................. 93
3.16 Action Map with Parallel Actions .................................. 95
3.17 Action Map with Homogeneous Instance Parallelism .......... 98
3.18 UMLet: Interaction Goal Hierarchy Development .............. 100
3.19 UMLet: Action Map Development .................................. 101

4.1 Order Book Scenario .................................................. 105
4.2 Prometheus Overview Diagram ...................................... 107
4.3 Integrated Overview Diagram ....................................... 108
4.4 Integrated Design Steps .............................................. 109
4.5 System Overview Diagram .......................................... 111
4.6 Interaction Goal Hierarchy ........................................... 113
4.7 Initial Action Map without Data Stores: Organize Delivery .... 116
4.8 Initial Action Map with Data Stores: Organize Delivery ...... 117
4.9 Action Map: Organize Delivery (with scenario variation) .... 119
4.10 Final Action Map: Organize Delivery .............................. 121

5.1 Implementation Overview ........................................... 125
5.2 Interaction Goal States and Valid Transitions ................... 127
5.3 Rollback Sequence .................................................... 136
5.4 Sample Execution ..................................................... 138
5.5 Parallel Interaction Goals .......................................... 140
5.6 Parallel Actions ..................................................... 141
5.7 Merging Successful Parallel Action Sequences ................. 143
5.8 Merging Failed Parallel Action Sequences ....................... 144
5.9 Code Generation Tool Overview ................................... 149

6.1 Designed Interaction is Easy to Follow ......................... 159
6.2 Participant 4’s Protocol Diagram (Partial) ....................... 161
6.3 Graphs of Flexibility Values ....................................... 165

C.1 Schedule Meeting Scenario (Steps 1–7) .......................... 186
C.2 Schedule Meeting Scenario (Steps 8–14) ......................... 187

H.1 Participant 9’s Interaction Goal Hierarchy ....................... 203
H.2 Participant 9’s Action Map .......................... 204
H.3 Participant 9’s Modified Action Map .............. 205
I.1 Prometheus Protocol Diagram ........................... 208
I.2 Prometheus Interaction Message Paths ............... 209
I.3 Hermes Action Map ..................................... 210
I.4 Hermes Interaction Message Paths ................. 210
I.5 Participant 14’s Protocol Diagram ...................... 212
I.6 Participant 14’s Interaction Message Paths .......... 212
I.7 Participant 14’s Compressed Interaction Message Paths 213
J.1 Overall Ease of Use ................................ 214
J.2 Designed Interaction is Flexible ...................... 215
J.3 Designed Interaction is Robust ...................... 215
J.4 Designed Interaction is Simple ..................... 216
J.5 Designed Interaction is Easy to Understand .......... 216
J.6 Designed Interaction was Easy to Design .......... 217
J.7 Designed Interaction was Fast to Design .......... 217
J.8 Designed Interaction is Easy to Follow ............ 218
J.9 Design Process is Well Guided ...................... 218
J.10 Design Process is Easy to Use ..................... 219
J.11 Design Process is Fast ................................ 219
J.12 Design Process is Simple ............................ 220
J.13 Design Process Does Not Require a Lot of Effort 220
J.14 Design Process is Easy to Learn ................ 221
J.15 Design Process is Conducive to Thought Process 221
J.16 Design Process is Expressive ........................ 222
J.17 Model/Diagrams Used in Design Process Intuitive 222
K.1 Overall Ease of Use ................................ 223
K.2 Designed Interaction is Flexible ...................... 224
K.3 Designed Interaction is Robust ...................... 224
K.4 Designed Interaction is Simple ..................... 225
K.5 Designed Interaction is Easy to Understand .......... 225
K.6 Designed Interaction was Easy to Design .......... 226
K.7 Designed Interaction was Fast to Design .......... 226
K.8 Designed Interaction is Easy to Follow ............ 227
K.9 Design Process is Well Guided ...................... 227
K.10 Design Process is Easy to Use .................... 228
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>K.11</td>
<td>Design Process is Fast</td>
<td>228</td>
</tr>
<tr>
<td>K.12</td>
<td>Design Process is Simple</td>
<td>229</td>
</tr>
<tr>
<td>K.13</td>
<td>Design Process Does not Require a Lot of Effort</td>
<td>229</td>
</tr>
<tr>
<td>K.14</td>
<td>Design Process is Easy to Learn</td>
<td>230</td>
</tr>
<tr>
<td>K.15</td>
<td>Design Process is Conducive to Thought Process</td>
<td>230</td>
</tr>
<tr>
<td>K.16</td>
<td>Design Process is Expressive</td>
<td>231</td>
</tr>
<tr>
<td>K.17</td>
<td>Model/Diagrams Used in Design Process Intuitive</td>
<td>231</td>
</tr>
</tbody>
</table>
List of Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Generic (Sequential) Compound Coordination Plan for Interaction Goal IG</td>
<td>130</td>
</tr>
<tr>
<td>5.2</td>
<td>Generic Atomic Coordination Plan for Interaction Goal IG</td>
<td>131</td>
</tr>
<tr>
<td>5.3</td>
<td>Generic Achievement Plan</td>
<td>132</td>
</tr>
<tr>
<td>5.4</td>
<td>Customer Rollback Plan (from Negotiate Price to Negotiate Details)</td>
<td>134</td>
</tr>
<tr>
<td>5.5</td>
<td>Generic (Parallel) Compound Coordination Plan</td>
<td>142</td>
</tr>
<tr>
<td>5.6</td>
<td>Intermediate Achievement Plan: Process 01 Success</td>
<td>143</td>
</tr>
</tbody>
</table>
List of Listings

2.1 Abstract BDI Interpreter ........................................ 14
2.2 A KQML Message .................................................. 21
2.3 A KIF Message ...................................................... 21
2.4 A FIPA ACL Message with FIPA SL used as the Content Language .......................... 22
2.5 Jadex Agent Definition File ....................................... 57
2.6 Jadex Plan Body ...................................................... 58
5.1 Hermes Interaction Grammar ..................................... 146
5.2 Sample Hermes Interaction ........................................ 148
A.1 Generated Compound Coordination Plan .............................. 175
A.2 Generated Atomic Coordination Plan ................................ 175
A.3 Generated Achievement Plan ...................................... 176
B.1 XML Schema for Syntactic Representation .......................... 178
B.2 XML Syntactic Representation, Part 1: Interaction Goal Hierarchy .................. 181
B.3 XML Syntactic Representation, Part 2: Action Maps ...................... 182
## List of Tables

3.1 Possible Failures and Remedial Actions for *Negotiate Price* .................. 80
3.2 Message Descriptor ................................................................................. 88

4.1 Mapping of Scenario Steps to Interaction Goals ........................................ 113
4.2 Possible Failures and Remedial Actions for *Organize Delivery* ................. 119

5.1 Belief Structure and Examples ................................................................. 128

6.1 Summary of Metrics .................................................................................. 155
6.2 Experimental Groups and Bands ............................................................... 156
6.3 Flexibility and Robustness Scales from Post-Evaluation Questionnaire .......... 158
6.4 Scenario Coverage Results ....................................................................... 163
6.5 Flexibility Results ...................................................................................... 164
6.6 Possible Variations ..................................................................................... 167
6.7 Design Time Results .................................................................................. 168

G.1 P-Values for Questionnaire Responses about Participants’ Perceptions of the “overall ease of use” of the Methodologies Employed ................................. 201
G.2 P-Values for Questionnaire Responses about Participants’ Perceptions of their Created Design ................................................................. 201
G.3 P-Values for Questionnaire Responses about Participants’ Perceptions of the Methodology Used ..................................................................................... 202

H.1 Scenario Coverage for Participant 9’s Design ............................................. 206
H.2 Robustness Results for Participant 9’s Design .............................................. 206
H.3 Flexibility Results for Participant 9’s Design ............................................... 206
H.4 P-values with and without Participant 9’s Design ......................................... 207
Abstract

Intelligent agents are goal-oriented software entities which exhibit a number of desirable characteristics, such as flexibility and robustness, which are suitable for complex, dynamic, and failure-prone environments. However, these characteristics of individual agents are not exhibited by their interactions with each other since traditional approaches to interaction design are message-centric, and these message-centric approaches force the intelligent agents to follow prescribed message sequences in order to achieve their interactions, thus usually resulting in interactions which have limited flexibility and robustness.

In this thesis an alternative to the traditional message-centric interaction design approaches is presented. In this approach, the interactions are designed based on interaction goals, and message sequences are not prescribed. Instead, message sequences emerge from the interactions as the intelligent agents attempt to achieve the interaction goals.

The main contribution of this work is Hermes, a methodology for the design and implementation of goal-oriented interactions. An important motivation for Hermes is to not only allow for the design and implementation of goal-oriented interactions, but to also be pragmatic and usable by practicing software engineers. To that end, Hermes has a clear and guided design process with a notation explicitly created for the design of goal-oriented interactions. Furthermore, Hermes, which covers the design and implementation of agent interactions only, has been integrated with Prometheus, a full agent system design methodology. Guidelines for the integration are provided so that, in future, Hermes may also be integrated with other existing methodologies if desired.

Hermes also provides guidelines for mapping its design artifacts to an implementation. As Hermes is goal-oriented, the implementation platform should be one that is goal-based. The guidelines help developers map the design to skeleton code. This contributes to the pragmatism of Hermes.

To further ensure that Hermes is pragmatic, two prototype software support tools have been developed. The design support tool allows for the graphical design of Hermes artifacts and the implementation support tool produces skeleton code for the Jadex agent platform based on a structured textual representation of Hermes design artifacts. Although only the Jadex agent platform is currently supported, the implementation tool can be extended to accommodate other goal-based agent platforms.
An empirical evaluation was carried out, and its results show that Hermes designs are significantly more flexible and robust than message-centric designs, although more time is required to design Hermes interactions. This suggests that Hermes is suitable for interactions which are complex and/or error-prone.
Chapter 1

Introduction

1.1 Intelligent Agents and their Interactions

Our ever-evolving and technologically advanced world is a place that is complex, dynamic, and failure-prone. Thus, for many real world applications to be practical, they must be able to meet the challenges of our environment. Intelligent agents are becoming known as a technology which is able to address the aforementioned real world issues [Jennings, 2001]. As further research and development of this technology continues, it is likely that it will be more suitable as a practical approach and technology to address real world issues.

Intelligent agents possess a number of characteristics, however, key characteristics that are commonly agreed upon are autonomy, reactivity, proactivity (i.e. goal-orientation), situatedness and social ability. Consider the example of an online Vendor agent. The Vendor is situated on a particular web server and is able to operate autonomously (i.e. without human intervention) given a number of goals. For example, the Vendor agent might be given goals of selling its wares, keeping its inventory stocked at a designated level and removing unpopular products from its inventory. If the Vendor’s product stock reaches a defined low level, it reacts by contacting a relevant Warehouse agent to re-stock its inventory. If a product proves to be unpopular, the Vendor can proactively advertise a sale to clear those particular products in its inventory and desist from selling or re-stocking that product (unless it proves to be popular at a later stage). The Vendor’s social ability emerges from its interaction with other agents, such as the Warehouse agent, customer agents and other vendor agents.

Other important concepts of intelligent agents are flexibility and robustness. An agent’s flexibility is determined by the number of ways it can achieve a goal. For example, assume the Vendor agent has an Accept Payment goal. If the Vendor is only able to accept payment from a particular type of credit card, it is a very rigid agent. However, if the Vendor is able to accept a variety of payment methods, such as numerous types of credit cards, debit cards and other electronic fund transfers (e.g. PayPal), the Vendor is more flexible, practical and useful.
Robustness is an agent’s resistance to failure and is related to flexibility. For example, if the Vendor agent is trying to process a customer’s credit card, but the card is declined, it does not simply terminate the attempt in failure. It will seek alternative ways to remedy this failure, such as enquiring about other credit cards or methods of payment available to the customer.

The combination of these characteristics and fundamental concepts allow intelligent agents to be a suitable solution for environments which are complex, dynamic and failure-prone. Currently, intelligent agents are used in a range of applications spanning a number of different domains. These include telecommunication systems [Chaib-draa, 1995; Jennings, 2001], process control [Sycara, 1998; Jennings et al., 1998], air traffic control [Sycara, 1998], business process management [Jennings, 2001], logistics [Benfield et al., 2006], production scheduling [Munroe et al., 2006], and many more.

The systems in the previous examples all employ multiple agents as “there is no such thing as a single agent system” [Wooldridge, 2002]. In such multi-agent systems, agent interactions are the crux of the matter, as the agents will need to interact in various ways in order to achieve their goals. The types of interactions agents engage in include communication, coordination, cooperation, and even competition and teamwork. The flexibility and robustness of agents play an important part in their interactions, as the interactions can be complex, dynamic and failure-prone.

Furthermore, interactions can allow agents to achieve goals which may require more than one agent to achieve. For example, from the previous Vendor agent description, the Vendor will obviously require the assistance of the customer agents in order to achieve its Accept Payment goal. This particular goal requires the two agents to be able to interact and determine how payment can be made. If a particular customer agent is not able to pay with credit cards, the interaction should be flexible enough to allow the customer agent to pay by other means, e.g. electronic funds transfer. Thus, flexibility and robustness in interactions can be a decisive factor in whether an agent successfully achieves its goals or not.

One of the advantages of using intelligent agents is their flexibility and robustness. Since interactions play such a central role in multi-agent systems, they must be crafted to exploit these characteristics, along with the autonomy and proactivity of intelligent actions. Ideally, it is desirable for agents to impart their inherent flexibility, robustness, autonomy and proactivity upon the interactions in which they partake.

However, current approaches to interaction design are message-centric. The design process is driven by messages that are exchanged during the interaction and is focused on the information passed within the messages. For example, in the Prometheus methodology [Padgham and Winikoff, 2004], as part of its interaction design process, the designers are advised to think about messages and alternatives. This is not restricted to Prometheus, but is also the norm in other methodologies such as Gaia [Jennings et al., 2004], MaSE [DeLoach et al., 2001] and Tropos [Bresciani et al., 2004].

In Prometheus, the interaction protocols are specified using the Agent UML (AUML) [Huget
et al., 2003] notation, an example of which is shown in Figure 1.1. The diagram depicts a protocol in which a Customer agent is attempting to purchase a laptop from a Vendor agent. As can be seen, the protocol is strongly focused on the messages passed between the agents.

![Prometheus Protocol Example](image)

**Figure 1.1: Prometheus Protocol Example**

The interaction begins with the Customer agent inquiring about the price of a particular laptop. When the Vendor agent receives such a message, it reacts by replying with a message containing the price of the laptop. Once the Customer agent receives the price, it places an order.

When an order is placed by a Customer agent, the protocol allows for two different possibilities. If the laptop is in stock, the Vendor requests for the Customer agent’s credit card details and the Customer complies. If the laptop is not in stock, the Customer is notified. If the credit card is declined or its details are incorrect, the interaction is terminated.
Using current message-centric approaches to create interactions results in a number of problems. The main problem is that designs resulting from message-centric approaches tend to be overly, and sometimes unnecessarily, constrained. The interaction in Figure 1.1 must begin with the Customer agent enquiring about the price of a laptop. It cannot, for example, enquire about the availability of a laptop first.

Furthermore, these restricted designs usually inhibit agent proactivity. If the laptop is out of stock, the Vendor could proactively send an appropriate “Laptop Out of Stock” message (i.e. it was not explicitly asked for the availability of the laptop) to the Customer agent before or after replying with the price. This would avoid unnecessarily progressing through the interaction only for the Customer agent to discover that the laptop is out of stock at the end of the interaction. However, this particular message-centric design does not allow for this and, thus, limits the Vendor agent’s proactivity.

Albeit designers can, and usually will, include alternative message sequences for envisioned alternatives, including failures, they can only add a limited number of alternatives that will usually be a small subset of all possible alternatives. For example, in the (straightforward and simple) interaction shown in Figure 1.1, the interaction designer considered a few failure points (invalid credit card details, credit card being declined, availability of the laptop) and provided some simple alternatives. Of course, the designer could have considered more failure points and provided more substantial alternatives. However, if the designer attempted to cover many more (let alone all!) alternatives and failures, this would result in a very complex message-centric protocol that is hard to specify and use.

The difficulty with message-centric approaches is that they tend to result in designs in which flexibility and robustness are limited. As the designs themselves are limited, when they are implemented strictly on agent platforms then the limitations are inherited\(^1\). If such limited designs are followed strictly during implementation, they result in interactions which may be unnecessarily constrained and, thus, contain only a limited amount of flexibility and robustness.

This lack of flexibility and robustness in interactions is problematic for intelligent agents. By following such limited designs, key intelligent agent characteristics, such as autonomy and proactivity, are greatly subdued and the fundamental concept of goal-orientation is being ignored. Thus, current approaches to interaction design are not congruent with the agent paradigm.

The problem with message-centric approaches is a combination of the processes used to design the interactions and the notations used to represent them. The general design process is such that the designer begins by creating a desirable but rigid message sequence and then “loosening” (i.e. improving flexibility and robustness) the protocol by adding alternatives. Naturally, the degree of flexibility and robustness of the interaction will depend on how many alternatives the designer

\(^{1}\)The natural mapping of the design to goal-plan agent platforms (e.g. BDI agent platforms) does not force the protocol to be followed. This can result in greater flexibility but can also produce incorrect behaviour. For example, important messages in the protocol may be skipped.
allows for, which is usually a limited amount. In theory, it is possible to add an unlimited number of alternatives, however, in practice, adding many alternatives using message-centric notations results in protocols that are difficult to understand and manage.

As such, a number of alternative approaches, one of which is developed in this thesis, have emerged. These alternative approaches include those based on social commitments, in which agents progress through interactions by making and fulfilling commitments to each other, a landmark-based approach, in which agents progress through partially ordered “landmarks” which represent different states of affairs, a goal-plan approach, in which interactions are realized through agent plans and goals, and there are also a number of other related work such as electronic institutions. These approaches are described in more detail in Section 2.3.

In general, these alternative approaches lead to more flexible and robust interactions as their design processes begin from the opposite end of the spectrum. The designers start with completely unconstrained interactions and then add constraints so that the protocols are restricted and lead only to desirable interactions. However, these alternatives are not without their failings: many are not pragmatic. In this thesis, a pragmatic alternative approach to message-centric designs which is congruent with the agent paradigm and results in greater flexibility and robustness in agent interactions is presented.

1.2 Aims and Scope of Thesis

Given the mismatch between intelligent agents and current interaction design approaches, this thesis aims to:

- develop a more pragmatic approach to specifying and designing flexible and robust agent interactions; and
- develop guidelines to implement designs developed using the new design methodology on any goal-based agent platform in a way that gives the exact behaviours specified by the design.

The main aim of this work is to develop an agent interaction design methodology that is congruent with the intelligent agent paradigm and pragmatic for practicing software engineers. The methodology is intentionally limited to the design and implementation of agent interactions and not for entire agent systems as there are already many good methodologies for such in existence. Instead of competing with those methodologies, this work demonstrates how the newly developed interaction design methodology can be integrated or incorporated into existing agent system design methodologies. The focus of the work is on design interactions and implementation, not on designing protocols for standards.

The work, as currently presented in this thesis, is intended for closed agent systems\(^2\) and it is assumed that either a single designer is in charge of designing the complete interaction, or,
if there are multiple designers involved, they will communicate and coordinate with each other (the language in the thesis discusses a single designer but it is intended to also cover the case of multiple communicating designers). However, during the development of this methodology, the applicability of this work to open systems was kept in mind. Although not specifically designed for such systems, the methodology should be able to be adapted to work with open systems.

1.3 Contributions

The main contributions of this work are:

- The concept of goal-oriented agent interactions, which is congruent with, and inspired by, the BDI model of agency.
- The Hermes methodology, which allows for the design and implementation of goal-oriented agent interactions.

As part of the main contributions, the following contributions are also made:

- A process for designing goal-oriented interactions.
- A design notation for representing goal-oriented interactions.
- Guidelines on how to integrate Hermes with full agent system design methodologies, including an example of how Hermes is integrated with the Prometheus methodology.
- Guidelines on how to implement Hermes design artifacts on any goal-based agent platform.
- A syntactic representation of goal-oriented interactions.
- Software tool support, including a design tool and a tool to automatically generate skeleton code for the Jadex agent platform.
- An experimental evaluation of the Hermes design methodology.

1.4 Thesis Overview

The remainder of this thesis is structured as follows. Chapter 2 provides a survey of the literature and the background material required to appreciate the work carried out in this thesis. Chapter 3 explains the design process, including notation and failure recovery mechanisms. A case study explaining the integration of Hermes with Prometheus, a full agent system design methodology, is presented in Chapter 4. The implementation process, along with guidelines, syntactic representation of Hermes interactions and the automatic skeleton code generator for the Jadex agent.

---

3 systems in which the components are not explicitly designed to interoperate but are able to do so by adhering to published standards.
platform, is presented in Chapter 5. The empirical evaluation of the work, including metrics that were developed and used, details on how Hermes was evaluated, and a comparison of Hermes and Prometheus (both process and notation), is in Chapter 6. Finally, Chapter 7 contains conclusions and future work.
Chapter 2

Background †

This chapter reviews existing literature and presents background material that is required to appreciate and understand the work carried out in this thesis.

In Section 2.1, agent concepts and models are presented. Details on agent social ability are presented in Section 2.2. This includes discussions on agent communication languages, the speech act theory, agent interaction patterns, agents working together, and societal design. The design of intelligent agent interactions using the traditional approach, as well as alternative approaches, is presented in Section 2.3. In Section 2.4 agent-oriented software engineering methodologies for full agent systems, not just agent interactions, are described. This also includes an explanation of the Prometheus methodology, with which Hermes is integrated (refer to Chapter 4). Intelligent agent implementation platforms are described in Section 2.5. The implementation platforms discussed are those that conform to the BDI (Belief-Desire-Intention) model of intelligent agents. As the implementation of this research is done in Jadex [Pokahr et al., 2003], that particular platform is described in Section 2.5.1.

2.1 Agents and Agent Models

A general simple everyday definition of an agent is, “a person who acts on behalf of another person.” However, in this thesis, the term “agent” refers to software agents, which can be more specifically defined as “a software entity that acts on behalf of its human user.” Although there is no universal consensus on the exact definition of a software agent, there have been many descriptions of characteristics that are required for a software program to be considered an agent [Maes, 1995; Wooldridge, 1995; Franklin and Graesser, 1996; Georgeff et al., 1999]. Of these descriptions, the most commonly accepted is the definition of Wooldridge [2002] (which is based on his previous work [Wooldridge and Jennings, 1995a]).

Wooldridge states that the two defining attributes of an agent are that it is situated in an

†Part of the work presented in this chapter has been previously published [Cheong and Winikoff, 2009].
environment and that it is able to operate autonomously. An example of such an agent in a software environment could be one that monitors the amount of free disk space and warns the user if less than, say, 10% of disk space is available.

The example demonstrates that an agent, as defined, is not what would be considered intelligent. Wooldridge defines an intelligent agent as an agent (i.e. it is situated and autonomous) with three additional characteristics: reactivity, proactiveness, and social ability. Padgham and Winikoff's definition [Padgham and Winikoff, 2004] of an intelligent agent has two further properties: flexibility and robustness. Thus, an intelligent agent has the following characteristics:

- **Situated**: Intelligent agents exist in an environment.
- **Autonomous**: Intelligent agents are able to operate and pursue their design objectives without human intervention.
- **Reactivity**: Intelligent agents are aware of their environment and are able to respond in a timely manner to changes in the environment.
- **Proactivity**: Intelligent agents take the initiative to achieve their design objectives and are able to pursue them persistently over time.
- **Social ability**: Intelligent agents are able to communicate with each other and humans.
- **Flexibility**: Intelligent agents have multiple ways in which they can achieve their goals.
- **Robustness**: Intelligent agents are able to recover from failures.

In this thesis, the type of agents referred to is intelligent agents (or more specifically intelligent software agents), which is defined as having the aforementioned characteristics.

To help make this definition of an intelligent agent concrete, consider an example in which an agent is to drive to a particular destination. Once the agent is given the destination, it is autonomous in that it can operate without human intervention. Thus, it attempts to achieve this objective by selecting a route from its current location to the desired destination. Once the route is selected, the agent begins its journey.

The agent is able to react to changes in its environment. For example, it is able move to avoid obstacles on the road.

The agent displays its proactivity by selecting an alternative route when it finds its selected route blocked due to construction work. That is, the agent persistently attempts to achieve its objective over time. This is also linked with the agent’s flexibility and robustness as it is able to recover from failure and continues to pursue its goal by other means.

The previous example is of a single agent, however, in multi-agent systems, there is obviously more than one agent. In such systems, an agent’s social ability is of prime importance as it enables the agent to communicate with other agents, which, in turn, allows for more complex behaviours.
However, before the social ability of intelligent agents is further explored (refer to Section 2.2), the different types of agent models are firstly described.

There are a number of existing models for agents. The models can be broadly divided into three categories: reasoning agents, reactive agents, and hybrid agents (a combination of reasoning and reactive agents). Reasoning agents reason about what state of affairs to achieve and how to achieve these states. That is, these agents reason before taking appropriate actions. In reactive agent architectures, however, the agents are purely reactive in that they react to stimulus with no complex reasoning. Hybrid agent architectures attempt to marry reasoning and reactivity in agents to gain the advantages of both. They are composed of separate layers or sub-systems which deal with either pure reactivity or reasoning. A more detailed explanation and discussion of these three categories follows.

2.1.1 Reactive Agents

Reactive approaches to agency view intelligence as a product of the interaction between agents and their environments. That is, intelligence is an emergent property derived from simple behaviours or reactions.

A number of reactive architectures exist, such as PENGI [Agre and Chapman, 1987; Chapman and Agre, 1986], situated automata [Rosenschein, 1985; Rosenschein and Kaelbling, 1986; Kaelbling and Rosenschein, 1990; Kaelbling, 1991], and the agent network architecture [Maes, 1989; 1990; 1991], however, the most well-known is likely to be Brooks’ subsumption architecture [Brooks, 1991a;b]. Brooks’ work revolves around the aforementioned concept that the intelligence of a system is not situated in the actual system, rather, it is emergent between the interaction of the system and its environment.

This concept is central to Brooks’ subsumption architecture, which has two defining characteristics. The first is that an agent’s process of decision-making occurs through a number of behaviour modules which are intended to achieve specific tasks. A behaviour continually obtains percepts (environmental data) as input and these trigger a responding action. These behaviour modules do not contain complex symbolic representations and do not perform any symbolic reasoning, however, they can retain limited state information. The behaviours are reactive in that they do not perform any complex reasoning and are usually implemented as rules which map a percept to a response action.

As there are a number of behaviour modules present, it is possible for a percept to simultaneously trigger multiple behaviours. As such, the behaviours are arranged in layers which make up the second defining characteristic of Brooks’ architecture, the subsumption hierarchy. The hierarchy is used as a mechanism to determine the priority of action responses.

In the hierarchy, the higher layers are inhibited by the lower layers, thus giving higher priority to behaviours housed on the lower layers. Typically, the higher layer behaviours are more abstract than those on the lower layers. For example, in an exploratory mobile robot, an avoid obstacle
behaviour would be placed in a low-level layer whereas an *explore* behaviour would be placed on a high-level layer. This results in the robot always avoiding obstacles and exploring the terrain whenever it is not busy avoiding obstacles.

The reactive approach yields a number of advantages, including simplicity and low computational costs, as there is no reasoning, which entails that reactive agents deal well with dynamic environments. However, there are also a number of limitations faced by these reactive approaches. Since reactive agents do not model their environments, they require adequate information in their local environment to determine what action to take. Furthermore, it is difficult to determine how these decision-making processes can take into account non-local information. Thus, reactive agents are only able to take short-term actions.

Although one of the strong points of reactive agents is that their overall intelligence emerges from simple interactions, it is also a disadvantage. Due to their emergent nature, it is difficult to engineer reactive agents for particular tasks. This often leads to an experimental process of trial and error which can be tedious. Furthermore, although effective reactive agents can be created with a small number of behaviours, it is much more difficult to create reactive agents that have many layers as the interactions become too complex to understand and design, i.e. the approach does not scale well.

**Reactive Agents and Social Ability**

Reactive agents are able to engage in message-less communication through the environment. One example is Steels’ Mars explorer [Steels, 1990], in which reactive agents based on ants communicate with each other by leaving pheromone-like trails. Such communication is, however, limited and only applicable in particular situations.

In general, it is not clear how reactive agents can deal with complicated interactions. Although they are able to send and receive messages, it is difficult to determine how complex interactions, such as those reasoning agents engage in (refer to Section 2.1.2), can be attained with reactive agents. This is particularly true due to the nature of reactive agents and their lack of reasoning ability. For example, the speech act theory (refer to Section 2.2.2) is not a natural fit with reactive agents. That is, to a reactive agent, a *request* message and an *inform* message carry no semantic differences.

**2.1.2 Reasoning Agents**

Certain types of agents, those advocating strong agency, reason about their behaviour using concepts that are typically attributed to humans [Wooldridge and Jennings, 1995b]. These include mentalistic notions, such as beliefs, goals, plans, commitments, knowledge, and obligations. One such approach is the BDI (Belief-Desire-Intention) model, which is described in the following.
BDI Agents

In the BDI model, the mental state of an agent is represented by three distinct mental attitudes, beliefs, desires, and intentions. The agent’s beliefs represent its understanding of the state of its environment and also of itself. That is, the agent has beliefs about its environment and itself. These beliefs can be used as the basis for reasoning and decision-making.

An agent’s desires are objectives that it wishes to achieve. These are essentially goals that the agent would like to achieve. These desires can be inconsistent. For example, an agent may desire to use all of its funds to pay for a holiday trip and, at the same time, it may also have another (inconsistent) desire to use all of its funds to purchase a car. According to BDI theory [Rao and Georgeff, 1991], goals, i.e. desires that an agent has chosen to pursue, must be consistent (although the BDI theory requires that goals and intentions be consistent, this is not captured in the axioms, but is instead specified informally, and assumed to hold in the formal system.). However, this is seldom required or checked in implemented systems or platforms.

The intentions of an agent are its commitments and instantiated plans that it has towards achieving its goals. Like goals, intentions must be consistent according to BDI theory but this is often not checked in BDI implementations.

In his philosophical work [Bratman, 1987; Bratman et al., 1988; Bratman, 1999], Bratman explores the mental states of humans and explains that intention is central in understanding the relationship between people’s mental states and their actions. Although Bratman’s work is philosophical, Rao and Georgeff apply his work to intelligent software agents.

Rao and Georgeff [1991; 1992] have created an agent architecture and have used logics to formalize Bratman’s work. These BDI logics (which are based on CTL*, with a branching time future) are used to formalize the relationships between goals, beliefs and intentions, and to model commitment strategies. For example, an agent should not have goals that it believes it has already achieved, and it should only have intentions that have associated goals. Rao and Georgeff [1991] formalise three commitment strategies: blindly committed, single-minded, and open-minded. A blindly committed agent will not drop its intention until it believes that the intention has been achieved. This approach can be problematic as the agent will constantly attempt to achieve its intention even though it may be impossible to achieve. A single-minded agent will drop its intention when it either believes it is achieved or when it believes it is no longer possible to achieve it. An open-minded agent will also drop its intention if it is no longer relevant, e.g. the associated goal has been dropped.

Rao and Georgeff [1995] have defined an abstract BDI interpreter which shows how incoming events are processed by a BDI agent. The abstract BDI interpreter is shown in Listing 2.1.

```plaintext
initialize-state();
repeat
  1. options := option-generator(event-queue);
  2. selected-options := deliberate(options);
  3. update-intentions(selected-options);
  4. execute();
  5. get-new-external-events();
```
The inputs to the BDI interpreter are events, which are triggers that require the agents to respond in some manner. These events are stored in the *event-queue* structure. Internal events are events which an agent generates and assigns to itself, whilst external events are generated by the environment or other agents and assigned to an agent.

At the start of the cycle, a number of options are generated from the *event-queue* (step 1). From these options, those which are to be adopted (step 2) are added to the intention structure (step 3). After updating its intention structure, an agent is able to perform an atomic action (in response to some event) if desired (step 4).

New external events occurring during the cycle are then added to the *event-queue* (step 5) and internal events are added to the *event-queue* as they occur. The agent then drops all successful goals and unrealizable intentions (steps 6 and 7). The cycle then repeats.

The abstract BDI interpreter has been implemented on a number of different agent platforms and although there are slight differences between platforms, most implementations are guided by the aforementioned interpreter.

In most BDI implementations, the *options* that are selected (step 2) are generally plans: a series of steps that an agent can follow to achieve a particular intention. A plan consists of two conditions (*invocation* condition and *precondition*) and a body [Rao and Georgeff, 1995]. The plan body contains the steps which the agent is to execute. The *invocation* condition specifies the event that triggers the consideration of the plan, whilst the *precondition* states what conditions must be met in the current state in order for the plan to be applicable.

In most BDI implementations, beliefs are simply data structures that the agent possesses and is able to manipulate (i.e. create, read, update and delete). These are used to store any type of arbitrary information that the agent requires to operate successfully, which, generally, is information about an agent’s environment and itself.

In most BDI implementations, goals are modeled as events. For example, when a new goal is acquired, an agent will need to achieve the goal. The acquisition of the goal is modeled as an event which triggers a response from the agent.

When a BDI agent wishes to achieve a particular goal, it selects and instantiates an appropriate plan to do so. If the plan succeeds, the goal is achieved. However, if the plan fails, then the goal may not be achieved. In such a case, in certain BDI platforms, the agent is able to try another (appropriate) plan to achieve its goal. If that plan should also fail, the agent will try another alternative plan and so on until either the goal has been achieved, or all suitable plans have been exhausted, in which case the goal ultimately fails. This method of trying alternative plans and persistently trying to achieve a goal is what gives a BDI agent its flexibility and robustness.
It is possible for plans to generate sub-goals, i.e. related goals which will aid in achieving the current goal. The sub-goals are treated as any other goal in the system. That is, they are handled by plans using the same goal achievement cycle. Furthermore, it is possible for the plans achieving those sub-goals to also generate sub-goals and so on. This results in goal-plan trees as shown in Figure 2.1.

From the goal-plan tree example shown, it is obvious that a given goal may be decomposed into smaller sub-goals which can also be further decomposed. Each of the sub-goals are then handled by appropriate plans. One advantage of this approach is modularity: decomposing a goal into smaller sub-goals which can be addressed by more specific plans leads to greater re-use of the plans. A further advantage, which may not be obvious, is that this approach also leads to greater flexibility and robustness in the overall system. For example, there may be at each level of handling a goal/sub-goal, multiple suitable plans. If the first selected plan fails, say at the lowest level, then an alternative plan is tried at that level. This process repeats until either the sub-goal is achieved or all the alternative plans have been exhausted, in which case the sub-goal fails. The failure of the sub-goal leads to the failure of its parent plan. This results in the selection of an alternative plan, if available, to achieve the parent goal. Thus, this approach provides a simple, but powerful way to create flexible and robust agents. Although, if there are too many appropriate plans and high levels of plan failures, the system can spend a lot of time attempting to re-achieve goals [Cheong, 2003], which is not desirable. This, however, is not typically a problem as there are usually relatively few applicable alternative plans at any level and it is unusual for failure rates to be high throughout the system.
Reasoning Agents and Social Ability

Reasoning agents can have complex behaviours, which can lead to complex social behaviours and to more meaningful interactions. Speech acts (refer to Section 2.2.2) can be used to give meaning to the messages exchanged between agents. The agents will thus be able to differentiate between message types, such as request (e.g. a message sent to request a particular service) or inform (e.g. a message sent to transmit information), and act accordingly. Thus, the speech acts provide a suitable foundation to reason about communication which can be used to achieve more complex levels of interactions rather than simply exchanging messages and reacting to them. For example, if multiple agents have the same goals, they can organize themselves into teams and coordinate their actions to achieve their common goals. Furthermore, because reasoning agents can reason, they can be extended to operate with societal constructs, such as norms, obligations, and laws.

For more details on the social ability of reasoning agents and the types of interactions in which they may engage, refer to Section 2.2. This includes discussions on agent interaction patterns (refer to Section 2.2.3), agents working together (refer to Section 2.2.4), and societal design (refer to Section 2.2.5).

2.1.3 Hybrid Agents

Hybrid agents are an attempt to combine features of both reasoning and reactive agents. That is, an attempt to create agents that have the advantages of both reasoning and reactive agents. It involves creating different and separate subsystems to deal with aspects of reactivity and proactivity. In this approach, the subsystems are placed into a hierarchy of layers.

Typically, there are at least two layers in a hybrid architecture; one to deal with the reactive behaviours and the other for the proactive behaviours. However, it is possible to have more than two layers, and most architectures do. In a horizontally layered system, as shown in Figure 2.2 (a), all incoming percepts are sent to all layers. In turn, all layers process the percepts and produce suggestions. Each layer acts as an agent and they are essentially competing against each other. As each layer will produce a suggested action, a mediator is required to determine which action to execute. This mediator, a centralized control, is problematic in two ways. Firstly, the designer will need to determine all possible interactions between all the layers (which can be quite large) and secondly it is a bottleneck in the system.

Vertically layered architectures are able to reduce some of these problems. There are two types of vertically layered architectures, one-pass and two pass. In a one-pass architecture, refer to Figure 2.2 (b), control flows systematically through all the layers, until the final layer produces an action. In a two-pass architecture, refer to Figure 2.2 (c), as with one-pass architectures, control flows up the layers, and then, additionally, back down the layers and an action is generated at the end.

Vertically layered architectures have less interactions between layers than horizontally layered architectures. In a vertically layered architecture, each layer interacts with its neighbouring layers
Figure 2.2: Layered Hybrid Architectures (based on [Müller et al., 1995])
(i.e. the layers directly above and below it), however, in a horizontally layered interaction each layer interacts with every other layer in the system. This reduction in interaction complexity results in a loss of flexibility: for a vertically layered architecture to generate an action, control must pass through each layer. Examples of horizontally and vertically layered architectures are TouringMachines [Ferguson, 1992] and InteRRap [Müller, 1997] respectively.

The horizontally layered TouringMachines architecture contains three layers, modeling, planning, and reactive, and a control subsystem that determines which layer should control the agent by suppressing sensory information. Of the three layers, the social aspect of the agent is in the modeling layer, which contains representations of entities in the environment, including other agents and the agent itself. The modeling layer predicts conflicts between agents and generates goals that need to be achieved to resolve these conflicts.

The two-pass vertically layered architecture of InteRRap is similar to the TouringMachines architecture in that it contains three layers, behaviour, plan, and cooperation, which serve similar purposes to their TouringMachine counterparts. Each of the layers has access to a layer-specific knowledge base, which is a representation of the world for that particular layer. The knowledge base for the cooperation layer, which deals with social interactions, called social knowledge, represents the plans and actions of other agents in the environment.

Hybrid Agents and Social Ability

As hybrid architectures are composed of both reasoning and reactive systems, the agents are able to communicate at any of the social, reasoning or reactive layers. The layers at which the agents communicate will depend on the designer and particular application.

2.2 Social Ability

The previous section on agents and agent models has outlined a commonly accepted core set of characteristics for intelligent agents and has described a number of agent models. As explained in Section 2.1.2, reasoning agents are able to engage in richer interactions than their reactive counterparts. Thus, the focus of this thesis is on reasoning agents, specifically those of the BDI tradition, and their interactions as reasoning agents are better suited to complex interactions than reactive agents. Furthermore, work done with reasoning agents should be able to be incorporated into hybrid agents.

In this section, the social ability of agents are explored in further detail, beginning with agent communication languages, and progressing on to speech acts, agent interactions to reach agreements and to work together, and finally societal design.

Through their social ability, intelligent agents are able to engage each other in various different types of interactions. At the lowest and most basic level, intelligent agents are able to communicate through exchanges of messages. These messages are usually conformant to defined agent
communication languages (ACL), such as KQML, KIF, and FIPA ACL (refer to Section 2.2.1).

Speech acts (refer to Section 2.2.2) provide semantics to messages exchanged between intelligent agents using the agents’ mental states. By using the agents’ ability to communicate through ACLs and to understand semantic details through speech acts, it is possible to further build upon these to create interaction patterns for common situations in which agreements need to be reached, such as auctions, negotiation and argumentation (refer to Section 2.2.3). Agents are not restricted to situations in which they attempt to agree with each other. They are able to engage in more generic behavioural interaction patterns, such as working together, including collaboration and teamwork (refer to Section 2.2.4), in which the agents are not only communicating with each other, but also have common goals and responsibilities towards each other.

At the top-most level of interaction, agents can be organized into organizational groups and societies, which in the future may resemble human groups and societies. In such situations, desirable behaviours are obtained from the agents by the existence of societal norms, obligations and laws (refer to Section 2.2.5). These different aspects of social ability and levels of interactions are discussed in the following sections.

2.2.1 Agent Communication Languages

Agent communication is important as it provides a means through which the social ability of intelligent agents is made possible. Agent communication can be seen simply as agents exchanging messages\(^1\), however, the ability to carry out this simple activity is the basis for more complex interactions.

In order for agents to communicate successfully, they must be able to understand the messages received from other agents and they must be able to form messages that will be understood by agents who receive them. For this to occur in open systems, i.e. systems in which agents are not specifically designed to inter-operate but are required to do so (e.g. AgentCities\(^2\) and OpenNet\(^3\)), the agents must use standard agent communication languages and ontologies. In closed systems, i.e. systems in which agents are specifically designed to inter-operate, a standard agent communication language is not as important as the designer is able to create specific messages that the agents will understand. However, in open systems, standard agent communication languages are critical for communication, as otherwise agents will not be able to communicate with each other.

Commonly used agent communication languages (ACLs) are KQML, KIF, FIPA ACL and FIPA SL. The Knowledge Query and Manipulation Language (KQML) [Patil et al., 1992; Mayfield et al., 1996] is an “outer” language for agent communication. An envelope is defined for a message

---

\(^1\)Generally, agent communication involves exchanges of messages, although it is possible to have message-free communication, such as using ant-like pheromones [Steels, 1990] or by an agent observing other actions of another agent.

\(^2\)http://www.agentcities.org/

\(^3\)http://x-opennet.org/
which allows the agent sending the message to state its recipient. For example, the KQML message shown in Listing 2.2, can be intuitively deciphered as a message sent to the warehouse agent to enquire about the number of laptops in stock. The ask-one performative states that exactly one reply is needed in response to this message. KQML has a number of defined performatives (e.g. achieve, ask-about, broadcast, and subscribe), a list of which can be found in [Wooldridge, 2002]. The other components of the message represent its attributes. For example, the content field is used to specify the contents of the message and the receiver specifies the intended receiver of the message. The language and ontology fields state in which particular language the contents are expressed and what ontology is being used for communication respectively. Other possible fields include: sender (specifies the sender of the message), reply-with (the sender states an identifier for an expected reply), and in-reply-to (used to reference the identifier received in the reply-with field of a previous message).

```
(ask-one
  :content (QTY LAPTOP ?price)
  :receiver warehouse
  :language LPROLOG
  :ontology E-COMMERCE)
```

Listing 2.2: A KQML Message

In contrast to KQML, the Knowledge Interchange Format (KIF), is an “inner” language for agent communication. It allows for the representation of knowledge about a particular domain and is primarily intended to be used to form the contents of KQML messages.

Listing 2.3 shows an example of a KIF expression states that the weight of laptop1 is 2 kg.

```
(* (weight laptop1) (scalar 2 kg))
```

Listing 2.3: A KIF Message

The FIPA ACL\textsuperscript{4} (Foundation for Intelligent Physical Agents Agent Communication Language) [Foundation for Intelligent Physical Agents (FIPA), 2002a;b] is similar to KQML in that it defines an “outer” language for messages. FIPA ACL contains 20 defined performatives (e.g. agree, inform, cancel, confirm, etc.) for defining the intention of message contents.

Listing 2.4 shows an example of a FIPA ACL message using the refuse performative. In the example, the Vendor agent is refusing to sell a laptop to a Customer agent as the Customer does not have enough funds. The example shows that FIPA ACL messages are very similar to KQML messages as the message attribute fields are similar and the structure of the message is the same. The FIPA ACL Message Structure Specification document [Foundation for Intelligent Physical Agents (FIPA), 2002b] contains details on the structure of the FIPA ACL messages. The main difference between FIPA ACL and KQML is the performatives they provide. A detailed list

\textsuperscript{4}The description herein is based on published standards which, although a few years old, are the most recent version. Work on FIPA is now continuing under the IEEE Computer Society but no further standards have emerged.
and definition of FIPA ACL performatives can be found in the *FIPA Communicative Act Library Specification* document [Foundation for Intelligent Physical Agents (FIPA), 2002a].

```plaintext
{(refuse
  :sender (agent-identifier :name vendor)
  :receiver (set (agent-identifier :name customer))
  :content
    "((action (agent-identifier :name vendor)
       (sell-laptop model-1 2000))
     (insufficient-funds acct123))"
  :language fipa-sl
}
```

*Listing 2.4: A FIPA ACL Message with FIPA SL used as the Content Language*

Using the language field of FIPA ACL, it is possible to specify which “inner” language to use. FIPA SL0 and FIPA SL1 (Foundation for Intelligent Physical Agents Agent Semantic Language) [Foundation for Intelligent Physical Agents (FIPA), 2002c] are two examples of such a “inner” languages and are intended to be used with FIPA ACL (although it is possible to use others as well). More details on FIPA SL can be found in the *FIPA SL Content Language Specification* document [Foundation for Intelligent Physical Agents (FIPA), 2002c].

The ACLs provide the syntax for messages which allows heterogeneous agents to inter-operate by using these standard communication languages. Furthermore, ACLs provide semantics using speech acts. These speech acts are explained in the following section.

### 2.2.2 The Speech Act Theory

Agent communication languages provide syntactical definitions of languages to allow agent communications to be realized. To further build upon that, semantics are required so that agents are not only able to detect different message types but to also understand what messages such as “inform” mean. This can be achieved by using the *speech act theory*, which is foundational to individual messages and their meaning, to give semantics to these agent communication languages in terms of the mental states of the intelligent agents.

Consider Figure 2.3, which provides a definition of the *Request* and *Inform* speech acts. These definitions show the link between the messages and mental states. The definitions state what preconditions (of which there are two types: can do and want) are required before a speech act is performed and what the effects of the speech act will be once it has been performed. For example, for the *Request* speech act to be performed, the following preconditions have to be true:

- The speaker, $S$, must believe the hearer, $H$, can perform $\alpha$;
- $S$, must believe that $H$ believes that $H$ can perform $\alpha$; and
- $S$ must believe that it wants to request $\alpha$ from $H$ (i.e. this instance of this request).

After the *Request* speech act has been performed, the effect is that the $H$ believes that the $S$ believes that $S$ wants $\alpha$, i.e. that $H$ is aware of $S$ wanting $\alpha$. The *Inform* speech act is defined in a similar way to the *Request* speech act.
Speech acts originated from the work of philosopher John Austin [Austin, 1962]. In his work, Austin identified a certain class of natural language utterances (referred to as speech acts) that had similar characteristics to actions, i.e. these utterances change the state of the world in a similar way to physical actions. Thus, the speech act theory views the performance of a communicative to be the same as any other action an agent might perform.

Austin identified a number of performative verbs, such as request, inform, and promise, which correspond to different types of speech acts. Furthermore, Austin identified three aspects of speech acts: locutionary act, illocutionary act, and perlocutionary act.

The locutionary act is the act of making an utterance. It is the actual act of speaking, i.e. making appropriate sounds that make up a sentence. The illocutionary act is the “linguistic function” or purpose of the utterance. For example, if the speech act is a promise, then there are changes to the mental states of the hearer and the speaker. For the speaker, this may mean an intention to carry out the promise and the hearer may believe the speaker will carry out the promise. The perlocutionary act is the intended effect on the hearer. For example, a threat, i.e. an illocutionary act, is intended to induce fear or compliance in the hearer.

Further work modeled speech acts as actions that could be performed by rational agents in order to achieve their goals [Cohen and Levesque, 1990].

### 2.2.3 Interacting to Reach Agreements

The agent communication languages and speech acts provide syntax and semantics for a single message, however, interactions are usually composed of more than one message. As such standard interaction patterns and application-specific patterns are required for commonly occurring scenarios. These patterns are usually defined in terms of protocols: sets of rules governing interactions.
These rules typically dictate matters such as which agent sends a message, when a message has to be sent and which messages can be sent in particular situations.

Protocols for common interactions, such as auctions, negotiations, and argumentation, in which agreements between agents are to be attained, have been extensively studied. These types of interactions are briefly discussed in this section.

In auctions, the agents are attempting to settle on a mutually acceptable price, whilst in negotiations, the agreement they are trying to reach is not solely limited to price and is more general (e.g. the agents are able to negotiate over a number of different aspects, such as the specifics of products, warranty, after sales service, etc.). Argumentation is even more general than negotiation. In this situation, the agents are not necessarily attempting to reach an agreement on a product; they are logically attempting to reach an agreement on an argument. In these settings of attempting to reach agreements, game theory [Binmore, 1992] is sometimes used to analyze which strategies are rational or dominant.

As auctions are structured interactions, it is possible to design protocols which agents can follow in such situations. Generally, an auction occurs between an auctioneer and multiple bidders. The objective of the auction is for the auctioneer to allocate the item to a bidder. In most situations, the auctioneer attempts to maximize the price at which the item is allocated, whereas the bidders wish to minimize the price. The manner in which the auctioneer and the bidders interact is defined by the auction protocol, which will vary depending on the type of auctions. The strategy of the bidders will also vary based on the type of auction (and on the agent).

Auctions are typically classified using the following attributes:

- Ascending or Descending: Are the offered bids ascending or descending?
- Open cry or sealed bid: In an open cry auction, when a bidder places a bid, all other bidders know of the bid, however, in a sealed bid, the other bidders do not know the bidding price.
- first- or second-price: In a first-price auction, the highest bid is the agreed price, whereas in a second-price auction, the second highest bid is the agreed price.
- One-shot or multiple round: In reference to the number of bidding rounds.

Different auction types include English, Dutch, first-price sealed bid, and Vickrey auctions. The English auction, for example, is a first-price, open cry, ascending, multiple round auction. That is, the auction has more than one round of bidding in which the bidders must place increasing bids. A bidder's bidding price is known by all other bidders in the auction and the bidder with the highest bid wins the auction, and will pay the price it bid.

The main focus of the body of work concerning auctions is to develop protocols that have desirable properties, such as honesty being a dominant strategy.

\footnote{In a Dutch auction, offered bids are descending as the value of the items on auction usually depreciate with time. For example, items such as flowers have less value at the end of the day than at the start.}
Although negotiations are less structured interactions than auctions, they are still fairly specific interactions as they aim to reach agreements over one or more issues. In general negotiations are composed of four components:

- A negotiation set which contains all the possible proposals the agents can make.
- A protocol which governs the legal proposals the agents can make, based on the prior history of the negotiation.
- A collection of strategies (one for each agent) which the agents can use to determine what proposals to make.
- A rule that determines when an agreement has been reached.

A negotiation occurs in a series of rounds in which every agent makes a proposal. The proposals which the agents make depend on their strategy, must be taken from the negotiation set, and must be legal in accordance to the protocol. If an agreement is reached, as determined by the agreement rule, then the negotiation is terminated.

Simple negotiations are one-to-one and single issue, i.e. there are only two agents involved and they are negotiating over a single issue, such as the price of an item. However, it is possible to have many-to-one and many-to-many negotiations. Furthermore, it is possible to have multi-issue negotiations, i.e. the agents involved negotiate over multiple issues, such as the price, warranty, delivery time, etc. These types of negotiations are much more difficult than single issue negotiations as there are usually no clear concessions (i.e. it is not clear which attribute values should increase or decrease).

As with auctions, the nature of the negotiation body of work is to develop strategies that have desired properties, and that perform well.

One difficulty with negotiations is that the agents are not able to justify their negotiation stances or propositions. Typically, when humans negotiate, they will justify their stances and propositions. For example, a vendor might justify a high price due to a particularly good feature of the item being sold. Furthermore, in negotiations, once an agent has taken a stance, it cannot change its position, e.g. if an agent wanted a particular feature to be present in an item it is purchasing, it is assumed that this preference will not change.

Argumentation-based negotiation can be seen as a more complex type of negotiation in which proposals and negotiation stances can be justified. In multi-agent systems, argumentation is a process in which an agent attempts to convince another agent of some state of affairs. In this process, the agents put forth arguments for and against propositions along with justifications of their arguments.

Philosopher Michael Gilbert [Gilbert, 1994] identified four types of arguments: logical (an appeal to logic), emotional (an appeal to feelings and attitudes, e.g. “How would you feel if ...”), visceral (the physical and social aspects of argumentation, such as participants stamping their feet
during an argument), and *kisceral* (an appeal to the intuitive, mystical, or religious). In regard to multi-agent systems, the logical mode of argument is the most appropriate.

Logical argumentation is usually deductive in nature, e.g. “if you accept $A$ and $A$ implies $B$, then you must also accept $B$.” Argumentation dialogue consists of a series of arguments by two agents that take turns in putting them forth. The first agent puts forth its argument to convince the other agent, however, the second attempts to defeat the first agent’s argument by undercutting (disproving an underlying fact required for the argument to hold) or rebutting it (proving a contradictory fact which is true). The first agent must then respond to the second agent’s counter argument, by presenting an argument that defeats it, if it can. This cycle continues until an agent cannot put forth any more arguments, in which case the other agent wins the argument (i.e. the agent that put forth the last argument is the winner).

The various interactions in which agents have to reach agreements have been described in this section. However, agent interactions are not limited to situations in which agreements need to be reached. More generic interactions, such as communicating about coordination, are described in the next section.

### 2.2.4 Interacting to Work Together

The previous sections have shown how intelligent agents are able to communicate, how speech acts give semantics to these communications, and how agent interaction patterns are used to allow the agents to exchange messages in a meaningful way. This section describes how intelligent agents are able to work together in a constructive manner.

In multi-agent systems, whether open or closed systems, agents are typically required to inter-operate to achieve their individual goals or system goals. The degree to which they work together will vary depending on a number of factors. For example, in a closed system, the agents can be designed such that they operate to achieve joint goals. In an open system, the same effect can be achieved if two self-interested agents share similar goals. However, if two agents have dissimilar but complementary goals it is also possible for them to work together to achieve their complementary goals. An example of such a situation is a vendor agent selling a particular item that an agent needs to purchase. These two agents work together (e.g. negotiate) in an attempt to achieve their respective goals of buying and selling.

As the previous examples demonstrated, there are different degrees to which agents are able to work together. A taxonomy of multi-agent interactions has been developed to help describe them [Parunak et al., 2002].

*Correlation* is the most general term to describe agents working together. It is also the lowest level of working together and is defined as joint information between agents. Correlation is a purely behavioural notion and has no cognitive notions. That is, agents do not necessarily have to be *intelligent* agents to have correlation with other agents. In order to determine correlation, only knowledge about the agents’ actions is required.
One should note that correlation is a broad definition as it does not take in consideration how the joint information comes about, only that it exists, e.g. there can be correlation without communication. *Coordination*, however, takes this into consideration and, in fact, is defined as correlation with a focus on information flow. That is, joint information that is brought about through the communication flow between agents. The communication itself may be direct, i.e. from agent to agent, or indirect, e.g. from an agent to its environment and then from the environment to another agent. Therefore, the inter-agent details are required to be able to identify coordination within a system.

*Cooperation* is similar to coordination, however, instead of focusing on the communication flow, it focuses on an agent’s intent. Specifically, cooperation requires agents to have joint intentions. Thus, to determine if agents are cooperating, their internal details are required.

*Collaboration* between agents is a result of cooperating agents that have direct decentralized communications with each other. This is particularly important in multi-agent systems as collaboration will enable agents to achieve more together than a single agent can. There are also certain activities that agents cannot undertake individually due to lack of resources, information, abilities, etc. Thus, through collaboration, agents can request and provide services to each other, and they can also share knowledge and act in a coordinated manner to achieve their goals.

An example of collaboration is *cooperative distributed problem solving*, which stems from the early distributed artificial intelligence research. Cooperative distributed problem solving focuses on how a given problem can be solved by multiple heterogeneous agent-like entities [Durfee et al., 1989]. These agent-like entities cannot solve the given problem individually and, thus, must collaborate to do so.

The *contract net* [Smith, 1980] is a well-known framework for distributed problem solving. The contract net protocol is a high-level protocol which facilitates the distribution of tasks. For example, when an agent encounters a task that it cannot or does not want to solve, the agent divides the task into smaller sub-tasks. The agent then advertises (i.e. broadcasts) these sub-tasks to a group of agents. If these agents are able and willing to solve the sub-tasks, they propose bids for particular tasks. The agent which advertised the sub-tasks then awards them to particular agents to be solved.

The work of Cohen and Levesque [1991] investigates agent *teamwork* and is in the same area as the work of Parunak et al. [2002], although, a different taxonomy is used. Whereas Parunak et al. [2002] provide a high-level taxonomy of activities that agents engage in together, Cohen and Levesque [1991] describe how teamwork can be specified between agents. A common point between the two is the agreement that joint intentions are required for agents to carry out more complex group activity than simply coordinating their actions (referred to as *cooperation* in Parunak et al. [2002] and as *teamwork* in Cohen and Levesque [1991]).

In the work of Cohen and Levesque [1991] teamwork is viewed as being more than agents simply communicating and coordinating with each other. To explain the difference between coordination
CHAPTER 2. BACKGROUND

and teamwork, as example based on automobile traffic and a convoy [Cohen and Levesque, 1991] is used.

Consider ordinary automobile traffic. The drivers on the road communicate with each other by using their indicators, and they coordinate their movements with each other to avoid accidents. If one vehicle should break down, the other drivers will simply avoid it and continue along their way. In this particular scenario, the drivers do not have a common goal. Thus, it can be said they are coordinating with each other, but they are not undertaking teamwork.

Consider now an example of a convoy. The drivers have a common goal of arriving at a particular destination together. If a vehicle should break down, the other drivers are likely to help repair the vehicle or transfer its driver and contents to their own vehicles, and then proceed to the destination. In this situation, the drivers are working together as a team as they have a common goal. As can be seen, the difference between coordination and teamwork becomes more apparent in problematic situations.

As teamwork is more than agents coordinating their actions, architectures which specifically support teamwork are required. One such example is STEAM (A Shell for TEAMwork) [Tambe, 1997b;a], which is a general model for teamwork. Rather than allowing for coordination plans to be created for individual agents, which is common in most agent platforms, STEAM takes a different approach and provides a general model of teamwork which allows agents to reason about coordination and communication. To that end, STEAM must provide agents with the facilities to allow for the representation of and reasoning about team goals and plans. As such, STEAM is based on the joint intentions theory [Levesque et al., 1997]. The joint intentions theory provides concepts for teamwork, such as joint persistent goals. A joint persistent goal is a team goal, i.e. a goal which a team of agents wishes to achieve.

Another teamwork-related theory is SharedPlans [Grosz and Kraus, 1996]. A SharedPlan (SP) is formed between collaborating agents to carry out a joint activity. This SP is composed of a set of individuals actions undertaken by individual agents, which when completed, will constitute carrying out the joint activity. There are two varieties of SharedPlans: Full SharedPlans (FSP) and Partial SharedPlans (PSP). The distinguishing factor is that in an FSP, the plan specifies the individual actions completely, however, in a PSP, only some of the individual actions may be specified. As the team progresses through the PSP, the individual agents must mutually agree on the following actions to be taken. Thus, for each action or sub-set of actions in the PSP, the related sub-team must form an SP.

An example of another team-oriented architecture is JACK Teams [Age, 2005b], which is an extension of JACK Intelligent Agents [Busetta et al., 1999]. In the JACK Intelligent Agents platform, the basic entity is an individual agent which is composed of plans and beliefs, and is a BDI reasoning entity. These agents are able to communicate with each other and cooperate to some degree. However, in JACK Teams, a team is a BDI reasoning entity which is composed of sub-teams. A sub-team is represented by the same team entity, and, thus, a sub-team is composed
of sub-teams. An individual agent is modeled as a team with no sub-teams. This allows for the construction of a hierarchical relationship between teams and sub-teams. The teams are equipped with programmer-defined teamplans which are used to coordinate and delegate tasks to sub-teams.

A team is not simply a collection of agents, as it has its own beliefs, goals and teamplans. The teamplans allow for specific agents or sub-teams to fill roles required to carry out the steps in the teamplans. Tasks can be delegated to sub-teams by the \texttt{@teamAchieve} statement. When a task is delegated in such a manner, the team that sent the \texttt{@teamAchieve} waits until the sub-team has completed its task.

Although the JACK Teams platform is appropriate for team work, it is specialized and is not appropriate for some circumstances. For example, if one was to model an interaction between a Customer and Merchant, there is a need for coordination and cooperation, but there is no clearly appropriate place to create a team as such. A more general approach, i.e. one which has no assumption of teamwork, would be more appropriate. Furthermore, the JACK Teams approach is centralized in that tasks are delegated by top-level team entities to sub-team entities. In some circumstances, a decentralized model may be more appropriate as it will allow greater autonomy to individual agents.

2.2.5 Agent Societies

The preceding sections have discussed how the social ability of intelligent agents allows them to realize and carry out complex behaviours, such as collaboration and teamwork. These discussions have been from the perspective of the individual agent, however, in this section, the social aspect of intelligent agents is presented from the societal perspective rather than from the point of view of individual agents. Unlike the previous sections, the discussions found herein are not about enabling the social ability of individual agents, but rather, they are about defining societal-level mechanisms, such as norms, obligations and social laws, to provide rules of interactions for intelligent agents.

Societal norms represent particular expected patterns of behaviour in given situations. These norms are relative to particular societies or groups. For example, in some societies, under certain circumstances, it is customary to shake hands when greeting others, whereas in other societies a bow is required. Social obligations and laws are similar to norms, but are stronger in that they carry more authority.

The difference is most noticeable in the degree of the punishment. For example, it might be a norm to queue in arrival order to board a bus. As this is a societal norm, it is not really enforced. Not conforming to this norm will not result in any drastic penalties (apart from, perhaps, a few rude stares from others). Social obligations and laws, on the other hand, are enforceable and carry with them stronger penalties. For example a social law is that one should not steal. If this is violated, then this law is enforced by punishing the violators. Thus, in a social setting, norms, obligations, and laws provide guidelines for agents and influence their behaviour. Although there are no (real) punishments for violating norms, the violation of obligations and social laws are
punishable [Dignum et al., 2000]. In this manner, an agent’s behaviour is more strongly influenced by obligations and laws than norms.

An agent’s social behaviour can be determined at three levels: private, contract, and convention. At the private level, the agent makes private decisions about actions that it will carry out based on its obligations and/or goals. The contract level contains obligations between agents, including their descriptions and consequences of their violations, which creates some dependency between agents. The convention level details conventions, e.g. norms, to which agents should adhere to.

Two approaches to societal design of agent interactions are Islander and OperA (refer to Section 2.3.2 for more details).

The social ability and the interactions of agents, which are quite varied, have been discussed in this section. Details on how to design such interactions, using both the traditional and alternative approaches, are presented in the following section.

2.3 Interaction Design

To create multi-agent systems, it is necessary to firstly design the agents and, secondly, to implement them. In this process, the design of agent interactions plays a crucial part. There are a number of ways in which agent interactions can be modeled and designed. The most obvious and simplest approach is to focus on information exchange between interacting agents, i.e. the messages, and to specify and design interactions in terms of possible sequences of messages. This is indeed the approach that many current design methodologies use. In this thesis, such approaches are referred to as “traditional” or “message-centric”.

Although simple and obvious, the problem with this approach is that it only captures the interaction at a superficial level. By focussing only on the communicative acts, information such as the reason for uttering the communication, is lost. This approach results in limited failure recovery options. For example, when failures are encountered, the only recovery option is to use alternative message sequences that have been explicitly prescribed. If other information, such as the reason for the utterance (i.e. the goal of the communication), is available, that can lead to greater failure recovery options, which will in turn lead to greater flexibility in the interactions.

These message-centric approaches prescribe legal sequences of messages possible in the interactions and the agents are forced to follow these message sequences. This is detrimental to intelligent agents as it subdues or even inhibits defining agent characteristics such as autonomy, flexibility and robustness. Thus, the message-centric approaches are a poor fit for intelligent agents.

The traditional message-centric approach to designing agent interaction is presented in Section 2.3.1. The standard process is explained along with the resulting protocols and a popular notation for message-centric design, AUML (Agent UML).

In Section 2.3.2, alternatives to the message-centric approach are presented. These alternatives include commitment-based and landmark-based approaches, as well as other work which affects agent interaction design.
2.3.1 Traditional Message-Centric Approaches to Agent Interactions

Agent interactions have traditionally been specified in terms of interaction protocols expressed in notations which focus on the message exchanges between the agents. Common notations for expressing such agent interactions are Agent UML (AUML) [Odell et al., 2000; Huget and Odell, 2004], Petri nets [Reisig, 1985], and finite state machines. Moreover, in certain methodologies, the design of the agent interactions also focuses on the messages. For example, to create an agent interaction in the Prometheus methodology, the designer is advised to discern what messages should be sent between agents by examining received messages and determining what messages can be sent in response.

As Petri nets [Reisig, 1985] can be used to model information flow, they can be used to represent agent interaction protocols [Cost et al., 2000]. A Petri net has three components: places, transitions, and arcs. The arcs are directional and are used to link places to transitions or transitions to places. Places may contain tokens, which are used to “fire” (i.e. execute) the Petri net. This involves moving the tokens around the Petri net through transitions which are enabled if the incoming place contains a token, and produces a token on the outgoing place.

We believe that Petri nets suffer from a number of similar issues as the AUML notation in regard to representing agent interactions. As such, in this section, we focus on the AUML notation, which is also used commonly in a number of methodologies. The AUML notation [Odell et al., 2000; Huget, 2004] has been used for a number of different aspects of agency. Examples include: as part of agent-oriented methodologies [Garcia-Ojeda et al., 2004; Padgham and Winikoff, 2004], to represent social structures [Parunak and Odell, 2001], and to allow execution of AUML interaction diagrams [Ehrler and Cranefield, 2004]. A software support tool has been developed [Winikoff, 2005] for textually specifying AUML sequence diagrams [Huget et al., 2003], which are quite often used to specify agent interactions.

For the purposes of this thesis, only the part of AUML related to agent interactions, i.e. the sequence diagrams [Huget et al., 2003], are considered. It should be noted that the AUML sequence diagram has developed from its original version [Bauer et al., 2000; 2001] to a more recent version [Huget et al., 2003; Huget and Odell, 2004] which is influenced by UML 2.0. In this thesis, only the more recent version is considered.

The AUML notation for interactions has been adopted by methodologies, such as Gaia, MaSE, Prometheus, and Tropos, and is commonly used. For example, consider the protocol depicted in Figure 2.4, in which a Customer is negotiating the details of a particular product it is interested in purchasing from a Vendor.

Many practicing software engineers will be familiar with such a diagram as it is based on UML 2.0 sequence diagrams and shares many similarities. The roles (e.g. Customer and Vendor) appear in rectangles at the top of the diagram with life lines extending from the rectangles. Similarly to UML sequence diagrams, time progresses down the life line and message exchanges are shown as labeled directed lines between the roles.
Figure 2.4: AUML Protocol diagram for Negotiate Details Interaction
The *alt* box indicates an alternative or branching of message sequences. Each region of the *alt* box represents one alternative message sequence. For example, when the *Customer* enquires about the availability of the product, the *Vendor* will reply with an *Available* message if it is available (the first region of the *alt* box in Figure 2.4), otherwise it will reply with a *Not Available* message (the second region of the *alt* box), and the message sequence will proceed according to the appropriate box region. That is, if the product is available, the agents then progress to negotiate over the details of the product, otherwise the interaction is terminated. A *break* box is used to indicate the end of a protocol (although the *break* box is not strictly necessary at this point in this particular interaction as the *Terminate Interaction* message is the last message of this protocol).

Similarly, when the *Customer* enquires about a particular colour of the product, the *Vendor* is able to reply with *Colour Accepted* or *Colour Rejected* messages. If the proposed colour is accepted, the agents progress to a sub-protocol, titled *Negotiate Price*, as indicated by the *ref* box. However, if the proposed colour is rejected, the *Customer* is able to propose another colour. This is shown on the protocol using an A UML *Continuation*, depicted by a rectangle with rounded corners which contains a label denoting its name. Continuations with a small triangle preceding their label signify a target (“label”) whereas continuations with a small triangle on their right side (after the label) denote a “goto”. When a *goto* continuation is encountered, the protocol moves to the beginning of its respective target label and continues the interaction from there.

### 2.3.2 Alternative Approaches to Traditional Agent Interactions

There are various alternative approaches to the traditional message-centric interaction design. These alternative approaches, in contrast to message-centric approaches, diverge from focusing on the messages to design the interaction. Instead, they focus on various other elements of the interaction, such as social commitments or the states of the interaction, which guide the agents to communicate (i.e. exchange messages). Thus, these alternative approaches are at a higher level of abstraction than message-centric approaches.

Although the end product is still agent interactions in which agents communicate through exchanges of messages, designing these interactions at a higher level of abstraction has a number of advantages, the foremost of which is that valid message sequences *emerge* from the interaction in a less constrained manner, which increases the flexibility of the interaction. This is quite different to message-centric interaction design in which, as explained in Section 2.3.1, valid message sequences must be *explicitly specified* and are often too constrained.

Alternatives to message-centric design include commitment- and landmark-based approaches, along with a number of other alternative approaches. In commitment-based interactions, agents are guided by social commitments to communicate and progress through the interactions.

There are a number of approaches based on the notion of social commitments [Singh, 1991; Castelfranchi, 1995]. However, the two main bodies of work in this area are the commitment machines of Yolum and Singh [2002; 2004], and the social commitments of Flores and Kremer
One reason for the popularity of social commitment-based approaches is that social commitments are verifiable. That is, social commitments are independent from an agent’s internal structure and mental states and are observable by other agents. These are two important properties, as the first allows the social commitment to be utilized by heterogeneous agents and the second allows all agents involved in the interaction to determine if a commitment has been violated or not.

**Commitment Machines**

In commitment machines [Yolum and Singh, 2002; 2004], a (social) commitment between two agents represents one agent’s responsibility to bring about a certain condition for the other agent. Agents progress through the interactions by the acquisition, manipulation, fulfillment and discharge of the commitments. Message sequences are not explicitly defined, instead, they emerge as the agents manipulate their commitments in an attempt to reach a desired state where they have satisfied all their commitments.

There are two types of commitments in commitment machines: base-level and conditional commitments. A base-level commitment is denoted as $C(x, y, p)$, which states that a debtor, $x$, must bring about a condition, $p$, for creditor, $y$. For instance, the commitment $C($vendor, customer, productDelivered$)$, states that the Vendor is committed to the Customer to bring about a state in which the product has been delivered.

A conditional commitment is denoted as $CC(x, y, p, q)$ and states that if a condition, $p$, is brought about, then debtor, $x$, will be committed to creditor, $y$, to bring about condition, $q$. For example, $CC($vendor, customer, paid, productDelivered$)$, states that if payment is made, then the Vendor will be committed to having the product delivered.

To manipulate the commitments, the following operations are available:

1. **Create**: Only the debtor is able to use this operation to create a commitment between itself and the creditor.
2. **Discharge**: Only the debtor uses this operation to absolve the commitment between itself and the creditor when the commitment has been successfully fulfilled.
3. **Cancel**: This operation cancels a commitment and is typically followed by the creation of another commitment to make up for the former commitment.
4. **Release**: This operation can only be performed by the creditor to release the debtor from its obligation to a commitment.
5. **Assign**: This operation is used to assign a new agent as the creditor of the commitment.
6. **Delegate**: This operation is used to replace the debtor. That is, it delegates the responsibility of fulfilling the commitment to another agent.
Using the operations specified, and the two types of commitments, the agents are able to navigate through interactions by manipulating these commitments to achieve their objectives.

Consider an e-commerce example based on a simplified version of the NetBill protocol, taken from [Yolum and Singh, 2001], in which a customer is attempting to purchase a product from a vendor. Figure 2.5 shows a representation of the interaction along with the appropriate commitments and operations. The definition of commitment machine interactions requires roles, interaction states (e.g. commitments and fluents), and action effects (the effects that defined actions have in the interaction), however, in the following example, only the commitments are focused upon.

The customer initiates the interaction by requesting a quote for the product from the vendor. Upon receiving the request, the Vendor sends a quote to the customer and makes an offer by creating two commitments:

\[ CC(\text{vendor, customer}, CC(\text{customer, vendor, productDelivered, paid}, \text{productDelivered}) \ (\text{COM-01}) \text{ and } CC(\text{vendor, customer, paid, receiptSent}) \ (\text{COM-02}) \].

That is, the vendor promises the customer that it will:

- bring about a state in which the product is delivered if the customer promises that it will bring about a state in which the payment has been made once the product has been delivered \((\text{COM-01})\); and

- bring about a state in which the receipt has been sent if the customer brings about a state in which payment has been made \((\text{COM-02})\).

In summary, if the customer agrees to the offer, the vendor will deliver the product, the customer will pay, and the vendor will send the receipt.

After considering the quote, the customer decides to accept the offer. That is, the customer agrees to bring about a state in which payment has been made, if the product is delivered. Thus, it creates the conditional commitment:

\[ CC(\text{customer, vendor, productDelivered, paid}) \ (\text{COM-03}) \].

As this matches the condition of the vendor’s first conditional commitment \((\text{COM-01})\), \text{COM-01} becomes the commitment:

\[ C(\text{vendor, customer, productDelivered}) \ (\text{COM-04}) \] .

In the next step (step 4), the vendor fulfills its commitment to bring about a state in which the product is delivered by delivering the product\(^6\). Thus, \text{COM-04} is discharged. As the product has now been delivered, the condition of \text{COM-03} is now satisfied and it becomes the commitment:

\[ C(\text{customer, vendor, paid}) \ (\text{COM-05}) \].

The customer must now fulfill \text{COM-05} by bringing about a state in which payment has been made. Thus, it pays the vendor and \text{COM-05} is discharged. Furthermore, bringing about a state in which payment has been made satisfies the condition of \text{COM-02}. Therefore, \text{COM-02} becomes

\[ C(\text{vendor, customer, receiptSent}) \ (\text{COM-06}) \].

\(^6\)Delivering the product creates \text{COM-02}, but this redundant in this particular sequence as \text{COM-02} already holds.
CHAPTER 2. BACKGROUND

1: Request Quote

2: Send Quote

+ COM-01: CC(vendor, customer, CC(customer, vendor, productDelivered, paid), productDelivered)
+ COM-02: CC(vendor, customer, paid, receiptSent)

3: Accept Quote

+ COM-03: CC(customer, vendor, productDelivered, paid)
+ COM-04: C(vendor, customer, productDelivered)
* COM-02: CC(vendor, customer, paid, receiptSent)

4: Deliver Product

- COM-04: CC(customer, vendor, productDelivered, paid)
+ COM-05: C(customer, vendor, paid)
* COM-02: CC(vendor, customer, paid, receiptSent)

5: Make Payment

- COM-05: C(customer, vendor, paid)
- COM-02: CC(vendor, customer, paid, receiptSent)
+ COM-06: C(vendor, customer, receiptSent)

6: Send Receipt

- COM-06: C(vendor, customer, receiptSent)

**Key**

+ Create Commitment
- Discharge Commitment

* Commitment Carried from Previous Step

Figure 2.5: NetBill Protocol Interaction Diagram with Commitments
The vendor sends the receipt and is able to discharge this last commitment. As there are no remaining commitments, the interaction is at an end.

This example shows one particular path through a commitment-based interaction, however, there are many different paths through the interaction, as shown in Figure 2.6, a finite state machines representation of the possible action sequences for the NetBill protocol. As can be seen, the finite state machine in Figure 2.6 shows many paths through the interaction. These paths are not typically what would be produced if a designer used a finite state machine approach directly.

In this example, the customer agrees to pay if the vendor agrees to deliver the product. For the interaction to proceed, one of the agents must take a risk. That is, the customer can send payment and risk the vendor not delivering the product, or the vendor can deliver the product and risk the customer not paying.

Further work on commitment machines includes an improvement in which agents are able to reduce the risk they face in interactions [Yolum and Singh, 2007]. This improved approach is based on the monotonic concession protocol and, in summary, the agents are able to take small iterative incremental risks as opposed to taking one large risk.

Although the commitment machines approach is suitable for creating more flexible and robust
interactions than current message-centric approaches, it has a number of disadvantages. Commitment machines have only been applied to a few small examples and it is not clear if they are applicable to larger or more realistic interactions. Additionally, it is unclear what software tool support exists for the facilitation of creating interactions based on commitment machines.

Another disadvantage is the lack of mature design processes for creating agent interactions using commitment machines. Little work has been done on this aspect of commitment machines. The work in [Yolum, 2005] describes a number of protocol conditions to be checked and provides algorithms to check these conditions. A methodology for the design of commitment machines has been recently presented [Winikoff, 2006] (along with a process for mapping commitment machine designs to a collection of plans [Winikoff, 2007]). However, this is only an initial methodology and it has not been applied to a wide range of examples. Furthermore, this methodology begins interaction design with a Prometheus-style scenario, which is a sequence of ordered steps. This tends to result in designs that are constrained and do not exploit well the flexibility and robustness that commitment machines are able to achieve.

Social Commitments

The work of Flores and Kremer [2004a;b], is another approach to social commitments, however, the notion of commitment is slightly different to that of commitment machines. A social commitment is an agreement between two agents in which one agent is responsible for the performance of a certain action for the other agent. Note that the debtor does not necessarily have to perform the action itself; it is only responsible for that action being performed, whether it performs it itself or employs another agent to perform it.

As with commitment machines, the agents progress through the interaction through the attainment, manipulation, and discharge of commitments. The agents manipulate commitments through communicative acts with specified conversational tokens. The defined conversational tokens are:

1. Propose: Used to propose the uptake or discharge of a social commitment.
2. Accept: Used to accept the uptake or discharge of a social commitment.
3. Reject: Used to reject the uptake or discharge of a social commitment.
4. Counter: Used to reject a previous proposal and to suggest the consideration of another proposal, i.e. a counteroffer.
5. Inform: Used to communicate data between agents.

Using these conversational tokens, the agents are able to perform negotiation of social commitments. As such, message sequences emerge from the interaction, as opposed to message-centric interactions, in which message sequences are explicitly defined.

A third commitment-based approach to agent interactions is the work of Fornara and Colombetti [2002; 2003]. As with the previous approaches, the social commitments are utilized to drive
the interaction. However, in this body of work, the commitments are defined as an abstract data type, the commitment class, which can be instantiated into a commitment object. The commitment abstract data type consists of a number of fields (such as debtor, creditor, state, content, and condition) which describe the properties of a commitment and a number of methods (such as make, cancel, reject) which are used to manipulate it. A complete list of fields and methods, including detailed descriptions, is available in [Fornara and Colombetti, 2002].

Both the work of Flores and Kremer, and Fornara and Colombetti are similar to commitment machines in that they are based on social commitments. As such, the operation of these approaches are similar to the commitment machine examples previously shown. Although these commitment-based approaches allow for complex, flexible and robust interactions, they have a number of common disadvantages. That is, given a particular interaction, it is not obvious what commitments are required to create a commitment-based interaction.

Landmark-Based Approach

In the landmark-based approach [Kumar et al., 2002b;a; Kumar and Cohen, 2004], a landmark represents a particular state of affairs and agent interactions are represented by a set of partially ordered landmarks. Agents navigate through the landmarks to reach a final desired landmark, that is, a desired state of affairs. To proceed from one landmark to another, the agents must communicate with one another.

Figure 2.7 presents a visual representation of the landmarks approach. The initial landmark, \( L_1 \), depicted by two concentric hexagons, represents the entry point into the interaction. Solid hexagons, such as \( L_2 \) and \( L_4 \), represent important intermediate landmarks. Optional landmarks, such as \( L_3 \), are depicted as hexagons with dashed borders. Final landmarks, such as \( L_5 \) and \( L_6 \), are shown as hexagons with dark borders.

As can be seen, it is similar to a finite state machine in which the landmarks represent states, however, instead of specifying state transitions, the directed lines show the ordering of the landmarks. For example, \( L_1 \) occurs before both \( L_2 \) and \( L_4 \), and \( L_2 \) occurs before \( L_3 \), and so on. The indicated ordering is partial as some landmarks, such as \( L_2 \) and \( L_4 \), do not have an explicit ordering and could occur in any order. Additionally, there are optional landmarks which agents may
opportunistically skip. Agents follow a particular path by performing actions in one landmark to advance to the next. The performed actions can be either single communicative acts or complex actions which consist of several atomic actions.

It is important to note that the work expresses that the states of affairs are more important than the actions (i.e. communicative acts) that bring them about. Thus, as with the commitment-based approaches, the message sequences are not explicitly defined, but rather, they emerge as the agents communicate in an attempt to reach a final desired state of affairs.

The landmarks approach is theoretical in nature and has a heavy reliance on expertise in modal and temporal logics, which practicing software engineers may not have. Although an implementation, STAPLE, has been mentioned, there have been no further details apart from the publication of two posters [Kumar et al., 2002a; Kumar and Cohen, 2004].

Pre-Commitments

As previously discussed, social commitment approaches can be used to create more flexible and robust interaction protocols as the social commitments provide a basis from which message sequences emerge. However, one limitation of the approach is that the agents deal with pre-specified social commitments.

The work of Pham and Harland [2007] proposes an approach in which agents can interact by specifying their own social commitments through pre-commitments. That is, social commitments emerge from pre-commitments and internal commitments, and message sequences, in turn, emerge from the social commitments. Thus, this approach should provide even greater flexibility and robustness in interactions than social commitment approaches.

A pre-commitment is more a fundamental form of a social commitment in that it is a potential commitment that an agent is willing to commit to. Agents are able to negotiate over these pre-commitments and once the agents agree upon the pre-commitments, they then become social commitments which the agents are to satisfy.

The work of Pham and Harland [2007] suffers from the same problems as other social commitments approaches. That is, there is a lack of design guidance and, additionally, the work is theoretical in nature. In the case of the work of Pham and Harland [2007], knowledge of temporal linear logic [Pham et al., 2007] is required. All of these drawbacks entail that the work is not pragmatic and not suitable for practicing software engineers.

Goal-Plan Approach

More closely related to the research in this thesis is the goal-plan approach of Hutchison and Winikoff [2002], in which interactions are realized using the plans and goals of BDI agents. The work proposed a process to translate a message-centric protocol to a set of goals and plans.

The work can be seen as a predecessor to this research. However, although a design process was outlined, it was not detailed and there is no mapping from design to implementation. Further
to lacking a clear design process, as with the aforementioned approaches, the goal-plan approach has not been integrated with any existing full agent system design methodologies.

Although this section describes alternative approaches to traditional message-centric design, there is not much discussion about how to design agent interactions in the presented approaches. This is due to the lack of design processes and methodologies with these alternative approaches. The approaches focus on novel ways in which more flexible and robust agent interactions can be represented and achieved, but as yet, they do not focus on how the interactions can be designed using these novel approaches. In the previously described approaches, the lack of design processes and methodologies is a recurring disadvantage and limitation. In fact, this lack of design processes is a key motivation for the research carried out in this thesis.

So far, the alternatives to the traditional message-centric approach presented have drawn on interaction elements at the individual agent level. However, other alternatives exist that are based on designing flexible and robust interactions at the societal level. These include the design of electronic institutions, and social design with concepts such as norms, obligations and social laws.

**Islander**

Islander [Vasconcelos et al., 2002a; Esteva et al., 2002], an approach to electronic institutions, focuses on the macro-level (societal) aspects of multi-agents systems, rather than the micro-level (agent level). Electronic institutions are similar to their human counterparts in that they regulate what interactions can occur between agents. More specifically, an electronic institution defines a number of interaction properties, such as what interactions can occur, which agents can and cannot interact and under what circumstances these interactions can take place. In Islander, this is achieved by a global protocol which specifies the interactions between all components of the system. The Islander approach also has a software tool\(^7\) [Esteva et al., 2002] for the creation of electronic institutions (with work on semi-automatic agent development [Vasconcelos et al., 2002b]).

An Islander electronic institution has four basic elements: dialogic framework, scenes, performative structure, and norms. The dialogic framework provides a common ontology and defines permissible illocutions (i.e. communicative acts). The definition of valid illocutions structures the interaction and a common ontology is important as it ensures that agents, which may have differing internal languages and ontologies, are able to communicate. The dialogic framework also defines roles which agents can adopt within the institutions.

A scene is defined as a multi-agent dialogic activity. In an institution, there may be multiple distinct and possibly concurrent scenes. All agent interactions occur within scenes, each of which has its own protocol. The scenes are modeled as finite state machines with some additional modifications.

The performative structure captures the relationship between scenes, which allows for more

\(^7\)http://e-institutor.iiia.csic.es/e-institutor/islander/islander.html
CHAPTER 2. BACKGROUND

complex activities. Agents are able to navigate through scenes but they are constrained by rules which define the relationship between the scenes. The performative structure is represented by a graph of scenes and an agent is able to participate in multiple scenes at the same time.

The norms place consequences on the agents depending on the actions which they have carried out. For example, some actions may place commitments or obligations on an agent to perform a future action, others may limit the paths which the agent can take through the performative structure.

As Islander focuses on the societal aspects of multi-agent systems, it is different to Hermes in that it does not aid the designers to develop individual agents, but rather entire societies. Furthermore, Islander is suited to particular types of applications and is not really needed in closed systems.

OperA

Similarly to Islander, OperA (Organizations per Agents) [Dignum, 2004; Dignum and Dignum, 2003] takes a macro-level view of agent systems and focuses on agent societies rather than individual agents. The motivation for OperA is that most existing agent-oriented methodologies design agents from the individual agent perspective, however, a wider perspective, such as a societal-level one, is required to design agent societies as a society, i.e. not just a collection of individual agents interacting together. Furthermore, some societal-level goals cannot be captured as a collection of individual agent goals. Capturing societal-level goals allows for the analysis of societal characteristics, which is a motivation for OperA.

OperA is a model and a design methodology for creating such agent societies. The OperA model itself consists of three interrelated models: Organizational, Social, and Interaction [Dignum and Dignum, 2003]. The OperA design methodology [Dignum, 2004] facilitates the design of these models for system designers. The methodology consists of processes to aid in designing each of the three models.

The organizational model uses four structures, Social Structure, Interaction Structure, Normative Structure, and Communicative Structure, to specify agent societies.

The objectives of the agent society, roles and models governing coordination are specified through the social structure. The interaction structure provides partial ordering which are used to specify interactions between roles. Societal norms are specified in the normative structure in terms of roles and interaction norms. The communicative structure is used to specify ontologies for the interactions.

The methodology aids in designing an organizational model which reflects the desired structure of an agent society. The organizational model describes a number of aspects such as the roles, and different scenes. However, these roles must be filled by particular agents and the scenes must be acted out by the agents.

The social model describes the agents in the society. In particular, it describes the capabilities
and responsibilities of the agents.

In the interaction model, possible interactions between the agents in terms of interaction contracts, such as interaction protocols, are specified.

The OperA methodology does not include the design of individual agents. Instead, any appropriate methodology, such as Prometheus, Tropos, Gaia, etc., can be used to design the individual agents.

Norms, Obligations, and Commitments

Other less closely related work to this research is the body of work which focuses on the societal level of intelligent agents [Lopez y Lopez et al., 2004] and attempts to improve social interactions by use of norms, obligations and commitments [Fasli, 2003]. Although not directly related, many of these concepts are applicable to agent interactions and communications.

Some of the work involve work on norms [Conte et al., 1999], including the communication of norms [Castelfranchi et al., 1999], and allowing for social reasoning and incorporating norms and obligations into the BDI interpreter loop [Dignum et al., 2000; Broersen et al., 2001].

2.4 Agent-Oriented Software Engineering

A number of agent concepts for the construction of agent systems have been explained in the previous sections. However, to be able to practically design these systems, software engineering methodologies are required. Such methodologies provide design processes, along with guidelines and heuristics, which will assist with design choices and will support the development of these systems. These methodologies are extremely important and usually some degree of support for a number of the phases of the software development life cycle is provided.

One obvious approach is to use existing software engineering methodologies, such as object-oriented methodologies, to develop multi-agent systems. This is particularly desirable as it allows for re-use of a methodology that many practicing software engineers are familiar with. Additionally, object-oriented methodologies have been in use for quite a while and as a result, they have been well studied and developed, and there are a number of software support tools to aid in the design and implementation.

However, even though there are commonalities between objects and agents, the differences between them are substantial enough to warrant specific agent-oriented methodologies for the design of multi-agent systems. For example, agents are proactive and goal-oriented entities while objects are reactive and passive, and object-oriented methodologies do not provide support to model these characteristics.

The need for specific agent-oriented methodologies has given rise to an area in the agents field known as Agent-Oriented Software Engineering (AOSE). AOSE, as can be expected from its name, covers all aspects of software engineering related to agent systems, including analysis,
design, implementation and software tool support.

A number of different agent-oriented methodologies are present in the literature [Bergenti et al., 2004; Henderson-Sellers and Giorgini, 2005], some of which are general purpose and others are domain-specific. Some of the more well-known general purpose methodologies are: Gaia [Wooldridge et al., 2000; Zambonelli et al., 2003], MaSE [DeLoach et al., 2001; DeLoach, 2006], Tropos [Mylopoulos et al., 2000; Bresciani et al., 2004; Giorgini et al., 2004] and Prometheus [Padgham and Winikoff, 2004]. One example of a domain-specific methodology is SODA [Omicini, 2000]. SODA is a full agent design methodology which treats interactions as first class entities. It, similarly to other agent methodologies, uses a message-centric design process for agent interactions as it focuses on resources and information exchange between agents. However, unlike general purpose methodologies, SODA is specifically intended for the analysis and design of Internet-based systems, not for generic agent systems.

The Gaia methodology [Wooldridge et al., 2000; Zambonelli et al., 2003] allows for the analysis and design of multi-agent systems both at macro, i.e. societal, and micro, i.e. individual agent, levels. The methodology is general in that it does not favour any particular domain or agent platform for implementation. Gaia does not cover the requirements capturing phase of the software development life cycle as it considers it to be independent of the paradigm used for analysis and design. Furthermore, Gaia does not have a detailed design phase in which the internals of individual agents are developed.

The MaSE (Multiagent Systems Engineering) methodology [DeLoach et al., 2001; DeLoach, 2006], similarly to Gaia, is a general purpose methodology for the development of multi-agent systems. The methodology is intended to aid designers to analyze, design and implement a multi-agent system from a set of system requirements. MaSE is also able to design organization-based multi-agent systems [DeLoach, 2006] and is supported by a software tool, agentTool [DeLoach and Wood, 2001]. It is important to note that MaSE views agents as a specialization of objects and as an abstraction which does not necessarily possess intelligence.

Tropos [Mylopoulos et al., 2000; Bresciani et al., 2004; Giorgini et al., 2004] is a general purpose methodology for the design and implementation of multi-agent systems. Unlike Gaia, Tropos supports the requirements phase of software development. In fact, Tropos has two requirement-related phases: early requirements and late requirements. The heavy focus on the requirements phase is one of the key ideas of Tropos and is intended to understand how the implemented system will meet organizational goals. That is, Tropos not only captures what a particular system does or how it does it, but it also captures why the system is developed. This concept is not new, it was first proposed in requirements engineering [Dardenne et al., 1993; Yu, 1996]. Indeed, Tropos is based on Yu’s i* model [Yu, 1996].

In contrast to MaSE, Tropos agents ascribe to the mentalistic notions of agency. Agent concepts and related mental attitudes are used to guide the designer throughout all the different phases of the methodology.
The software support tool, TAOM4E (Tool for Agent Oriented visual Modeling for the Eclipse platform)\(^8\), is available for Tropos.

The Prometheus methodology [Padgham and Winikoff, 2004] was chosen to be integrated with Hermes due to local expertise and initial compatibility. Both methodologies share a goal hierarchy which was used as an initial starting point in the integration of the two methodologies. In the following section, the Prometheus methodology is presented with particular focus on its approach to interaction design.

2.4.1 Prometheus

Prometheus [Padgham and Winikoff, 2002; 2004] is a general purpose multi-agent design methodology which includes tool support [Thangarajah et al., 2005; Padgham et al., 2005a]. It is a complete methodology in that it covers system development from system specification through to implementation, along with some work on debugging [Padgham et al., 2005b] and maintenance [Dam et al., 2006]. In this section, the entire Prometheus methodology is briefly described in general. The interaction design aspects of the methodology are particularly focused upon and described in much greater detail as this thesis focuses on agent interaction and not full agent system design. The reader is referred to Padgham and Winikoff [2004] for more details on the Prometheus methodology.

The Prometheus methodology consists of three phases: System Specification, Architectural Design and Detailed Design. In the system specification phase, the actors, scenarios, actions, percepts, goals and roles of the system are identified, whilst in the architectural design phase, agent types are developed and their interactions, which will help to achieve the system goals, are specified. The detailed design phase focuses on designing the internals of the agents. As the next phase is implementation, the detailed design phase produces artifacts that are detailed enough to allow a direct implementation. Figure 2.8 depicts an overview of the Prometheus methodology. The interaction design processes are displayed as shaded boxes.

Although the focus of this thesis is on interaction design, it is necessary to understand the other processes in Prometheus as interaction design process is not independent of them. The interaction design process is affected by a number of other processes and design artifacts, and similarly it, in turn, affects other processes and artifacts. Thus, to better understand the Prometheus interaction design process, it is necessary to understand the entire methodology in general.

The development of a Prometheus design begins in the system specification phase with the designer analyzing a description of the desired system (as defined by stakeholders) or problem. This involves firstly identifying external entities\(^9\) (called actors) which will interact with the system in some manner. Once actors have been identified, the system designer progresses to identify key scenarios in which the actors interact with the system. After actors are associated with scenarios,

\(^8\)http://sra.itc.it/tools/taom4e/

\(^9\)External entities can be humans or other software systems.
Figure 2.8: Prometheus Overview Diagram (adapted from [Padgham et al., 2008])
the system designer continues to identify *percepts* (inputs to scenarios) and *actions* (produced by scenarios). At this point, the interface to the system has been defined in terms of percepts (inputs) and actions (outputs). These details are captured on an *Analysis Overview Diagram* (refer to Figure 2.9).

![Prometheus Analysis Overview Diagram](image)

*Figure 2.9: A Prometheus Analysis Overview Diagram*

In the next step of the methodology, the designer iterates between scenario development and development of a goal hierarchy. The specification of the scenarios is the first step in the interaction design process. A scenario is a sequence of steps that needs to be performed by particular roles in order to achieve a particular goal. It is important to note that a scenario shows only *one specific sequence of steps* and not the range of all possible sequences of steps. Typically, the scenario will show a successful sequence of steps and will describe possible variations (e.g., failures and alternatives).

For example, consider the scenario shown in Figure 2.10, in which a *Customer* role is attempting to purchase a certain product from a *Vendor* role. This scenario presents a particular sequence of steps in which everything progresses successfully and leads to the purchase of the product.

The *Steps* section of the scenario lists a sequence of steps that lead to the successful completion of this scenario. In the first step, the user initiates the interaction by requesting the *Customer* to find and purchase a product. In the second step, the *Customer* wishes to firstly ascertain that the *Vendor* has the product in stock before negotiating the price. This is listed as the *Customer*’s own individual goal (i.e., entered as “goal” in the *Type* column and named as “Check Availability”). In step 6, after the negotiation of the price has been successful, the scenario refers to a sub-scenario, named *Purchase Product*, which details how the agents will interact to purchase the product.
Name: Negotiate Price

Description: A Customer and Vendor negotiates the price of a product which the Customer wishes to purchase.

Trigger: Percept: Purchase Product

Steps:

<table>
<thead>
<tr>
<th>Step</th>
<th>Type</th>
<th>Name</th>
<th>Role</th>
<th>Description</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Percept</td>
<td>Request Purchase</td>
<td>Customer</td>
<td>User requests product to be purchased.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Goal</td>
<td>Check Availability</td>
<td>Customer</td>
<td>Customer requests availability of product.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Determine Availability</td>
<td>Vendor</td>
<td>Vendor determines availability of product.</td>
<td>Uses: Stock DB</td>
</tr>
<tr>
<td>4</td>
<td>Goal</td>
<td>Propose Price</td>
<td>Customer</td>
<td>Customer makes an offer to Vendor.</td>
<td>Uses: Finance DB</td>
</tr>
<tr>
<td>5</td>
<td>Goal</td>
<td>Evaluate Proposed Price</td>
<td>Vendor</td>
<td>Vendor determines if proposed price acceptable.</td>
<td>Uses: Cost DB</td>
</tr>
<tr>
<td>6</td>
<td>Scenario</td>
<td>Purchase Product</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Variations: Step 3: Product is not available. Terminate interaction.
Step 5: Proposed price too low. Re-propose higher price.

*Figure 2.10: Negotiate Price Scenario*
Exceptions to the norm or foreseeable failures of the typical scenario are listed as variations to the scenario. In Figure 2.10, two variations are shown. In the first, if the product is not available (step 3), then the interaction cannot progress and is terminated. In the second, if the price is too low (step 5), then the Customer is able to re-propose a higher price.

There is always a goal associated with each scenario. By using a number of heuristics and refinements, more goals and sub-goals are elicited. These goals are then grouped and arranged into a goal overview diagram in which goals are arranged in a direct acyclic graph as shown in Figure 2.11.

As can be seen, the goal overview diagram displays the composition/decomposition relationship between the goals. The relation between the goals are either AND or OR branches. An OR branch specifies alternative ways in which the parent goal can be achieved whilst an AND branch specifies that each sub-goal is part of the parent goal and must be achieved to achieve the parent goal.

\[
\begin{align*}
\text{Sell Books} & \rightarrow \text{Set Price} \quad \text{AND} \quad \text{Update Inventory} \\
& \quad \text{AND} \quad \text{Determine Competitive Price} \\
& \quad \text{AND} \quad \text{Process Order} \\
& \quad \quad \text{AND} \quad \text{Get Book Details} \quad \text{AND} \quad \text{Accept Payment} \\
& \quad \quad \quad \text{AND} \quad \text{Obtain Payment Details} \\
& \quad \quad \quad \text{AND} \quad \text{Send Shipment} \\
& \quad \quad \quad \quad \text{AND} \quad \text{Obtain Delivery Details} \\
& \quad \quad \quad \quad \text{AND} \quad \text{Arrange Delivery} \\
& \quad \quad \quad \quad \quad \text{AND} \quad \text{Obtain Payment Details}
\end{align*}
\]

\textbf{Figure 2.11: A Prometheus Goal Overview Diagram}

Once the goal overview diagram has been created, roles, i.e. a coherent “chunk” of behaviours, are determined by grouping related goals together.

In the architectural design phase, roles are grouped together to form agent types. It is not always obvious how to group the identified roles into agent types. Furthermore, given the subjectivity of design, there are no definitive “right” designs. As such, the designer is advised to group roles based on coupling and cohesion, and to also consider the relationship of roles to data. To as-
 sist with the grouping of roles, an Agent-Role Grouping diagram (referred to as “Agent Grouping” in Figure 2.8) can be used.

To help check that roles have been reasonably grouped, an agent acquaintance diagram can be used. These diagrams display the communication or data links between agents; if the agents are heavily coupled (e.g. all agents communicate with all other agents), then it may be worthwhile to reconsider the design.

Agent descriptors (refer to Figure 2.8) are used to describe each identified agent type present in the system.

The first step in Prometheus interaction design is to create an interaction diagram from a scenario (refer to Figure 2.8). An interaction diagram provides a graphical view of the interaction showing the agents involved and the inter-agent messages being passed between them as described by the scenario. The interaction diagram, like a scenario, depicts one instance of an interaction, not all the possible interactions.

Creating an interaction diagram from a scenario is straightforward. Firstly, the names of the agents involved in the interaction are placed in a rectangle with a life line at the top of the diagram. Actors are represented similarly to agents but a dashed rectangle is used instead.

The next step is to determine where communication occurs. Communication includes percepts (from actors to agents), messages (from agent to agent), and actions (from agents to actors). Communications are depicted by as labelled directed lines between the corresponding actors’ and agents’ life line. A message is labelled as “message name”, whereas a percept is labelled as “>percept name<” and an action is labelled as “<action name>”. A useful heuristic is to consider whether a communication is required whenever a step in the scenario is followed by a step which is assigned to a different agent or actor. In the case of the scenario shown in Figure 2.10, a communication is needed between each step, since each of these involve a different agent or actor.

In addition to identifying where messages are required, the designer also needs to give the messages names, which are displayed along the arrow indicating that a message is sent.

Figure 2.12 shows an interaction diagram for the scenario (refer to Figure 2.10).

![Figure 2.12: Negotiate Price Interaction Diagram](image-url)
Interaction protocols are created by generalizing interaction diagrams to capture all of the legal sequences of messages of an interaction and are described using a variation of the AUML notation [Huget and Odell, 2004] (refer to Section 2.3.1). Interaction protocols are derived from interaction diagrams by considering alternative messages that might be sent at each point. For example, instead of Vendor replying with Proposed Price Accepted, what happens if it sends a Proposed Price Rejected message instead? More generally, the interaction designer should not just consider alternative responses, but also other continuations of message sequences. The scenario variations are usually a useful starting point in identifying such alternatives and continuations.

In the case of the proposed price, there are two possible continuations, as depicted by the first Alternative box in Figure 2.13. If accepted, the Customer and Vendor can progress to the sub-scenario, Purchase Product. If rejected, then the Customer will have to propose a higher price to the Vendor. To re-propose a price, the agents will have to repeat steps 4 and 5 from the scenario. This is specified on the interaction protocol by using an AUML Continuation.

The system overview diagram (refer to Figure 2.14) is an important design artifact from the architectural design phase. As the name suggests, the system overview diagram provides an overview of the entire multi-agent system. The diagram shows the interaction between agents, specifying which protocols are used in these interactions, which agents require access to which data, and which agents handle which messages and percepts, and which agents perform which actions.

The last part of the Prometheus interaction design process is in the detailed design phase and requires the creation of process diagrams from the interaction protocols. Process diagrams form part of interaction design as they follow on from splitting an interaction protocol into processes that the agents can use to realize their interactions. Thus, unlike the previous two diagrams in the interaction design, which showed interactions between agents, process diagrams deal with the internals of single agents. Specifically, they show the internal processing of agents upon receiving messages. Typically, each agent in an interaction will have a process diagram for its processing associated with that interaction. Figure 2.15 shows a process diagram for the Vendor agent.

Messages are used as triggers in process diagrams. For example, when the Vendor receives a Request Product Availability query, its Determine Availability process is triggered. That process, in turn, triggers either its Send Available or its Send Unavailable process. These processes will appropriately send a Product Available or Product Unavailable message as shown in Figure 2.15.

Along with the process diagrams, design diagrams (such as agent overview diagrams) and various descriptors (i.e., forms), are created in the detailed design phase of Prometheus (refer to Figure 2.8). The capability descriptors and capability overview diagrams define and describe the capabilities that the agents have. These capabilities allow an agent to be decomposed into smaller behavioural parts. The capabilities are often based on the roles that were grouped to create the agent. By allowing the agents to be decomposed into smaller capabilities, a number of advantages are afforded, such as modularity and re-use of the capabilities.
Figure 2.13: Negotiate Price Interaction Protocol
Figure 2.14: A System Overview Diagram
Figure 2.15: Process Diagram for Vendor Agent in the Negotiate Price Interaction Protocol
The agent overview diagrams present an overview of an individual agent type. These include the capabilities and the data that the agent contains, the messages and percepts that the agent type handles, and the messages that the agent sends. The agent overview diagrams are similar to the system overview diagram of the architectural design phase.

The capability overview diagrams provide an overview of individual capabilities. These are similar to agent overview diagrams in that they show the data the capabilities contain, what messages and percepts they handle, what messages they send out, and plans or sub-capabilities they contain.

Capability, event, data, and plan descriptors are all created in the detailed design phase of Prometheus. These descriptors are typically textual descriptions of their namesake elements presented in a tabular format. Of particular importance is the plan descriptor, which contains procedural details that, if detailed enough, may be implemented directly.

2.5 Implementation Platforms

There are a number of agent development platforms which are based on the BDI model, such as JACK [Busetta et al., 1999], Jadex [Pokahr et al., 2003; Braubach et al., 2004], JAM [Huber, 1999], and Jason\(^\text{10}\). These agent development platforms generally provide support for the development, debugging and deployment of multi-agent systems. Some also support the design, as well as programming, of multi-agent systems.

There are many commonalities between the aforementioned agent platforms as they are based on the BDI model. In this section, these commonalities are described in general. As our work is implemented on the Jadex platform, which adds BDI functionality to JADE (a non-BDI agent platform), we provide specific details on it.

Implemented BDI agents consist of beliefs, goals, and plans. The objectives of the agents are defined as goals, which most platforms model as events. These events are used to direct the agents to achieve particular objectives by triggering the appropriate plans. For example, if an agent has a goal of purchasing a particular product, an event might be used to trigger the agent to perform the appropriate actions.

Agents are equipped with a library of plans which are written by the developer and are descriptions of how to achieve a goal. The plans are usually hierarchical, with the top ones being abstract. The plans in the agent’s plan library that are able to achieve a particular goal or respond to a particular event are relevant for that goal or event (refer to Figure 2.16). However, not all will be applicable to achieve the given goal or event in the current context. If plans that are both relevant and applicable are found, one plan is selected for execution. The process for plan selection varies from platform to platform. Some platforms select the plan based on priorities placed by the implementor whilst others use meta plans created by the implementor for plan selection. In any

\(^{10}\text{http://jason.sourceforge.net} \)
case, once a plan is selected, its plan body is executed. The plan body contains a number of steps (which may include the achievement of sub-goals) which when completed successfully will achieve the goal.

![Diagram of Plan Selection Process](image)

*Figure 2.16: Plan Selection Process*

For example, consider an agent attempting to achieve a goal to travel to a particular location. From its plan library it will select relevant traveling plans, such as walking, driving, and using public transport. From these plans the agent must select applicable plans. For instance, the walk plan may only be applicable if the agent is a short distance from the destination and it is not raining, and the driving plan may only be applicable if the agent has a working vehicle at its disposal. These are stated in the plans’ context conditions.

From the applicable set, the agent selects one plan and instantiates it (refer to Figure 2.16). If the plan is successful, the agent will have achieved its desired goal. However, if the plan is unsuccessful, the agent fails in achieving its goal and will then select another applicable plan (if one exists) in order to re-attempt to achieve the goal. This cycle will continue until either the agent has achieved its goal or it has exhausted all possible applicable plans, in which case the goal fails as there is no other alternative.

The work in this thesis is suitable to be implemented on any agent development platform which supports goals and plans in the manner of the BDI tradition. However, as the work is implemented on the Jadex development platform, the next section provides more details on Jadex.

### 2.5.1 Jadex

The Jadex BDI Agent System\(^\text{11}\) is developed at the University of Hamburg and it is an extension of JADE\(^\text{12}\) (Java Agent DEvelopment Framework). JADE itself is an agent development platform,

\(^{11}\text{http://vsis-www.informatik.uni-hamburg.de/projects/jadex/}\)

\(^{12}\text{http://jade.tilab.com/}\)
however, it does not cater for mental attitudes of intelligent agents. The Jadex extension provides facilities to implement BDI agents in a mixture of XML (eXtensible Markup Language) and Java. It should be noted that Jadex is a pure Java API and not an extension to Java. This provides a number of advantages, such as developers not being required to learn a new programming language and being able to use a wide range of existing integrated development environments (IDEs) for Java.

In Jadex, an intelligent agent is represented by an Agent Definition File (ADF) and a set of plans [Pokahr et al., 2006]. The ADF is specified in XML and it represents the initial state of the agent, i.e. its current goals, beliefs, running plans and library of plans.

In the ADF, the agent’s beliefs, goals and plan instantiation conditions, i.e. which goals, events or belief changes trigger particular plans and under what conditions the plans are applicable (context conditions), are specified. An (abbreviated) example of an ADF for a Customer agent is shown in Listing 2.5.

```
<agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="http://jade.sourceforge.net/jade.xsd"
       name="customer" class="jadex.runtime.JadeWrapperAgent"
       package="edu.rmit.cs.Ecommerce">

  <!-- Imports -->
  <imports>
    <import>jadex.planlib.*</import>
  </imports>

  <!-- Plans -->
  <plans>
    <plan name="productAvailable">
      <constructor>new ProductAvailable()</constructor>
      <filter>ProductAvailable.getGoalEventFilter()</filter>
      <precondition>$event.goal.available</precondition>
    </plan>
  </plans>

  <!-- Beliefs -->
  <beliefs>
    <belief name="role" class="String">
      <fact>"Customer"</fact>
    </belief>
  </beliefs>

  <!-- Goals -->
  <goals>
    <achieve goal="priceAccepted">
      <parameter name="product" class="String" />
      <parameter name="price" class="int" />
    </achieve>
  </goals>

</agent>
```

Listing 2.5: Jadex Agent Definition File

The beliefs are simply represented as XML tags within the ADF itself. The beliefs do not use any special knowledge representation. Instead, arbitrary Java objects are able to be stored as beliefs.

As with the beliefs, the goals are also specified by XML tags in the ADF. There are a number of different types of goals in Jadex, such as achieve, maintain, perform, and query. The achieve goal
is the most basic and is used when an agent wishes to achieve a particular state. The maintain goal is used when a particular state or condition needs to be constantly monitored and maintained. For example, an agent making monthly electronic repayments for its user will continue to make the repayments until the loan is completely repaid. A perform goal is used when particular actions are to be executed. The focus is on executing the actions, not the state which they bring about. A query goal is used to retrieve specific information.

The plan heads, i.e. instantiation conditions, are specified in the ADF, however, the plan body, i.e. the steps which the agent needs to carry out to achieve its goal, is implemented as a Java class. This approach allows plan bodies to be re-used in different agents. An example of a plan body for an achievement plan to request the availability of a product is shown in Listing 2.6.

```java
public class RequestAvailability extends ThreadedPlan {
    public static GoalEventFilter getGoalEventFilter() {
        return new GoalEventFilter("requestAvailability");
    }

    public void body() {
        // Synchronize with Coordination plan
        waitFor(createCondition("beliefbase.inDetermineAvailabilityStage", MCondition.IS_TRUE));

        RBeliefbase beliefset = getBeliefbase();
        RAbstractGoal goal = ((RGoalEvent) getInitialEvent()).getGoal();
        String merchantName = (String) (getBeliefbase().getFact("merchantName"));
        String msgContents = "Determine Availability;" +
                            beliefset.getFact("product") + ":";

        // Request product’s availability from Merchant
        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
        msg.addReceiver(new AID(merchantName, AID.ISGUID));
        msg.setContent(msgContents);
        sendMessage(createMessageEvent(msg));
    }
}
```

Listing 2.6: Jadex Plan Body

As can be seen in Listing 2.6, the plan bodies are written in pure Java code and the Jadex API has a number of classes which are useful. Most plans in Jadex inherit from the Plan or ThreadedPlan provided as part of the Jadex platform. Furthermore, the Jadex API provides a number of other classes to facilitate implementation of multi-agent systems, such as the ACLMessage class which provides a representation of ACL messages (refer to Listing 2.6).

A plan is considered to have been successfully achieved if no Java exceptions are produced. The plan classes (Plan or ThreadedPlan) provide methods such as passed() and failed(), which can be overridden, to handle plan success or failure and are executed on plan success or failure respectively.

Goal failure handling can be specified using the BDI flag attributes of the goals. These include attributes such as retry and retrydelay which can be set for each individual goal. The retry attribute is used to enable BDI-like behaviour of goal, that is, if a plan fails to achieve the goal, another applicable plan is selected in an attempt to achieve the goal.

Jadex also provides a number of software support tools [Pokahr et al., 2005] to ease the development process. The Introspector is particular useful for debugging as it allows the system
implementor to see the agent’s beliefs, goals and plans at runtime.

2.6 Summary

Existing literature has been reviewed and background material required to appreciate and understand the research carried out in this thesis has been presented in this chapter. The concepts of agents and intelligent software agents has been explored and for the purposes of this work, an intelligent software agent has been defined as a software entity with the following characteristics: situated, autonomous, proactive, reactive, social, flexible and robust. It has also been stated that the agents discussed in this work are of the BDI tradition.

A survey of the different aspects of social ability has been presented. This includes agent communication languages, speech acts, and different types of agent interactions.

Agent interaction design, including the traditional and alternative approaches have been discussed. The traditional approaches usually express the interactions in notations such as AUML, Petri nets and finite state machines. The design process is message-centric and the resulting interactions are limited in terms of flexibility and robustness.

Alternative approaches explored include a number of commitment-based approaches, a landmark-based approach, and a goal-plan approach. Other related work discussed were electronic institutions, such as Islander, and work on improving the social ability of agents by use of norms, obligations and commitments.

Agent-Oriented Software Engineering (AOSE) and a number of general agent design methodologies, such as Gaia, MaSE, Tropos, and Prometheus, along with SODA, a specialized agent design methodology which treats interactions as a first class entity, were discussed. In particular, a detailed description of Prometheus was presented as the work in this thesis is integrated with it.

Finally, a discussion on implementation platforms was presented. The discussion described the commonalities of agent development platforms based on the BDI model of agency, such as PRS, dMars, JACK, Jam, Jadex and Jason. Specific details on Jadex were then presented as the research in this thesis is implemented in Jadex.
Chapter 3

Designing Goal-Oriented Interactions †

In this chapter, we present the design aspect of the Hermes methodology. The contributions of this chapter are:

- A design process that guides the designer to create goal-oriented interactions from an initial high level description of an interaction through to a design which can be implemented on goal-plan agent platforms, including steps for identifying and handling failure.

- Failure handling mechanisms which increase the flexibility and robustness of the goal-oriented interactions.

- Notation for capturing and modeling key goal-oriented design artifacts.

Existing notations, such as AUML, and some design processes in existing methodologies, such as Prometheus, are based on UML. Although UML is suitable for object-oriented design, agent-oriented design, especially the approach taken by Hermes towards interaction design, is quite different to object-oriented design. AUML is heavily based on the messages exchanged between agents; an approach which Hermes avoids. Thus, Hermes has been developed independently from these notations and approaches to try and ensure that it is not influenced by or biased towards them.

This chapter is structured as follows. Section 3.1 presents an overview of the Hermes methodology. The subsequent sections progress through the design process showing how particular goal-oriented design artifacts are developed. Parallelism in interactions is described separately in Section 3.7.

†Part of the work presented in this chapter has been previously published [Cheong and Winikoff, 2005; 2006a; 2009].
A simplified version of the NetBill [Sirbu and Tygar, 1995] protocol is used as the basis for the design of the interaction in this chapter. This protocol has been commonly used in work related to flexibility and robustness in agent interactions [Winikoff et al., 2004; Yolum and Singh, 2001; 2002]. Although the protocol, as shown in Figure 3.1, is fairly constrained, it provides a good basis for expository purposes.

![Figure 3.1: Simplified NetBill Protocol](image)

### 3.1 Methodology Overview

An overview of the Hermes methodology is shown in Figure 3.2. As can be seen, the methodology follows an incremental mini-waterfall model, a progressive approach in which each step is derived from prior steps. Changes in one step affect not only subsequent steps, but also preceding steps, as is typical of design methodologies. Furthermore, it is also an iterative process which encompasses both the design of the interaction and the design of the agent internals.

The methodology is divided into three phases, as shown in Figure 3.2. The division, based on the three core aspects of designing an interaction in Hermes, allows designers to determine what phase they are currently in and adjust their thinking to that particular phase.

The first two steps fall into the first phase, the *Interaction Goal Hierarchy Design* phase. In this phase, the designer is focused on the overall design of the interaction. The designer is concerned with *what* the interaction is to achieve and *who* (i.e. which roles) are involved in the interaction. Thus, Hermes begins by identifying roles and interaction goals as they are required before one can develop actions in the second phase. The identified roles and interaction goals are simply captured in a list. Afterwards, in the second step, these interaction goals are organized into an *Interaction Goal Hierarchy*, which is the final\(^1\) artifact produced by this phase.

---

\(^1\)A *final* design artifact is defined as a non-intermediate artifact that is retained for design documentation purposes. In some steps of Hermes, *intermediate* artifacts are created to either provide a logical path which will guide the designer from one step to another or to allow the designer to check the created design. These artifacts are not intended to be retained as design documentation.
CHAPTER 3. DESIGNING GOAL-ORIENTED INTERACTIONS

1. Roles and Interaction Goals
2. Interaction Goal Hierarchy
3. Action Maps
4. Action Sequence Diagrams
5. Action Message Diagrams
6. Message Descriptors

Key
- Final Design Artifact
- Intermediate Design Artifact
- Derives/Feedback
- Crosscheck

Figure 3.2: Hermes Methodology Overview Diagram
The second phase, the *Action Map Design* phase, requires the designer to think about *how* the roles involved can achieve the interaction goals identified in the previous phase. As such, actions which the interacting roles will need to carry out are identified in step 3 and are organized into appropriate execution sequences. In step 4, these execution sequences are tested for validity. The final artifacts produced by this phase are the *Action Maps* (resulting from step 3), which define possible sequences of actions executed by the roles to achieve the interaction goals. There may also be intermediate *Action Sequence Diagram* artifacts (resulting from step 4), which are used to ensure that the sequences in the action maps are sufficient to allow the roles to achieve the interaction.

In the last phase, the *Message Design* phase, the designer’s attention shifts from actions to communications between the roles, i.e. messages, as they are required to complete the interaction definition. Step 5 requires the designer to identify where messages are required to be exchanged between roles, while step 6 calls for the designer to define what information the messages will contain. The final artifacts from this phase are the message definitions, which will vary depending on whether the designer is using platform-specific message types or standards, such as KQML, FIPA or SOAP. The message definitions are recorded in *message descriptors*.

The three phases purposefully divide not only the design process, but also the design artifacts in a coherent, modular and logical fashion. The logical division of the process and artifacts have been explained in the preceding paragraphs. The design artifacts themselves are coherent; for example, the interaction goal hierarchy contains all information relevant to what interaction goals need to be achieved. The action maps are also coherent as they only contain information relevant to what actions need to be executed in order to achieve the interaction goals.

As the designer progresses through the design phases, the interaction becomes less abstract and more concrete. The interaction goal hierarchy design phase guides the designer to consider the goals of the interaction and what needs to be achieved in order for the interaction to be successful. The action map design phase allows the designer to determine how these interaction goals can be achieved. This is done by creating actions that the interacting roles can execute to achieve the interaction goals. In the message design phase the designer determines what the roles will need to communicate to each other so that the interaction can be successfully achieved.

As with some message-centric approaches, one of Hermes’ advantages is that its design is modular and one is able to easily exchange design artifacts. For example, an action map which achieves an interaction goal, *Sell Item*, by following an English auction can easily be exchanged for one which follows a Dutch auction. Furthermore, the hierarchical nature of the interaction goal hierarchy allows for interaction goals, or even branches of the hierarchy to be exchanged with other interaction goals or branches of another hierarchy. Thus, this promotes reusability of existing designs.

The following sections explain each of the phases and steps of the design process in detail and provides an example of how a design is created in Hermes.
3.2 Interaction Goal Hierarchy

The interaction goal hierarchy is primarily used as a coordination mechanism which is common to the roles involved in an interaction. Goal-trees, which are effectively goal hierarchies, have been used in a number of agent-oriented methodologies such as MaSE [DeLoach et al., 2001] or Prometheus [Padgham and Winikoff, 2004] and provide a number of desirable properties. The interaction goal hierarchy provides a good overall visual description of the interaction at a glance. As the hierarchy is structured and the relationship between goals and sub-goals are shown, it can help to determine why a particular goal is being achieved. Alternatively, it can also help to determine how goals are realized in terms of sub-goals that need to be achieved. A structured hierarchy also provides advantages such as composition and re-usability. Furthermore, the hierarchy naturally allows the designer to view the interaction at different levels of abstraction. For example, the top-most goal provides an overall idea of what the interaction is about whilst the leaf-node goals provide the finer details of what needs to be achieved to complete the interaction. Additionally, the goals in between the top-most goal and leaf-node goals, and the relationships between them, show why particular goals need to be achieved and, at times, why a particular goal achievement order is required.

The first step in creating the interaction goal hierarchy is to determine the roles involved in the interaction and the interaction goals which they are attempting to achieve. Consider an agent type, Academic Agent. This agent can take a number of different roles in different interactions. For example, in an academic paper reviewing interaction, the Academic Agent can undertake any of the following roles: Author, Reviewer, Editor, etc.

Therefore, a role usually represents a subset of what an agent can do and one agent is able to assume other roles in other interactions. Although possible, it is not usual for one agent to assume multiple roles in the same interaction. In fact, there may be rules preventing an agent from undertaking multiple roles in the same interactions. For example, in the aforementioned academic paper reviewing interaction, an agent cannot play both the roles of Reviewer and Author on the same paper.

The second step of developing the interaction goal hierarchy is to refine and organize the interaction goals identified in the previous step. Where possible, the interaction goals are broken down into smaller sub-interaction goals and are organized into a hierarchy. The hierarchy should only have one interaction goal at its apex, which captures the overall goal of the entire interaction.

As an example, consider an e-commerce scenario based on the NetBill [Sirbu and Tygar, 1995] protocol (refer to Figure 3.1), in which a Customer role is attempting to purchase a monitor online from a Merchant role.

The overall goal of this interaction is for the Customer and Merchant to trade cash and goods. Thus, the top-most interaction goal is Trade. The top-most interaction goal is always the most abstract goal in the hierarchy.

To continue developing the interaction goal hierarchy, the remainder of the identified interaction
goals are placed into the interaction goal hierarchy using decomposition relationships, that is, interaction goals are placed into parent-child relationships. It is often the case that the designer adds more goals at this point. This is usually because some interaction goals may have originally been overlooked or the designer has determined that particular interaction goals will need to be added for the interaction goal hierarchy to make more sense.

Some of these interaction goals that will need to be added will be obvious to the designer, however, the designer is able to use a number of approaches to identify the interaction goals. These include taking a top-down or a bottom-up or approach. The designer can also use a mixture of these two approaches.

In the top-down approach, the designer analyzes each existing interaction goal and determines if it can be decomposed into smaller, more concrete goals. Decomposition should only go as far as producing interaction goals that require more than one role to complete. If goals that require only one role to achieve them have been identified, the designer has decomposed the interaction goals too far, as the identified goals are individual agent goals and not interaction goals.

Taking a bottom-up approach requires the designer to identify and aggregate bottom-level, concrete interaction goals into abstract ones and progress up the interaction goal hierarchy. As previously mentioned, a bottom-level interaction goal should require more than one role to achieve it.

It does not matter whether the designer takes a top-down or bottom-up approach, or even a mixture of both approaches, as long as the interaction goals are identified. Another approach is for the designer to brainstorm and simply identify goals that are required for the interaction and then the designer can place these goals into the hierarchy. If Hermes is integrated with other full agent system design methodologies (which it is intended to be), this step is generally easier as the full agent system design methodologies usually have artifacts (e.g. scenarios and system goals in Prometheus) which guide the designer as to what interaction goals are required. Details of the integration of Hermes with a full agent system design methodology are presented in Chapter 4.

For example, by taking a bottom-up approach and analyzing the interaction to be designed, the designer will determine that interaction goals such as Order, Negotiate, Transfer Goods, Payment, and Send Receipt will be required. Continuing with the bottom-up approach, the designer will recognize that the following interaction goals, Order and Negotiate can be grouped into an interaction goal such as Agree as they are all interaction goals in which the Customer and Merchant are attempting to agree on the particulars of the trade. Similarly, the interaction goals Transfer Goods, Payment and Send Receipt can be grouped under an interaction goal such as Exchange. A suitable interaction goal hierarchy is shown in Figure 3.3. The undirected lines in Figure 3.3 denote parent-child or sub-goal relationships.

If a top-down approach had been taken, i.e. determining that Trade should be composed of two sub-interaction goals, Agree and Exchange, and that both Agree and Exchange should be composed of further sub-interaction goals, the designer could have arrived at a similar hierarchy
as the bottom-up approach.

In the interaction goal hierarchy presented in Figure 3.3, note that the top-most interaction goal is the most abstract. Progressing down the hierarchy, the interaction goals become less abstract and more concrete. This allows the roles involved to complete the bottom-level interaction goals, which are termed atomic interaction goals (as they are not decomposable into simpler interaction goals), in order to achieve the other interaction goals in the hierarchy, which are named composite interaction goals (as they are able to be decomposed into simpler interaction goals).

The lines from Trade to Agree and Trade to Exchange indicate that for the Trade interaction goal to be achieved, the Agree and Exchange interaction goals must be achieved. Furthermore, the Agree and Exchange interaction goals are achieved when their sub-goal are achieved. Thus, when the bottom-level goals (i.e. atomic interaction goals) are achieved, the entire interaction is completely achieved.

Once the designer has settled on an appropriate interaction goal hierarchy, temporal dependencies (depicted as directed lines in Figure 3.4) are to be added. The temporal dependencies provide an effective way for the designer to place constraints on the sequence of the interaction and, thus, restrict the order in which certain interaction goals can be achieved. For example, the directed line between Agree and Exchange depicts a temporal dependency between the two interaction goals and states that the Agree interaction goal must be achieved (successfully) before the Exchange interaction goal can start.

While temporal constraints are useful to restrict certain undesirable sequences of interaction
goal achievement from occurring, they should be used sparingly as the more temporal constraints are used, the less flexible the interaction. For example, the particular design shown in Figure 3.4 is a strongly constrained design (as it is based on the NetBill protocol and its constraints), however, alternative designs could, for instance, simultaneously transfer goods and make the payment, thus, relaxing some of the temporal constraints.

![Figure 3.4: Final Interaction Goal Hierarchy](image)

It is important to note that the directed lines represent *dependencies*, which are different to *causalities*. The dependency between *Agree* and *Exchange* shows that *Exchange* is dependent on *Agree*. That is, the *Exchange* interaction goal cannot be attempted until the *Agree* interaction goal has been successfully achieved.

A causality would be different in a number of ways. Firstly, the direction of the line would be reversed; the line would originate from *Agree* and point towards *Exchange*. Secondly, a causality would state that the achievement of the *Agree* interaction goal causes the *Exchange* interaction goal, which is not quite correct. What one wishes to capture here is not that achieving *Agree* will cause the roles to move on to achieve *Exchange*, rather, the roles are free to achieve *Exchange* only when *Agree* has been achieved. This distinction is not apparent when the roles follow a straightforward sequence, however, it is more apparent when interaction goals have to be achieved out of sequence, especially in rollbacks (refer to Section 3.3.4). Thus, although the use of dependencies and, especially, the reversed directed line of the dependencies may appear to be initially counter-intuitive, dependencies do capture better the meaning of the relationship between the interaction goals.
Where temporal dependencies are placed will depend on the designer and the interaction itself. In general, they are placed to ensure that interaction goals are achieved in a sensible sequence. For example, it does not make sense for the roles to achieve the *Exchange* interaction goal before achieving the *Agree* interaction goal. Thus, a temporal dependency is placed between the two to ensure that *Agree* must be achieved before *Exchange*.

As part of developing the interaction goal hierarchy, the designer should also assign to each interaction goal the roles that are involved in that interaction goal. In this example, it is quite straightforward as there are only two roles involved, *Customer* and *Merchant*, and each interaction goal should have at least two roles assigned to it. Thus, both roles are involved in every interaction goal. In other interactions, this may not be the case, and the designer will need to decide, based on the interaction, which roles are involved in which interaction goal. For example, a *Bank* role may be involved in the *Payment* and *Send Receipt* interaction goals. In Figure 3.4, the roles involved are shown in the circles as $R: C, M$, denoting that a particular interaction goal involves the *Customer* and *Merchant* roles.

The designer must also identify an *initiator* for each interaction goal. The initiator represents the role which initiates and is initially responsible for a particular goal of the interaction. Providing an initiator for each interaction goal is necessary as it ensures that when each interaction goal is reached, at least one role has the initiative and will begin interacting in order to achieve the interaction goal.

Valid initiators are specified as one of the roles involved in a particular interaction goal (e.g. $C$ or $M$). In Figure 3.4, the *Merchant* is always responsible for initiating the *Transfer Goods* and *Send Receipt* interaction goals, whilst the *Customer* is always responsible for initiating the *Payment* interaction goal.

Sometimes, an interaction can be initiated (i.e. initiating the top-most interaction goal) by different participants. In such a case, we allow the initiator to be “unbound” (termed *inherited initiator* and denoted by $\uparrow$ in Figure 3.4), and whichever participant initiates the interaction at runtime is inherited by the lower interaction goals. For example, in Figure 3.4, either the *Customer* or the *Merchant* can start the interaction. If the interaction is started by the *Customer*, it is then responsible for starting the following atomic interaction goals: *Order* and *Negotiate*. The *Customer* will also be the initiator for *Agree*, an intermediate interaction goal, but this has no real significance. Similarly, if the *Merchant* is the interaction initiator, it would then be responsible for initiating those interaction goals.

If an atomic interaction goal has an inherited initiator, its action map must allow for at least two different roles to execute initial actions, otherwise a specific initiator should be identified for the interaction goal (refer to Section 3.3).

The interaction goal hierarchy provides an overview of the interaction and depicts what the interacting roles need to achieve in order to achieve the interaction.

Up to this point, only the common and coordination aspects of the interaction have been
CHAPTER 3. DESIGNING GOAL-ORIENTED INTERACTIONS

designed. The next step in developing a Hermes interaction is to consider the internal design of the agents. This is done by creating an action map for each atomic interaction goal. Each action map shows how its corresponding interaction goal is to be achieved. The development process of action maps is described in the following section.

3.3 Action Maps

The interaction goal hierarchy dictates what the roles need to achieve to complete the interaction. It is used as a common coordination mechanism between the roles involved in the interaction. In contrast, the action maps consider the internal designs of the participating agents and concern how the agents which play particular roles can achieve these interaction goals.

Hermes has been designed without adopting up-front the notations and concepts of existing notations such as UML. However, there is a degree of resemblance between Hermes action maps and UML activity diagrams. The similarity is primarily due to the purpose of the artifacts: they are both intended to model the flow or sequence of actions (in the case of action maps) or activities (in the case of activity diagrams). The core symbol for activity diagrams is the activity and its counterpart in action maps is the action. Both activities and actions depict the same thing: a process to be executed.

Activity diagrams allow for conditional branching by use of the diamond symbol. Branching is also supported in action maps; it is modeled as multiple causality links leaving an action.

Activity diagrams support parallelism by allowing the designer to fork and merge particular sequences of activities. Although parallelism is supported in action maps, it is not described here. Instead, parallelism in Hermes interactions is discussed in Section 3.7.

There are no notions of different types of activities in activity diagrams, however, actions in action maps must have a type (e.g. Independent, Caused, Final Caused, and Final Independent). This allows the designer greater control to specify action sequences, such as which actions are the entry and exit points into the action maps. A similar effect is achieved in activity diagrams by use of the start and end symbols, which allow designers to prescribe a fixed entry point into and exit point from an activity diagram. Note that an action map may have multiple entry points, whilst an activity diagram can only have one.

A further difference is that action maps not only depict the flow of actions, but also the flow of data between the roles. As such, it has a data store symbol to represent the source of data and the designer is able to specify where the data flows between roles and where it finally ends up. Activity diagrams, however, are only able to specify the flow of activities and have no concern for data flow.

An interaction goal is completed when all of its sub-goals are achieved, thus, only atomic (and consequently, concrete) interaction goals need to be achieved. Therefore, each atomic interaction goal from the interaction goal hierarchy will have one corresponding action map that defines all the different ways in which the interaction goal can be achieved.
An action is a discrete step towards achieving an interaction goal taken by a single role. Action maps capture these actions which are performed by roles involved in achieving a particular interaction goal.

By dividing the interaction such that each atomic interaction goal has one action map which includes all relevant roles, the design artifacts are being split in a different fashion to the approach of other methodologies, such as Prometheus, MaSE, and Tropos. In its detailed design phase, Prometheus divides its interaction protocol into process diagrams (one per agent). This is the general approach of methodologies which employ a message-centric approach to interaction design, such as MaSE and Tropos. This difference is quite significant, because although splitting interactions on a per agent basis appears logical, it can be problematic. An interaction is a multi-agent activity, that is, an agent’s action in an interaction is usually sensible in the context of that interaction relative to what actions other agents are undertaking. If the interaction is divided per agent, each agent’s individual activity is removed from the context of the interaction and may not make sense.

On the other hand, if an interaction is divided per interaction goal (and its relevant action map), this problem does not occur. This is chiefly due to the action map containing multiple agents interacting with each other to achieve a given interaction goal, i.e. a coherent subset of the entire interaction. For more details on the effects of this difference, refer to Section 6.1.

For ease of explanation, the action map development process is described in four distinct steps. However, it is not intended that designers rigidly follow these steps. The steps are only provided as a guide and designers are free to re-order step sequences, omit certain steps if they are deemed irrelevant or use multiple iterations if they believe that this would result in better designs.

The four steps are as follows:
1. Develop the initial action maps.
2. Add data to the action maps and consider data flow issues.
3. Generalize the action maps
4. Extend the action maps to handle foreseeable failures

The following sections explain each of the aforementioned steps.

### 3.3.1 Develop Initial Action Maps

The initial development of action maps is broken down into three steps:

1. Identify actions and assign them to roles involved in the interaction;
2. Establish action sequences by use of causality links; and
3. Identify the type of each action.
The first step in developing an action map is to divide it into “swim lanes” (one for each role involved in the interaction), identify actions and assign them to the interacting roles. For example, in the e-commerce interaction, there are only two roles involved, Merchant and Customer. Thus, Figure 3.5, an action map for the Negotiate interaction goal of the e-commerce example, requires two swim lanes: one for the Customer role and the other for the Merchant role.

Once swim lanes have been added to the action map, the designer will need to determine what actions need to be carried out by the participating roles in order to achieve the interaction goal. To identify required actions, the designer will need to consider the abilities of the relevant roles and the interaction goals they have to achieve. Once the actions have been determined, they are assigned to roles by placing them in the relevant role's swim lane.
CHAPTER 3. DESIGNING GOAL-ORIENTED INTERACTIONS

If Hermes is integrated with an full agent design methodology, there are usually design artifacts from those methodologies that can be used to guide the designer to identify the actions required. For example, in Prometheus, the Scenarios will aid the designer to identify these actions, whereas in Tropos, the Tasks will be of use to the designer at this stage.

Figure 3.5 shows an example of three possible actions for the Negotiate interaction goal: Request Price, Propose Price, and Consider Price. The Request Price and Consider Price actions are assigned to the Customer, whilst the Propose Price action is allocated to the Merchant. When the Customer executes the Consider Price action it needs to notify the Merchant of the acceptance of the price, thus, an additional action is required from the Merchant to receive the notification. In this case, the action is named Receive Decision, and as it is the last action to be executed in this action map, it is final action, i.e. an action which terminates an interaction goal (explained at the end of this section). Such actions are required to terminate action maps.

Once actions have been identified and assigned to the involved roles, the action execution order must be established. This is achieved by placing causality links between the actions. The causality links impose temporal restrictions and indicate the action flow of the action map, i.e. which actions can be attempted after an action has been executed. A causality link between Action A and Action B means that once Action A has been executed, Action B should then be executed.

Causalities, as opposed to dependencies (which are used in interaction goal hierarchies), are used in action maps as they “push” the execution of actions forward. That is, the execution of an action will cause the roles to execute other actions until a final action is executed.

To clarify the difference between causality and dependency, consider a situation in which there are three actions: Action A, Action B, and Action C. Action A causes Action C and Action B also causes Action C. In this case, Action C is triggered when either Action A or Action B complete because causalities are used. However, if arrows are viewed as dependencies, then Action C can only occur after both Action A and Action B have completed (as Action C depends on both)\(^2\).

It is possible to have more than one causality link extending from one action, in which case, a choice is implied. For example, if Action D has two causality links extending from it to Action E and Action F respectively, this implies that once Action D is completed, either Action E or Action F will be executed.

Causality links are not necessarily inter-agent; they can also be intra-agent. Furthermore, causality links are able to be labeled with conditions. This is useful to clarify the causality paths on the action map.

In Figure 3.5, inter-agent causality links have been placed between the following pairs of actions: Request Price and Propose Price, Propose Price and Consider Price, and Consider Price and Complete Interaction Goal.

\(^2\)Using parallel actions (refer to Section 3.7.2), it is possible to specify that two actions together cause another action to be executed.
Where to place causality links will depend on the designer and the interaction, and is usually common sense. For example, in the NetBill protocol, Propose Price must occur after the Request Price action, hence, a causality link is placed between the two. To ensure that all causality links have been identified, the designer should check each action against all other actions and ensure that the established sequence is sensible. This will involve checking that dependencies are correct (e.g. Consider Price cannot occur until the Merchant sends a price) and that all actions are reachable (i.e. all actions are connected by causality links).

The last part of the initial development of the action maps is to determine the action type of each action. The action types are needed to indicate which actions start and terminate the action maps, because it is necessary to allow for multiple start and end points. The different action types are:

- Independent Action
- Caused Action
- Final Caused Action

An independent action is one that can start without being triggered by another action, that is, it is not necessarily triggered by another action, but it may be triggered by another action. For example, consider a situation in which there are two independent actions, Action A and Action B, and there is a causality link extending from Action A to Action B. In such a situation, Action A can be triggered independently. Action B can also be triggered independently, or it can be caused by Action A. Typically, independent actions are used as entry points into the action maps. An independent action is denoted as a rectangle with a dashed border (refer to the key in Figure 3.5).

A caused action is one that must be triggered by another action. That is, a role cannot start executing a caused action until an action which triggers it is completed. Caused actions are represented as a rectangle with a solid line (refer to the key in Figure 3.5).

A final caused action, denoted by a rectangle with a thick solid line (refer to the key in Figure 3.5), is a caused action which terminates an interaction goal. Being a caused action, it must be triggered by another action. Furthermore, as a final action, it signifies that once it is executed, no further actions will be executed for that particular action map.

In the initial action map, refer to Figure 3.5, as previously discussed, the Receive Decision is a final action. Both the Customer’s Request Price action and the Merchant’s Propose Price action are initial actions. That is, either action can be the first action to be executed in this action

---

3Another possible combination of independent and final action type could be one called final independent action; an independent action that terminates an interaction goal. As a final independent action is both an independent action and a final action, it both starts and ends an interaction goal for a role. However, as Hermes action maps should involve at least two roles, an action which both initiates and terminates an interaction goal (i.e. an action map with involves only one roles) seems to be questionable, since it doesn’t involve any interaction. Thus, this action is not used.
map. This adds flexibility as either the Customer can request a price from the Merchant, or the Merchant can propose a price to the Customer without being prompted. The remaining action, Consider Price, is a caused action.

At the end of this step, a rudimentary action map is created. Notice that the presented action map, refer to Figure 3.5, is constrained as it is heavily based on the NetBill protocol. The subsequent steps will refine it into a more flexible, robust and complete action map.

3.3.2 Adding Data to Action Maps

This step involves identifying and adding data stores to the action maps. This is important as particular actions will require appropriate data. The designer must also ensure that data which the roles require is accessible. To identify the necessary data stores, the designer analyzes the actions carefully and considers what data is required for the actions to execute successfully. It is also useful to determine what data needs to be passed from one action to another. Once the data has been identified, data stores are placed in the swim lane of the role to which they belong. Note that only relevant data stores are displayed on an action map; not all the data stores that a role contains. This avoids unnecessarily cluttering the action maps.

For example, in Figure 3.6, the Customer will need to somehow keep track of how much money it has to spend. This is captured by its Funds data store. Similarly, for the Merchant to operate, it will need to know information such as the production cost of the products it sells which is represented by the Production Cost data store.

Simply adding data stores is not sufficient. The designer must ensure that actions which read and write data have direct access to the data stores. The designer must also ensure that all actions will have access to needed data even if the data store resides in another role. This may mean that required data is read from a data store and is passed to a particular action that requires the data. In order to ensure all these, the designer should consider for each action what data is needed, where the data will be obtained from, and where the data will (finally) end up.

For example, the Merchant’s Propose Price action needs to return the product name, along with the price, to the Customer because in this design, the Customer doesn’t store the product name. As such, the product name, which originated from the Customer’s Purchase List data store (which contains a list of all products the Customer is to purchase), is passed along the causality link between the Customer’s Request Price action to the Merchant’s Propose Price action and then passed onto the Customer through the causality link between the Propose Price and Consider Price actions. This is denoted in Figure 3.6 by use of the Note Indicators.

Dashed lines were chosen to represent the data flow from data stores to actions and actions to data stores as they differentiate the data flow from the control flow, i.e. the causalities, which are represented with solid lines, and avoids cluttering the action maps with solid lines. Where

\[\text{Note Indicators}\]

In practice, it is more sensible for the Customer to store the product name in a belief, however, in this design it is passed through the actions to illustrate how data is passed from action to action.
CHAPTER 3. DESIGNING GOAL-ORIENTED INTERACTIONS

Figure 3.6: Action Map with Data Stores
required, note indicators are used to clarify what data is passed along the causalities (refer to Figure 3.6).

In this step, data stores have been added and the correctness of data flow between actions has been ensured. The next step will generalize the action map to make it more flexible and more complete.

### 3.3.3 Generalizing Action Maps

In this step, the designer seeks to improve the action map by generalizing it, i.e. providing multiple ways in which the action map can be followed to achieve its corresponding interaction goal. There are two ways in which this can be done. The first is for the designer to add alternative paths to success in the action map. The second is for the designer to identify where problems can occur in the action map (i.e. an action fails) and provide failure handling for the foreseen problem.

Although these two approaches are both used and can be performed together, for ease of explanation, they have been divided into two different steps. In this step, the addition of alternative ways to successfully achieve the interaction goal is explained, whilst the following step (refer to Section 3.3.4) describes how to identify failures and how to handle them.

To generalize the action maps, the designer seeks alternative ways in which the action maps can achieve the interaction goal. The addition of alternative ways in which an action map can achieve its interaction goal improves the flexibility of the interaction as the roles will have more than one way to complete the interaction goal. Identifying appropriate places for adding alternative paths can be difficult as it is dependent on the domain, the roles involved and the actual interaction. However, although there are no set guidelines for identifying where alternative paths can be added, the designer can systematically analyze each action and determine if the action can be achieved in a different manner or if additional useful actions can be added.

For example, in the e-commerce interaction, when the Merchant sends a price, the Customer simply accepts the price (refer to Figure 3.6). However, it is possible that the Customer will reject the price. As such an alternative pathway is added to reflect this in Figure 3.7. The Customer is able to now reject the price, in which case the interaction will terminate unsuccessfully as it cannot proceed if a price is not agreed upon. Although this alternative pathway does not, at present, lead to a successful completion of the interaction goal, it will in subsequent steps.

Although this step appears simple in this example, in other designs this step can be quite involved. The action map may go through dramatic changes during this step and can result in a bigger and more complicated action map, especially if multiple alternative paths are added. Furthermore, the changes in this step will affect changes in succeeding steps, as shown in the following section.

---

An alternative to terminating the interaction is for the Customer to re-propose a price. However, this is not used here as it is used later on in the development of the action map to illustrate one way to handle failures.
Figure 3.7: Action Map with Generalizations
3.3.4 Adding Failure Handling to Action Maps

This step focuses on failures and how to attend to them. Failure handling is of crucial importance as it is what gives the action maps, and thus Hermes, the majority of their robustness\(^6\). By being able to handle foreseeable failures, the roles are able to persevere through these failures in order to complete the interaction.

To better demonstrate failure handling, we artificially modify the interaction goal hierarchy presented in Figure 3.4 at this point. Note that this modification is purely for expository purposes. The modified interaction goal hierarchy is presented in Figure 3.8. The difference between the modified (Figure 3.8) and original (Figure 3.4) interaction goal hierarchies is that in the modified version, an interaction goal, Negotiate Details, is introduced and the Negotiate interaction goal is renamed to Negotiate Price. The Negotiate Price interaction goal depends on the Negotiate Details interaction goal and, is exactly the same as the Negotiate interaction goal from the original interaction goal hierarchy. Thus, we continue the development of the action map (refer to Figure 3.7) corresponding to the Negotiate interaction goal as they are also exactly the same. In the Negotiate Details interaction goal, the Customer and Merchant negotiate over details of the product, such as colour.

![Figure 3.8: Modified Interaction Goal Hierarchy](image)

The different types of failures possible in Hermes are described next, followed by the available Hermes failure recovery mechanisms which can be used to address these failures. An explanation of how to determine and add failure handling to the action maps is then presented.

\(^6\)Additional robustness comes from the rollback mechanism which is discussed later on.
Failure Types and Failure Recovery Mechanisms

There are two types of failures in Hermes:

- action failure; and
- interaction goal failure.

An action failure is when an action fails to achieve its interaction goal or intended purpose. For example, if the Merchant proposes a price and the Customer rejects the price. An interaction goal failure is more dire. In such failures, the roles are unable to achieve the interaction goal. For example, the Customer rejects the Merchant’s proposed price and the Merchant is unable to offer a lower price. In this case, the Negotiate Price interaction goal fails as they cannot reach agreement.

To manage and address failures, Hermes offers three types of failure handling mechanisms:

- termination;
- action retry; and
- rollback.

Terminations and action retries can be used to handle action failures. A termination ends the entire interaction and can be used in situations in which re-attempting the failed action or pursuing an alternative course of action will not result in progressing through the interaction. For example, if the Merchant simply cannot sell the desired item because of a lack of stock and production of the item has been discontinued.

The concept of an action retry is simple: it allows a failed action to be recovered from by retrying that, or another, action. The interaction then proceeds as per normal, i.e. following the causality links, which may lead the roles to execute actions which they have previously executed. For example, if the Customer rejects the Merchant’s proposed price, the Merchant is able to retry its Propose Price action with a lower price instead of the interaction goal failing at this point.

If an action fails and is not able to be handled by action retries, this can lead to interaction goal failure. When this occurs, the interaction can be either terminated or rolled back to a previous interaction goal. The main notion of a rollback is that if an interaction is returned to a previous interaction goal and the interaction goal is achieved in a different manner (which leads to a different intermediate result than before), the failed interaction goal may then be successfully achieved.

Handling Failure in Action Maps

There are two parts to adding failure handling to action maps:

1. failure identification; and
2. addressing the identified failure by adding an appropriate failure handling mechanism.
Determining the appropriate failure handling mechanism will depend on the type of failure, as explained in the previous section. Thus, the following sections will address failure identification and adding the three types of failure handling mechanisms provided by Hermes: termination, action retry, and rollback.

**Failure Identification**

In order to identify where possible failures might occur, the designer should think about each action and determine whether it can fail or not. If the action can fail, the designer should determine what types of failures can result from it. Once failures have been identified, the designer should determine ways in which the failures can be addressed.

For each action map, the designer should create a table to summarize all the possible failures and ways to rectify them. For example, the Merchant’s Propose Price could fail, i.e. the proposed price is rejected by the Customer (refer to Figure 3.7). This can be rectified by the Merchant proposing a lower price. Table 3.1 shows the failure table for the Negotiate Price interaction goal.

In this simple example there is only one possible failure but there may be more than one possible failure in other cases.

To further enhance flexibility and robustness, the designer can also analyse each action and determine different ways in which it can succeed (i.e. determine alternative success paths). Once failures have been identified and remedial actions determined, the designer can then update the action maps with terminations, action retries and rollbacks.

**Action Retries**

Adding action retries to action maps is relatively straightforward. In the case of the identified failure and its proposed remedy (refer to Table 3.1), a single action is required to be added: Revise Price. The Revise Price action is placed between Consider Price and Propose Price (refer to Figure 3.9). Thus, if the Customer rejects the proposed price, the Merchant can execute the Revise Price action to retry the Propose Price action with a lower price. If the Merchant cannot offer a lower price, the interaction is terminated. The Revise Price action is, in effect, a decision node.

An effect of adding action retries is that it can lead to loops between actions. Note that a loop has now formed between the following actions: Propose Price, Consider Price and Revise Price (refer to Figure 3.9). It is important to ensure that there are no endless loops in action maps. This
CHAPTER 3. DESIGNING GOAL-ORIENTED INTERACTIONS

Figure 3.9: Action Map with an Action Retry

- **Customer**
  - Purchase List
  - Request Price
  - Consider Price
  - Price accepted
  - Price rejected

- **Merchant**
  - Production Cost
  - Propose Price
  - Revise Price
  - Terminate Interaction
  - Receive Decision

**Key**
- Independent Action
- Final Caused Action
- Data Store
- Data Flow
- Causality
- Note Indicator

- Includes Product Name
- Includes Product Name and Price Data
- Cannot offer lower price
- Offer lower price
is done by providing an exit condition: if Re-propose Price fails (i.e. the Merchant cannot propose a lower price), the interaction is terminated. Experienced designers will be able to immediately add such exit conditions but novice designers may not realize they are required. However, novice designers should be able to identify, in a second iteration through the action maps, that the Revise Price action could fail and, as such, this failure should be handled. In this case, it is handled by providing an action which will terminate the interaction.

Rollbacks

As previously explained, the purpose of a rollback is, when faced with an interaction goal failure, to re-attempt a prior interaction goal in the hopes that if the prior interaction goal is achieved in a different manner which produces different intermediate results, the failed interaction may be successfully achieved. Thus, it is necessary for the rollback to produce a different intermediate result than previously acquired (otherwise the interaction will enter an infinite loop). In order to ensure that the same result is not produced upon rollback, the implementation will need to keep track of all previous results, or in some cases, the last result (e.g. when making increasing offers). For more details on the implementation of rollbacks, refer to Section 5.4 (Chapter 5, page 132). If a different intermediate result is not able to be produced, the interaction will terminate as all possible alternatives have been exhausted.

To determine where rollbacks can be issued from, the designer should analyze each action individually and consider whether it is sensible to issue a rollback from it if it fails. The designer should pay careful consideration to termination actions as they can often be substituted with rollbacks. As a test, the designer should be able to clearly explain the purpose of issuing a rollback from a particular point in the interaction and what advantages it brings to the interaction. Once a potential rollback has been identified, the designer will need to determine what interaction goal the rollback will roll back to.

For example, in Figure 3.9, instead of terminating the interaction when the Merchant’s Revise Price action fails (i.e. the Merchant is no longer able to offer a lower price), the interaction can be rolled back to the Negotiate Details interaction goal. That interaction goal can then be achieved in a different manner, e.g. the Merchant suggests a different colour of the product which may lead to a lower price (which the Customer can afford). Thus, the Negotiate Price interaction goal will then be able to be achieved. If, however, the Merchant is unable to suggest a different colour, the interaction will terminate. To specify the rollback on the action map is simply a matter of replacing the Terminate Interaction action with Rollback To Negotiate Details, as shown in Figure 3.10.

The usual behaviour of a rollback is as follows. If the interaction is being rolled back to interaction goal \( IG1 \), then all interaction goals which depend on \( IG1 \) are to be retried. For example, if for some reason while achieving the Payment interaction goal, the interaction needs to roll back to Negotiate Price (e.g. the price of the product exceeds the Customer’s credit card limit and the Customer would like to re-negotiate the price), Negotiate Price itself and Exchange
Figure 3.10: Action Map with Rollback
would need to be re-achieved. Re-achievement of *Exchange* involves re-achieving *Transfer Goods*, *Payment* and *Send Receipt*. In this particular design, all these interaction goals need to be re-achieved as the interaction goal hierarchy is strongly constrained.

The rollback approach is different to that of trying alternative plans to achieve a goal in the BDI architecture. The rollbacks do not capture alternative ways of achieving an interaction goal at the interaction goal hierarchy level, whereas the BDI approach does capture alternative ways (i.e. plans) in which a goal can be achieved. Furthermore, rollbacks are more precise than BDI failure as they specify a particular target interaction goal in the hierarchy to which the interaction returns to directly and re-attempts. This provides flexibility from the hierarchical structure, unlike the BDI approach which requires all alternatives for a failed goal to be retried before proceeding to a prior (i.e. higher-level) goal in the hierarchy. Consider an example in which a bottom-level goal fails in a hierarchy and the top-most goal needs to be re-achieved. Using the rollback approach, it is possible to directly re-attempt the top-most goal, however, in the BDI approach, all alternatives for all the intermediate goals have to be retried and failed so that the top-most goal is re-attempted (assuming there are alternative ways to achieve it).

A similar approach to the Hermes rollback is ASKIT’s backtracking approach [Yoshimura et al., 2003]. In ASKIT, a composite service can be seen as a team of agents in which each member provides a service. ASKIT keeps track of dependencies between steps of the plan (to provide a composite service) and if a particular step fails, it and its dependencies are marked as “not executed”. The team member providing that particular service is replaced, the plan is re-executed and only the unexecuted steps are attempted.

As the behaviour of rollbacks is explicitly coded during the implementation phase (refer to Section 5.4), it can be customized as necessary. The default behaviour of a rollback is to re-achieve all dependent intermediate interaction goals. It is desirable for each rollback to maintain a list of dependent interaction goals which should *not* be re-achieved, i.e. a list of “exceptions”. This is a more concise approach than maintaining a list of all dependent interaction goals to be re-achieved. If a list of exceptions is not provided, then all dependent interaction goals are re-achieved.

Interaction goals at which rollbacks can be issued and which interaction goals can be rolled back to is both domain- and application-specific. Therefore, it is up to the designer to determine this. The designer must thus indicate whether rollbacks are permissible for each interaction goal, and if so, which interaction goal should the interaction be allowed to roll back to. Additionally, the designer specifies for each rollback a list of interaction goals that should *not* be re-achieved. This gives the designer the flexibility to skip completed interaction goals which do not need to be re-achieved.

One constraint of rollbacks is that in order to be able to roll back to a previous interaction goal, the current interaction goal must be dependent on the interaction goal which it desires to roll back to. That is, for interaction goal *B* to roll back to interaction goal *A*, *B* must depend on
A (as A must occur before B in order for the rollback to make sense).

**Terminations**

Similarly to rollbacks, where and when an interaction can be terminated is also domain- and application-specific. As with the rollbacks, the designer will have to carefully analyze and consider each action map and determine whether it is sensible to terminate an interaction at that point. For example it is not sensible for the e-commerce interaction to terminate after the goods have been transferred but before payment has been made.\(^7\)

Terminations can be identified at points in the interaction where no alternatives are possible (i.e. all possible alternatives have been exhausted) and essential particulars of the interaction cannot be agreed upon. Terminations are usually placed at points that “make or break” the interaction. If the roles involved cannot agree on a particular of the interaction and there are no alternatives, then the interaction simply cannot proceed.

Terminating in response to failure provides a graceful exit from an interaction which cannot be successfully achieved. As such, when a termination occurs, all roles involved in the interaction should leave the interaction in a desirable state. For example, in the e-commerce interaction, when a termination occurs, both parties should leave the interaction without incurring any loss. It would be undesirable for the *Merchant* to have transferred the goods and for the *Customer* to not proceed with the payment.

**Completed Action Map with Failure Handling Mechanisms**

After the final iteration of this step, the action map (refer to Figure 3.10) is now in a completed state. It is also more flexible and robust than the initial action map developed in Figure 3.5.

### 3.4 Action Sequence Diagrams

Action sequence diagrams are simple and minor Hermes artifacts which can be used to check action maps. These artifacts are optional and are to be used at the designer’s discretion.

An action sequence diagram follows a specific trace from an action map. It is different from action maps, which show all possible execution sequences, as an action sequence diagram shows one specific sequence of actions being executed.

Action sequence diagrams are similar to UML sequence diagrams. The role names are placed in a box which have life lines extending downwards, along which time progresses. Actions that are executed by the roles are placed on their life line. These actions use the same type-based notation as used in the action maps, e.g. a final action is denoted by a thick border. Actions that are carried out to achieve a particular interaction goal are enclosed in a shaded box which

---

\(^7\)Unless one adds compensatory actions, in this case, to return the product.
represents that interaction goal. Figure 3.11 shows a particular example of an action sequence
diagram showing the Negotiate Details and Negotiate Price interaction goals.

![Figure 3.11: Action Sequence Diagram](image)

To develop an action sequence diagram, the designer traces through the action maps and
interaction goal hierarchy, and makes appropriate choices of what action is executed at particular
points. That is, the designer simulates an execution of the interaction.

Action sequence diagrams cut across interaction goal hierarchies and action maps as by fol-
lowing the action sequence diagram development process, the designer is able to clearly see which
actions are used to achieve particular interaction goals. This gives the designer a better under-
standing of the “mechanics”, i.e. the workings, of the designed interaction.

The purpose of an action sequence diagram is to check that identified actions from the action
maps are sufficient to allow for complete and successful interactions. It also allows the designer to ensure that specifically desired interactions can be generated by the interaction goal hierarchy and its associated actions. Furthermore, the action sequence diagrams can be used to show typical interactions and identify possible interaction failures. They can also be used to ensure that action retries and rollbacks execute as desired, as they can simulate these failure handling mechanisms by allowing the designer to systematically step through the actions to be executed.

3.5 Messages

In this phase of the interaction design, messages need to be identified. These messages are necessary to realize inter-agent triggering of action/causeality links as defined in action maps. To identify the messages, the designer will need to analyze the action maps and determine where one role needs to trigger an action of another role or where data needs to be transmitted from one role to another.

Part of defining messages will also involve determining the data carried by the messages. A message between the Merchant’s Propose Price and the Customer’s Consider Price actions will obviously need to contain a price. The way the data is represented will depend on the message standards being used, which, in turn, will depend on the intent of the implemented interaction. For example, if the implemented interaction is to be used in open systems, then standards such as KQML, FIPA, or SOAP might be appropriate. If the implemented interaction is to be used in a closed system, then the default message type of the agent platform being implemented upon might suffice.

As there is such a wide variety of ways in which messages will be formed, and since the Hermes philosophy itself does not place great importance on the messages, Hermes does not have strong guidelines for the format of messages. This allows for great flexibility in the definition of messages, both at the design and implementation phases. However, the identification of messages and the contents the messages carry are important and Hermes does provide some guidelines for these.

To identify where messages are required, the designer should analyze the action maps. Whenever an action by one role is followed by an action from another role, a message is required, assuming that both actions are within the same interaction goal. There is no need to have messages between interaction goals, as moving between interaction goals is handled (automatically) by the coordination plans (refer to Chapter 5).

Consider the causality link between the Merchant’s Propose Price action and the Customer’s Consider Price action (refer to Figure 3.10). For this causality to be realized, there will need to be a message sent from the Propose Price action to the Consider Price action.

An understanding of the data flow of the action maps is important to be able to determine the data contents of a message. During the course of designing the action maps, the designer has considered the data flows between the actions and has likely recorded them through annotations on the action maps. The origin and destination of data is from either data stores or actions. This
information is used to determine what data is being transmitted from actions that read or write to data stores. For example, the Merchant includes the price in the message between its Propose Price action and the Customer’s Consider Price action.

Data which is repeatedly transmitted through a series of actions might be easily overlooked, thus, the designer must pay careful attention to such data. For example, it is obvious that a suggested price must be part of the message sent by the Merchant’s Propose Price action to the Customer’s Consider Price action. It is also obvious that the Customer’s reply message will require a positive or negative response. However, it is not obvious that the Customer will have to re-transmit the suggested price as the Merchant will need to know which price the Customer is responding to and, in this design, the Merchant does not store this information. Furthermore, if the Customer and Merchant are not in agreement, the Merchant will need those details to propose another price.

Messages are defined in message descriptors, which are based on the Prometheus message descriptors. Table 3.2 shows the different fields in an example message descriptor, along with a description and an example for each field.

### 3.6 Action Message Diagrams

As with action sequence diagrams, action message diagrams are simple and minor Hermes artifacts. Similarly, they are optional and are used for checking interactions. In this case, they are used to ensure that identified messages are adequate to complete the interaction successfully.

Action message diagrams are developed by adding messages to action sequence diagrams. Action message diagrams can also be useful in identifying what data needs to be carried in the message. When the messages are placed between the actions, the designer can consider what data needs to be communicated between the roles. For example, in Figure 3.12, the message between the Customer’s Propose Details action and the Merchant’s Consider Details action will need to
carry across data such as the colour of the monitor.

![Action Message Diagram](image)

**Figure 3.12: Action Message Diagram**

Note that in the action message diagrams, both the message type and the contents (the key-value pairs in parentheses in Figure 3.12) are shown. As action message diagrams show a particular interaction, the contents of the messages are necessary. The message types help to clarify the diagram and also allows the designer to refer to the appropriate message descriptor if required.

### 3.7 Parallelism in Interactions

So far, parallelism has not been discussed, however, parallelism in interactions is clearly important. This is especially true in multi-agent systems in which multiple agents are able to interact and execute in parallel. This section explores the modeling of parallelism in goal-oriented interactions.

There are a number of different types of interaction parallelism, which can be categorized as
shown in Figure 3.13. The broadest level of distinction between different types of interaction parallelism is \textit{intra-agent} and \textit{inter-agent} parallelism. \textit{Intra-agent} parallelism refers to the possibility (but not the necessity) of attempting more than one part of the interaction in parallel internal to a role. For example, an agent playing a role is able to pursue more than one interaction goal from the same interaction goal hierarchy in parallel (\textit{interaction goal parallelism}) or the agent is able to carry out an action while carrying out another action from the same action map in parallel (\textit{action parallelism}). In \textit{inter-agent} parallelism, instances of agents that play a particular role, which we henceforth refer to as \textit{role fillers}, carry out the interaction (or parts of the interaction) in parallel with each other, but there is no parallelism internal to the agent instances.

![Interaction Parallelism Taxonomy](image)

\textit{Figure 3.13: Interaction Parallelism Taxonomy}

As previously mentioned, \textit{interaction goal parallelism} and \textit{action parallelism} are two types of intra-agent parallelism that are specific to Hermes. Interaction goal parallelism is suitable when large parts of an interaction can be carried out in parallel, whilst action parallelism is suitable when smaller specific parts of an interaction can be carried out in parallel. Depending on the particular interaction, it may be possible to have combinations of both interaction goal and action parallelism. As Hermes allows for both interaction goal and action parallelism, the designer has substantial control over the design of parallel interactions.

There are two types of inter-agent parallelism, \textit{heterogeneous instance parallelism} and \textit{homogeneous instance parallelism}. Heterogeneous instance parallelism occurs in interactions in which different types of role fillers are interacting in parallel. This is quite common in, and inherent to, multi-agent systems since agents, which fill the roles involved in the interactions, execute au-
tonomously. For example, in the previous e-commerce interaction, the Customer and Merchant agents are different types of role fillers which are running and interacting in parallel. Modeling such interactions can be done using the action map notation outlined earlier in this chapter.

Homogeneous instance parallelism, however, is more interesting and somewhat more difficult to model. It occurs in interactions in which multiple role fillers of the same role type are interacting in parallel. One example of such a situation is multiple Bidder role fillers in an auction.

This section discusses these types of interaction parallelism. Sections 3.7.1 and 3.7.2 discuss interaction goal parallelism and action parallelism respectively. Homogeneous instance parallelism is discussed in further detail in Section 3.7.3. Heterogeneous instance parallelism is not discussed further as it can be modelled with the Hermes design notation discussed in the preceding sections.

It should be noted that the parallelism aspect of this body of the work is less developed than the work presented in the earlier part of this chapter. It is an initial attempt to capture parallelism in goal-oriented interactions and will require further work and refinement.

3.7.1 Parallel Interaction Goals

In this section, a notation to explicitly highlight parallelism in the interaction goal hierarchy is discussed. This notation is somewhat redundant as, if there are no constraints between interaction goals, they can be executed in parallel. However, it is sometimes useful to be able to specify explicitly where parallelism occurs in the interaction goal hierarchy (refer to the following example).

When modeling parallelism, it is necessary to specify the start and end point of the parallelism. This is easily possible with the hierarchical nature of the interaction goal hierarchy and the action maps. An obvious way of adding parallelism in the interaction goal hierarchy is to specify explicitly interaction (sub-) goals to be achieved in parallel. The parallelism can be specified by marking an interaction goal thereby denoting that its sub-goals are to be achieved in parallel. Given the hierarchical organization of interaction goals, this marks both the start and end points of the parallelism. For example, in Figure 3.14, a goal interaction hierarchy which involves the construction of a table, the Acquire Parts interaction goal serves both as the start and end point of the parallelism. As with any composite interaction goal, in order for the Acquire Parts interaction goal to be achieved, its sub-goals must be achieved. Thus, the agents will attempt to achieve the Manufacture Table Top and Purchase Table Legs interaction goals in parallel as specified by the parallelism symbol (refer to Figure 3.14).

As earlier stated, the parallelism notation (refer to Figure 3.14) is redundant as interaction goals with no dependencies are executed in parallel by default. That is, even if the designer did not specify that Manufacture Table Top and Purchase Table Legs should be achieved in parallel, they can be achieved in parallel as they have no dependencies. However, the parallelism symbol is useful as it can be used to highlight intended parallelism.

When and where to use interaction goal parallelism will depend on particular interactions. Identifying parallel interaction goals is usually obvious as, according to our parallel notation, they
must not have any inter-dependencies and must always share a common (direct) parent interaction goal. That is, the interaction goals to be achieved in parallel must be siblings in the interaction goal hierarchy.

This approach to interaction goal parallelism appears to impose a limitation that an interaction goal’s sub-goals must all either be executed in parallel or in sequence. However, it is possible to work around this issue by using intermediate interaction goals. For example, consider an interaction goal $A$, which has three sub-goals, $B$, $C$, and $D$. $A$’s sub-goals can all either be executed in sequence or in parallel. To mix sequential and parallel execution, an intermediate interaction goal is required. For example, an intermediate interaction goal, $CD$, which has $C$ and $D$ as sub-goals, could be created and made dependent on $B$ (refer to Figure 3.15). Thus, this allows for the mixing of sequential and parallel interaction goal achievement.

No changes are required for failure handling with interaction goal parallelism. Action failures are not affected as the parallelism lies within the interaction goal hierarchy. It is possible to roll back to individual parallel interaction goals (e.g. Manufacture Table Top or Purchase Table Legs from Figure 3.14) or their parent goal (e.g. Acquire Parts). For example, if the Assemble Parts interaction goal fails, then the agents can rollback to Acquire Parts, which is re-attempted as per normal. However, if the agents decide to roll back to an individual parallel interaction goal (e.g. Purchase Table Legs), then it, along with its parent goal, is to be re-achieved, but not its
Figure 3.15: Interaction Goal Hierarchy with a Mixture of Sequential and Parallel Interaction Goals

Sibling parallel goals (e.g. Manufacture Table Top), assuming its sibling goals have already been successfully achieved. The parent interaction goal, Acquire Parts, is unachieved until its child interaction goal, Purchase Table Legs, is achieved.

Rolling back from a parallel interaction goal to another interaction goal is more complicated as sibling parallel interaction goals may need to be terminated because they may depend on something that is being re-achieved differently. For example, consider a situation in which the Build Table interaction goal (refer to Figure 3.14) depends on another interaction goal such as, say, Negotiate Table Details and the interacting roles have agreed on a four-legged table. However, the role attempting to achieve the Purchase Table Legs interaction goal is only able to secure three legs and wishes to roll back to Negotiate Table Details to determine the possibility of building a three-legged table. This would then require the Manufacture Table Top interaction goal, which is being achieved in parallel to Purchase Table Legs, to be rolled back.

A possible issue that may come up during execution of parallel interaction goals is that messages used to achieve one interaction goal may be confused with messages used to achieve another interaction goal in parallel. However, this is not limited to parallel interaction goals, as even if an agent is using sequential interaction goals, the agent may be involved in more than one interaction at a time. This issue is not part of the design and it is up to the implementor to ensure that there is no overlap between messages from different interaction goals. This can easily be addressed by
simple schemes such as labeling each message with interaction goal IDs (similar to FIPA ACL’s conversation identifiers).

### 3.7.2 Parallel Actions

As with interaction goals in the interaction goal hierarchy, actions can be executed in parallel in the action maps. However, unlike the interaction goal hierarchy, in which the start and end point of parallel sub-goals can be specified in one place, at their parent interaction goal, the beginning and end of the parallelism in action maps need to be specified at different places. This is due to the different structures of the two artifacts (i.e. the interaction goal hierarchy is hierarchical whereas the structure of the action maps is not).

Consider a meeting scheduling interaction in which there are three agents: an Organizer, a Participant, and a Venue Manager. In this interaction, the agents must first determine a date and time at which the Organizer and Participant can meet and the Organizer must also ensure that a room is available from the Venue Manager to hold the meeting. Once availability has been determined, the Organizer confirms the meeting and room booking. Figure 3.16 shows the action map for the Order interaction goal\(^8\).

In this particular interaction, there are a number of actions that can be carried out in parallel. The start and end of action parallelism is denoted by the parallel symbol (parallel double bars) as shown in Figure 3.16. The parallel symbol denotes the synchronization points in the action map. That is, the action before a parallel split must prepare the interaction to allow for the start of parallelism whilst the action after a parallel merge prepares the interaction to end the parallelism.

A parallel split changes a singular action sequence to multiple parallel action sequences (signified by the single directed line entering the parallel bars and multiple directed lines leaving it). A parallel merge is the opposite, that is, it changes the interaction from multiple parallel action sequences into a single action sequence (e.g. when both the Participant and a room are available, the parallel action sequences checking their availability are merged into a single action sequence). However, there are situations in which the interaction will need to revert to a single action sequence without all the parallel action sequences merging at the same time.

An example of this is when failure occurs in one of the parallel action sequences. In Figure 3.16 this occurs when either the Participant or a room is not available for the meeting. As both the availability of the Participant and the room are required, if either is unavailable, the meeting cannot proceed. In this situation, there is no dependency between the parallel action sequences, however, booking the meeting is dependent on both the availability of the Participant and the room. In such a case, failure in one parallel action sequence is enough to terminate the parallelism as the outcome of the other parallel action sequence is irrelevant. This is signified by a single directed line entering the parallel bars and a single directed line leaving the parallel bars.

The parallel merge symbol can be interpreted as a logical and, with the incoming parallel

---

\(^8\)This action map does not cater for the situation in which there are no initial date and time available.
Determine Availability

<table>
<thead>
<tr>
<th>Organizer</th>
<th>Participant</th>
<th>Venue Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suggest Date and Time</td>
<td>Timetable</td>
<td>Check Room Availability</td>
</tr>
<tr>
<td>Timetable</td>
<td>Check Participant Availability</td>
<td>Room Available</td>
</tr>
<tr>
<td>Set Provisional Date and Time</td>
<td>Participant Available</td>
<td>Room Available</td>
</tr>
<tr>
<td>Select Next Alternative</td>
<td>Participant Not Available</td>
<td>Room Not Available</td>
</tr>
<tr>
<td>No Date or Time Available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminate Interaction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key**

- Independent Action
- Caused Action
- Final Caused Action
- Action Causality
- Note Indicator
- Data Flow
- Parallel Split
- Parallel Merge
- Data Store

*Figure 3.16: Action Map with Parallel Actions*
sequences as its inputs. That is, when all the specified incoming parallel action sequences have reached the parallel symbol, all parallel action sequences are terminated. The incoming parallel action sequences can be a subset of all parallel action sequence. For example, the two parallel merges before the Select Next Alternative both have only one incoming parallel action sequence although there are two action sequences being executed in parallel. In this case, the termination of one of the parallel sequences will also terminate the other.

Where parallel actions will be used will depend on particular interactions. Generally, there must not be any dependencies between the action sequences being executed in parallel.

Failure handling within parallel action sequences is difficult to specify. The problem is that parallel action sequences must have a clear starting and ending point for synchronization. If there is a rollback action in one parallel action sequence and not in other parallel sequences it is difficult to specify exactly what will happen when the rollback is executed. The action sequence with the rollback will return to a previous interaction goal, however, the other parallel sequences will need to be terminated. This difficulty is compounded with the issue of race conditions in parallelism. As such, although rollbacks within parallel action sequences are possible, it is advised that they are not used within parallel action sequences. On the other hand, action retries can be used in parallel action sequences, as long as they remain within the scope of the parallel action sequences and do not retry actions outside of the parallel action sequences. This ensures that action retries do not affect other parallel action sequences. In general, failure handling within parallel action sequences, especially the rollback mechanism, requires future development and refinement.

3.7.3 Homogeneous Instance Parallelism

In multi-agent systems, it is common to have instances of agents which fill different roles interacting with each other in parallel (which we have termed heterogeneous instance parallelism, refer to Figure 3.13). This is relatively easy to model, however, the case where there are multiple instances of the same agent type interacting with each other (which we have termed homogeneous instance parallelism) is more difficult to model. This is especially so when there is an unknown number of the homogeneous instances and the designer wishes to model the interaction based on the agent types rather than the individual instances. In this section, we present an initial approach using the Hermes notation to model such situations.

Figure 3.17 shows an example of an action map with homogeneous instance parallelism. In this case, the action map is based on an iterative auction protocol which is common to multiple round auction protocols such as the English and Dutch auctions.

Note that the number of agent type instances are shown next to the agent type names at the top of the action map. In this particular case, there is only one Auctioneer and at least two and at most $n$ Bidders. Parallel causality links are denoted by the “crow’s feet” symbol and the multiplicity is shown beneath it.

For example, the Auctioneer’s Request Bids action causes $n$ (i.e. all) Bidders to use their
Valuate Item action. This can be seen as a broadcast from the Auctioneer’s point of view. The opposite of this broadcast is when the Auctioneer is collecting replies from multiple Bidders, such as when it uses its Continue Auction action in response to the different Bidders’ use of their Acknowledge Results Received actions.

At times it is necessary to use parallel actions in conjunction with homogeneous instance parallelism. This is usually the case when a singular agent type instance (e.g. the Auctioneer) must send different messages to the different multiple instances (e.g. the Bidders). For example, at the end of the auction, the Auctioneer must send a successful message to the one winning Bidder and unsuccessful messages to the other Bidders. This can be achieved with a combination of the parallel actions and instance parallelism as shown in Figure 3.17.

Another example is when the Auctioneer must determine the leading Bidder and notify all Bidders. In this case, there are two possibilities. If there is only one leading Bidder, the Auctioneer sends it a message indicating that it is leading and a message to all other Bidders informing them that they are not leading. In the case where there are equal leading Bidders, then no Bidder is considered to be leading and the Auctioneer informs both the equal leading Bidders and the other Bidders of the situation.

After the Auctioneer has notified the Bidders of their situation, the Acknowledge Results Received action is required to synchronize all the instances of the Bidders before the Auctioneer can proceed with its Continue Auction action. The Acknowledge Results Received action is required as, according to our notation, it is not possible to merge multiple causality links using the “crow’s feet” construct.

The designer is free to use any of the Hermes failure handling mechanisms with homogeneous instance parallelism as, although the different instances are executing in parallel, their internal agent processing is independent of each other. It is possible to rollback to previous interaction goals, however, all instances need to be rolled back.

3.8 Design Software Support Tool

For Hermes to be pragmatic, it is important to have software tools to support the process. Software tools are important as they can ease the design process by providing support in various ways. For example, the initial creation of a design can be very unstable as many things change frequently. If there is no software tool support, each time there is a significant change, the relevant design documents will have to be re-drawn by hand. This can be especially cumbersome if it is a large design or if there are many dependencies between design artifacts. Software support can make the creation and maintenance of design documents easier to manage. This allows the designer to focus on actually designing the interaction rather than performing tedious tasks such as re-drawing design documents.

Software support tools can also be used to aid the designer to follow the design process. The support tool can place restrictions on how the designer progresses in the design of interactions.
CHAPTER 3. DESIGNING GOAL-ORIENTED INTERACTIONS

Figure 3.17: Action Map with Homogeneous Instance Parallelism
For example, the support tool for Hermes could prevent designers from creating an action map for a compound interaction goal.

Support tools can also be used to verify design artifacts and constraints. For example, it can prevent designers from making an independent action dependent on another action. Alternatively, the support tool could be given an action message diagram that depicts a desired interaction which it then checks against the relevant action maps to verify whether it can occur or not.

Thus, support tools are useful in that they can be used to support the Hermes design process and to also ensure that designs are valid. To be able to verify a design, a set of rules to check consistency is required. These would include, for example, rules to ensure that it is possible to execute each action (i.e. there is a causality that triggers all actions that are not independent), or that rollback actions will roll back to an existing interaction goal.

As part of the experimental evaluation (refer to Chapter 6), a prototype design tool was created and made available to participants. The tool is based on UMLet\(^9\) and allows users to create interaction goal hierarchies and action maps. The focus of the software tool was the drawing of Hermes design artifacts and, thus, it was not concerned with verifying constraints in the design artifacts created. However, such an addition to the tool will definitely be useful in the future.

Figure 3.18 shows a screenshot of the development of an interaction goal hierarchy in UMLet. The top right-hand panel displays the template from which Hermes artifacts can be created. In this case, as the *Hermes Interaction Goal Hierarchy* template is selected, the template only shows relevant elements for creating an interaction goal hierarchy.

The left-hand panel contains the interaction goal hierarchy being created by the user. As can be seen, all the required elements to create the interaction goal hierarchy are available.

A screenshot of the development of an action map in UMLet is shown in Figure 3.19. As with the screenshot of the interaction goal hierarchy development, the template is in the top right-hand panel. Since this is the *Hermes Action Map* template, all elements required to create an action map are present. The left-hand panel shows an action map in development.

Although the software support tool is primitive, as it further develops, it will better support the design process, which in turn will help designers to more efficiently create Hermes designs.

### 3.9 Summary

The design aspect of the Hermes methodology has been presented in this chapter. This has included both the design notation, which was especially created for Hermes, and the design process that Hermes employs.

The design process is divided into three phases, the *Interaction Goal Hierarchy Design* phase, the *Action Map Design* phase, and the *Message Design* phase, each of which has two steps. The design process follows a mini-waterfall model in which each step is derived from its predecessor and also provides feedback to its predecessor.

\(^9\)http://www.umlet.com/
Figure 3.18: UMLet: Interaction Goal Hierarchy Development
Figure 3.19: UMLet: Action Map Development
The design process was illustrated by use of an e-commerce example in which a Customer role attempts to purchase a product online from a Merchant role. Techniques which designers can use to help them design interactions were explained along with descriptions of Hermes artifacts such as the interaction goal hierarchy, and the action maps, which are used to model Hermes interactions. Furthermore, we described action sequence diagrams and action message diagrams, artifacts which can be used to ensure that designs are appropriate and that they will achieve the desired interactions. Also provided was an explanation of how messages are identified and used within Hermes.

Three types of parallelism, interaction goal parallelism, action parallelism, and instance parallelism were also identified and discussed.

A primitive software design tool for designing Hermes interactions was presented along with a discussion on the vision and benefits that such tools contribute in order to make a methodology practical to use.
Chapter 4

Integrating Hermes into Prometheus †

As Hermes focuses on the design of agent interactions, it does not cater for the design of other aspects of agent systems. However, for Hermes to be practical and easily accepted, there must be some way of addressing other aspects of system design. Given the number of existing agent system design methodologies, such as Gaia, MaSE, Tropos, and Prometheus, and the fact that the AOSE community has not converged towards one of these methodologies, the integration of Hermes with these existing methodologies, rather than extending Hermes to encompass full agent system design, is a more desirable option. Integration is more advantageous as extending Hermes to a full agent system design methodology would create yet another agent design methodology; an unnecessary addition at this point. Furthermore, integration is preferable as it entails that agent designers who are knowledgeable with an existing methodology will only have to learn the Hermes methodology as opposed to learning a completely new methodology to gain the benefit of goal-oriented interaction design.

Integrating two methodologies can be seen as a form of method engineering [Brinkkemper et al., 1998; Henderson-Sellers, 2003]. Method engineering deals with defining a meta-model which is used to integrate parts of existing methods or methodologies, known as method fragments, into a project-specific methodology. The method fragments are integrated into a methodology by use of defined assembly rules and mechanisms [Brinkkemper et al., 1998]. Our work on integrating Hermes into another existing methodology, Prometheus, has commonalities with method engineering, such as artifacts from Hermes feeding into Prometheus processes and vice versa. However, unlike method engineering, our integration work focuses on the integration details between these two methodologies without the overhead of providing a general formal meta-model for the integration. The integration is carried out informally as the two methodologies are closely related. Thus, in

†Part of the work presented in this chapter has been previously published [Cheong and Winikoff, 2006c].
this chapter we present a case study explaining how Hermes is integrated with Prometheus, a full agent system design methodology.

Prometheus was chosen primarily due to local expertise and serves only as an example of the integration. From the integration of Hermes and Prometheus, general integration guidelines can be drawn to explain how Hermes can be integrated with other methodologies.

In general, integrating Hermes with a full agent design methodology will require analyzing the methodology and identifying which parts of it constitute the interaction design process. Those parts will then need to be replaced by Hermes. This will also involve determining what dependencies are required by the methodology’s current interaction design process and what other processes depend on the design artifacts produced by the interaction design process. Once that has been determined, the design artifacts which the methodology’s current interaction design process depends on can be used to better guide the Hermes design process. Hermes’ final design artifacts also will need to be integrated into the methodology.

To explain the integration process, an electronic book store example is used. The following section describes the book store example, after which the succeeding sections explain the integration process and how to use the amalgamated methodology to create an interaction design.

4.1 Book Store Case Study

The example for this chapter is taken from [Padgham and Winikoff, 2004]. In particular, an interaction is developed around ordering a book, based on the Order book scenario [Padgham and Winikoff, 2004] which is reproduced (in abridged form) in Figure 4.1.

In addition to the scenario, the defined agent types are required, including the roles and data contained within. The book store example defines the following agent types:

- **Sales Assistant**: comprising the roles of Book finding, Welcoming, Purchasing, and Online interaction.
- **Customer Relations**: comprising the roles Profile monitor and Customer contact, and the Customer DB database.
- **Delivery Manager**: comprising the roles Delivery handling and Lost goods management, and the databases Customer orders and Delivery problems. This agent type also has access to external databases of couriers and postal areas.
- **Stock Manager**: comprising the roles of Stock management, Competition management, Price setting, and Catalogue management, and the databases Pending orders, Books DB, Stock orders and Stock DB.
**Name:** Order Book  
**Description:** An order is received from the WWW page interface (goal Place Order). Information is obtained in order to place the order and the order is placed.  
**Trigger:** Goal: Place Order  
**Steps:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Type</th>
<th>Name</th>
<th>Role</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>Obtain delivery options</td>
<td>Delivery handling</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>Goal</td>
<td>Calculate delivery time estimates</td>
<td>Delivery handling</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Present information</td>
<td>Online interaction</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>Percept</td>
<td>User input</td>
<td>Online interaction</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>Goal</td>
<td>Obtain credit card details</td>
<td>Purchasing</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>Percept</td>
<td>User input</td>
<td>Online interaction</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>Action</td>
<td>Bank Transaction</td>
<td>Purchasing</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>Percept</td>
<td>Bank transaction response</td>
<td>Purchasing</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>Goal</td>
<td>Arrange Delivery</td>
<td>Delivery handling</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>Action</td>
<td>Place delivery request</td>
<td>Delivery handling</td>
<td>uses: customer order record</td>
</tr>
<tr>
<td>11</td>
<td>Goal</td>
<td>Log outgoing delivery</td>
<td>Delivery handling</td>
<td>produces: Customer Orders</td>
</tr>
<tr>
<td>12</td>
<td>Goal</td>
<td>Log books outgoing</td>
<td>Stock management</td>
<td>uses: Customer order record produces: Stock DB</td>
</tr>
<tr>
<td>13</td>
<td>Goal</td>
<td>Update customer record</td>
<td>Profile monitor</td>
<td>produces: Customer DB</td>
</tr>
<tr>
<td>14</td>
<td>Action</td>
<td>Send email</td>
<td>Customer contact</td>
<td>uses: Customer DB</td>
</tr>
</tbody>
</table>

**Variation:** Book is not currently available. Include information with delivery options. Replace steps 7–12 with steps to add the order to an orders pending file.

*Figure 4.1: Order Book Scenario (From [Padgham and Winikoff, 2004])*
CHAPTER 4. INTEGRATING HERMES INTO PROMETHEUS

4.2 Hermes and Prometheus Integration Overview

Prometheus is an agent system design methodology which supports design activities from developing initial system requirements through to detailed agent internals which are ready to be implemented on agent platforms, such as JACK and Jadex. Prometheus also has some debugging support [Padgham et al., 2005b] and a software tool to support its design processes [Padgham et al., 2005a; Thangarajah et al., 2005] which is also able to produce skeleton code. A variation of the tool [Jayatilleke, 2007] produces fully executable code. Prometheus employs a message-centric approach to interaction design and uses AUML to model the interactions. Refer to Section 2.4.1 for more details on Prometheus.

As previously mentioned, to integrate Hermes with a full agent system design methodology, the methodology’s interaction design process, design artifacts and their dependencies must be identified. In the case of Prometheus, the interaction design process encompasses the creation of interaction diagrams, interaction protocols and process diagrams (refer to the shaded boxes in Figure 4.2). The interaction diagrams are derived from scenarios, which are part of the System Specification phase and interaction protocols are in turn derived from interaction diagrams. Process diagrams are part of the Detailed Design phase and are derived from interaction protocols. Although Prometheus’ interaction design process primarily draws on the scenarios, other information, such as agent roles and data stores are also required. Artifacts which depend on the interaction design process are the plan descriptors and messages.

The main difference between Hermes and the Prometheus interaction design process is that the former is goal-oriented whilst the latter is message-centric. Although the Prometheus design process is message-centric, it does take into consideration system goals (as shown in Figures 4.1 and 4.2) as its scenario design artifacts are closely linked to the system goals. However, as the design process progresses into the Architectural Design phase and the interaction diagrams and protocols are developed (refer to Figure 4.2), the system goals are no longer explicitly considered. In contrast, Hermes considers the system goals more explicitly in the Architectural Design phase when it is integrated with Prometheus as they are used to derive some of the interaction goals. This is emphasized in Figure 4.3 by the direct connection between the system goals and the interaction goal hierarchy. Figure 4.3 shows an overview of the amalgamated methodology in which Hermes artifacts are represented by shaded boxes.

The Hermes interaction goal hierarchy and action maps replace the following Prometheus interaction artifacts: interaction diagrams, interaction protocols and process diagrams. Note that action maps are a mixture of interaction protocols and process diagrams in that action maps contain both inter- and intra-agent details whilst interaction protocols purely contain inter-agent communications and process diagrams contain solely internal agent processes.

As the Prometheus scenarios capture interactions between agents, they are a good source from which to derive the interaction goal hierarchy. Although system goals are incorporated in scenarios, there is also a direct link between the system goals and the interaction goal hierarchy.
This serves as a reminder to the designer that the interactions to be designed are goal-oriented and allows the designer to draw interaction goals from the system goals. As per the usual Hermes process, the action maps are derived from the interaction goal hierarchy. Additionally, information for datastores can be obtained from Prometheus data coupling diagrams.

At the end of the Hermes design process, messages (Prometheus artifacts) can be drawn from Hermes message descriptors.

As actions from the action maps are implemented as plans (refer to Chapter 5) there could be a link between the action maps and plan descriptors. However, this link is not shown as plan descriptors are not derived directly from actions.

The integration of Hermes messages into Prometheus provides the link which transitions from the Hermes design process back to the Prometheus methodology. The details of this integration and of the integrated process design is described in Section 4.3.
4.3 Integrated Process

The integrated methodology follows the process outlined in Figure 4.4. The first five steps follow the typical Prometheus procedure as they are not involved with interaction design. However, prior to step 6, which is the start of the interaction design, the designer must determine whether it is worthwhile to use Hermes or to carry on with the usual Prometheus message-centric interaction design process.

In step 1, System Description, a textual description of the system to be designed is developed. This is usually developed by discussing the purposes of the system with the users involved. Once the system’s purpose has been established, the designer analyzes the System Description and begins to create the analysis overview diagram (refer to Figure 4.3). This is part of step 2 in which actors, percepts, actions, and scenarios are identified. The actors are then associated to scenarios and system goals are identified and developed.

In step 3, the roles are derived from the system goals by grouping the goals into logical and coherent groups. Thus, the roles are simply groupings of similar or related system goals. The roles are recorded in the intermediate Initial Role Descriptor artifacts. As these are intermediate design artifacts, they are not retained in the final design documentation, however, further on in the design process, these descriptors are used to develop the final Agent Descriptor artifacts, which
1. System Description

2. Develop Analysis Overview Diagram

3. Identify Roles by grouping goals

4. Develop Scenarios

5. Determine agent types by grouping roles

6. Identify Hermes Roles and Interaction Goals

7. Develop Interaction Goal Hierarchy (Section 4.4)

8. Develop Action Maps (Section 4.5)

9. Develop Action Sequence Diagrams and Action Message Diagrams (Optional)

10. Identify Messages

11. Develop System Overview Diagram (Prometheus)

12. Proceed with remainder of Prometheus process

Before step 6 an assessment is made of whether it is worthwhile to use Hermes. If not, then replace steps 6-10 (shown in italics) with the existing Prometheus methodology.

Figure 4.4: Integrated Design Steps

are retained.

Step 4 calls for the development of the scenarios. The scenarios are developed in consideration of both the system goals and the roles, as the scenarios capture sequential interactions between roles and suggest how agents can achieve the system goals. Scenarios are recorded in the final Scenario artifacts.

In step 5, the agent types are then determined by grouping together related roles. This leads to the creation of the final Agent Descriptor artifacts, in which they are recorded.

The interaction design is composed of steps 6–10. At this point, the designer must assess if it is worthwhile to use the Hermes approach to create the interactions, as using Hermes has both advantages and disadvantages (refer to Chapter 6).

While Hermes produces more flexible and robust interactions, it is also more time consuming and is thus unnecessarily complex for simple interactions. Thus, this decision point allows the designer to use Hermes if the interaction is likely to be complex and/or failure prone or the typical Prometheus interaction design process if the interaction is to be simple, such as a query and
response interaction. If Hermes is to be used, steps 6–10 are used, otherwise, the existing steps from Prometheus are used.

The remainder of this chapter will cover in detail the integrated Hermes and Prometheus approach to interaction design. The reader is referred to Section 2.4.1 and to Padgham and Winikoff [2004] for more details on the Prometheus design process.

The identification of Hermes roles (step 6) is straightforward as they are usually taken from Prometheus agent types. For example, in the e-commerce interaction, merchant and customer are two obvious agent types.

In Prometheus, agent types are composed of Prometheus roles, thus when a Hermes role is based on an agent type, the Hermes role will take on all the roles that the agent type is composed of. However, it is also possible to base Hermes roles on Prometheus roles. For example, consider an agent type named Trader which is composed of two Prometheus roles, customer and merchant. In certain situations, the Trader acts as a customer to purchase stock from other agents, whereas in other situations, it acts as a merchant and sells stock to other agents. Depending on the interaction being modelled, the relevant Hermes roles can be based on either of the two Prometheus roles (i.e. customer or merchant) or the agent type (i.e. Trader).

In the example used in this chapter, we have based the Hermes roles on agent types rather than Prometheus roles.

The process for developing the interaction goal hierarchy and its relevant action maps (steps 7 and 8) are slightly different than the original version of Hermes. On the whole, their creation is more straightforward and more guided as the designer is developing these artifacts based on Prometheus’ scenarios. Sections 4.4 and 4.5 describe, respectively, the refined process in which the interaction goal hierarchy and action maps are designed.

Action sequence diagrams and action message diagrams remain unchanged from the original Hermes approach. Their intent is still the same, i.e. to check that the created designs can produce the desired interactions.

Messages that are produced from Hermes are to be re-integrated (step 10) into Prometheus. Hermes interactions are depicted in Prometheus’ system overview diagram in the same manner that protocols are shown in the original Prometheus: as a protocol symbol between the agents involved in the interaction. For example, Figure 4.5 indicates that the Customer and Merchant agent types are involved in the Trade protocol. The protocol symbol, on which its name is written (in this case Trade), abstracts all the messages involved in the protocol. The bidirectional arrows between the agents and the Trade interaction indicates that the agent types both send and receive messages.

At the system overview level, protocol types are indistinguishable. That is, the system overview diagram simply depicts that there is an interaction between agents; it does not specify whether the interaction is message-centric or goal-oriented, although, clearly, the notation could be slightly extended to show this. To determine what type of interaction the agents are involved in, the
actual Hermes or Prometheus interaction design artifacts are referred to.

The messages identified from the Hermes design can be carried across and adapted to Prometheus message descriptors as the message descriptors from both methodologies are very similar. Interaction goals (from the interaction goal hierarchy) and actions (from the action maps) are implemented as different plan types (refer to Chapter 5), thus, they are added to the agent overview diagrams.

### 4.4 Interaction Goal Hierarchy

The interaction goal hierarchy design process in the integrated methodology is slightly different to the original Hermes approach, with the main difference stemming from the fact that the interaction in the integrated methodology is derived from a defined scenario and Prometheus’ goal overview diagram, which is a structured overview of all system goals. As most of the interaction goals will be identified from the Prometheus scenario and system goals, this means that the creation of the Hermes interaction goal hierarchy will be more guided.

As with the version of Hermes described in Chapter 3, the first step in creating the interaction goal hierarchy for the interaction being designed is to determine the overall intent of the interaction. In this case, as the designer is working from a defined Prometheus scenario, this should be obvious. Usually this is captured by the name of the scenario which has a link to a system goal of (usually) the same name. For example, the overall intent of the book store example is `order book`. Thus, it is placed at the top of the hierarchy to represent the purpose of the interaction.

The next step is to identify the remaining interaction goals and place them in the hierarchy. Once again, these are generally drawn from the Prometheus scenario. Eliciting interaction goals from the scenario and system goals can be done by:

- mapping Prometheus system goals into interaction goals.
- decomposing scenario steps into interaction goals; or
- abstracting or grouping the scenario steps into cohesive interaction goals;

Prometheus system goals can be mapped directly to either interaction goals or actions. Whether a system goal maps to an interaction goal or an action will depend on the specific system goal and its properties. Usually high-level system goals (i.e. those that are abstract or easily decomposable
CHAPTER 4. INTEGRATING HERMES INTO PROMETHEUS

into sub-goals) or those that involve more than one agent to achieve it map to interaction goals. For example, consider a system goal entitled Purchase. This goal will likely be mapped to an interaction goal as it is a goal which requires at least two agents, i.e. a Customer and a Merchant, to achieve it.

In contrast, low-level goals (i.e. those that are concrete or not easily decomposable into sub-goals) or system goals that involve only a single agent are usually mapped to Hermes actions. For example a Send Confirmation Email system goal would most likely be mapped to a Hermes action as it is low-level and requires only one agent to achieve it.

There is an intentional overlap in the heuristic described above, that is, system goals can sometimes be mapped to either an interaction goal or a Hermes action. In such cases, it is up to the designer to decide whether the system goal should map to an interaction goal or an action.

It is also possible to identify interaction goals by decomposing scenario steps. This is required when scenario steps are too coarse-grained to be mapped directly to an interaction goal, however, this is uncommon as scenario steps are usually fine-grained.

A more common way to derive interaction goals from scenarios is to abstract or group scenario steps together rather than decomposing the scenario steps\(^1\). For example, consider the book store interaction described in Section 4.1. Upon analysis of the scenario, it appears that steps 1–4 are related. These steps gather, present and calculate information about delivery details. Thus, they can be grouped together into one high-level interaction goal, Retrieve Delivery Choice.

Similarly, steps 5 and 6 are also related. They gather and present information about credit cards. As such, they can be grouped together in a Retrieve Credit Card Details interaction goal.

Since these two interaction goals, Retrieve Delivery Choice and Retrieve Credit Card Details, are very similar, they can be further abstracted into a single interaction goal, Retrieve Details. Therefore, the Retrieve Details interaction goal is composed of two sub-goals, Retrieve Delivery Choice and Retrieve Credit Card Details.

The scenario shows that the Retrieve Delivery Details interaction goal (steps 1–4) is to be completed before the Retrieve Credit Card Details (steps 5 and 6). As this is logical (payment details are usually gathered at the end, just before the delivery of the product), this temporal restriction is retained. Thus, a temporal dependency link is placed between the Retrieve Delivery Choice and Retrieve Credit Card Details interaction goals. This temporal dependency link states that Retrieve Credit Card Details cannot be attempted until the Retrieve Delivery Choice interaction goal is successfully achieved.

The Retrieve Details interaction goal and its sub-goals, Retrieve Delivery Details and Retrieve Credit Card Details are added to the interaction goal hierarchy as a sub-goal to Order Book, the top-most interaction goal (refer to Figure 4.6).

As scenario steps 7 and 8 are related to performing transactions with the bank, they can be

\(^1\)Scenario steps which are of type “sub-scenario” may need to be treated differently. Refer to Section 4.5.1 for more details.
abstracted into a Process Bank Transaction interaction goal. Steps 9–14 are grouped into an Organize Delivery interaction goal as they deal with organizing delivery of the book purchased to the Customer.

Both Process Bank Transaction and Organize Delivery can be grouped under a Process Order interaction goal. As with Retrieve Delivery Details and Retrieve Credit Card Details, the scenario shows that Process Bank Transaction (steps 7 and 8) occurs before Organize Delivery (steps 9–14). Once more, as this is a sensible restriction, it is retained in the interaction goal hierarchy by placing a temporal dependency link between Process Bank Transaction and Organize Delivery. The mapping of scenario steps to interaction goals is summarized in Table 4.1.

The Process Order interaction goal is then added to the interaction goal hierarchy as a sub-goal to the Order Book interaction goal. Since Retrieve Details occurs before Process Order, as implied in the scenario steps, a temporal dependency link is placed between the two.

The resulting interaction goal hierarchy (refer to Figure 4.6) is strongly temporally constrained. In order to successfully achieve the Order Book interaction, the following goals must be achieved in sequence: (1) Retrieve Delivery Choice, (2) Retrieve Credit Card Details, (3) Process Bank
In fact, as Prometheus scenarios are specified as a sequence of steps, any interaction goal hierarchy or action map which closely follows the scenario will be strongly temporally constrained. Therefore, an important part of creating interactions from Prometheus scenarios, whether using Hermes or the original Prometheus process, is to determine where the ordering represents actual temporal constraints.

At the end of the interaction goal hierarchy design process, the produced interaction goal hierarchy is the final artifact. Although it is complete, in that it represents what is specified in the scenario, sometimes changes will need to be made. The need for these changes, if any, usually becomes apparent in the action map design process, which is described next.

4.5 Action Maps

The explanation of the integrated methodology's action map development process is described in four distinct steps to be directly comparable with the explanation in Chapter 3. However, as stated in Chapter 3, it is not intended for the designer to strictly follow these four steps.

The actual four steps themselves remain the same, however, the sources where each step draws its information from are different from the original Hermes process. As with the interaction goal hierarchy, the main difference is that the Hermes process is better guided as most of the information can be identified from the Prometheus scenario.

4.5.1 Develop Initial Action Maps

The initial step in creating action maps with the integrated methodology is much simpler and more straightforward than the original Hermes. In this step, the interaction designer simply transcribes relevant steps from the scenario onto the action maps.

There are five types of steps in Prometheus scenarios: goal, action, percept, other and scenario. The scenario steps that are goals correspond to goals of an individual agent and, thus, are mapped to actions in the action maps. Scenario steps that are actions map (obviously) to actions. For example, when creating an action map, step 10 of the scenario, an action titled, Place Delivery Request, will be mapped to a Hermes action of the same name.

In Prometheus, percepts are information that agents can determine or receive from the environment. When translated to Hermes, this means that percepts are actions that somehow gain the required information in the interaction. Typically, this means that percepts are mapped to actions that either wait for incoming information or actively retrieve the information.

For example, step 8, Bank Transaction Response, will be modelled as a caused action that will be triggered once the bank transaction response is received. In contrast, step 6, User Input, will be modelled as an action that actively sends a request to obtain the user's input.
The scenario steps that are marked as other are also mapped as actions. As they are a more generic type of step, the designer’s judgement will be required.

Scenario steps marked as scenario indicate a sub-scenario, i.e. a reference to another scenario which is to be completed as part of the current scenario. The use of scenario steps are usually to decompose large scenarios into smaller, more manageable scenarios and/or to promote the re-use of common scenarios.

How the sub-scenario step is designed in Hermes will depend on the size and complexity of the sub-scenario. If it is a small and simple sub-scenario, it can be added to the interaction goal hierarchy as a single interaction goal with a relevant action map which will achieve it. On the other hand, if the sub-scenario is rather large and complex, it can be added to the interaction goal hierarchy as a branch.

That is, a small interaction goal hierarchy is created from the sub-scenario steps, with the interaction goal at its apex representing its overall intent and sub-goals placed under it. The small interaction goal hierarchy is then added to the main interaction goal hierarchy and action maps are developed as usual.

Designing the initial action map for the Organize Delivery interaction goal (scenario steps 9–14) requires placing five steps (10–14, step 9 is omitted as it is achieved by steps 10–14) onto the action map. Although it is best to follow the scenario steps as closely as possible, it is likely that some slight deviations from the scenario will be necessary. Deviations include changing the ordering of some of the actions or creating new actions to clarify certain parts of the action maps.

As with the original Hermes, the action maps are divided into “swim lanes” and the roles are placed at the top of the lanes. In the original Hermes process, the designer needs to analyze the interaction, determine what roles and actions are required, and place those actions in the appropriate swim lane. However, in the integrated design process, creation of the action maps is more straightforward and better guided.

Determining the correct swim lanes is a matter of assigning steps to the Hermes role that has been formed from the corresponding Prometheus role (which may be through an agent type). For example, scenario step 10, Place Delivery Request is associated with the Delivery Handling role, which is part of the Delivery Manager agent type. Thus, this action is placed in the Delivery Manager (Hermes role) swim lane.

As per the original Hermes, action types will need to be selected for each action, such as independent, caused, final caused, and final independent (refer to Section 3.3.1 on page 70).

Once the actions have been placed in their relevant swim lanes, causality links are added between the actions to identify the flow of actions. The placement of the causality links are based on, but not restricted to, the sequence exhibited in the scenario. That is, they are generalized from the scenario sequence.

The result of this step, transcription of the roles and actions onto an action map, and the addition of causality links, is shown in Figure 4.7.
CHAPTER 4. INTEGRATING HERMES INTO PROMETHEUS

Organize Delivery (Initial without data stores)

<table>
<thead>
<tr>
<th>Sales Assistant</th>
<th>Customer Relations</th>
<th>Delivery Manager</th>
<th>Stock Manager</th>
</tr>
</thead>
</table>

**Key**
- Independent Action
- Caused Action
- Final Caused Action
- Action Causality

- Update Customer Records
- Send Email

**Figure 4.7: Initial Action Map without Data Stores: Organize Delivery**

Note that Figure 4.7 presents one possible design, however, there are a number of alternatives possible, including some in which actions are carried out in parallel.

### 4.5.2 Adding Data to Action Maps

As with the previous step, this step is simpler and more straightforward in the integrated methodology due to working from a defined scenario. The data which the agents use is already identified in the Prometheus scenario. Thus, this step involves transcribing information from the scenario onto the action map.

Identifying relevant data stores is simply a matter of analyzing the scenario to determine which data stores need to be added to the action maps. The data column specifies which data stores agents will need to have read and write access to. For example, in step 12, Log Books Outgoing, according to the data column, the Stock Manager will need to use (i.e. read) the Customer Order Record and produce (i.e. write) data to the Stock DB. Thus, the designer will need to ensure that the Stock Manager will have access (either directly or through a number of actions) to these data stores.

The remainder of this step is as per the original Hermes process (refer to Section 3.3.2).

In the case of the action map for the Organize Delivery interaction goal, the Prometheus scenario steps 10–14, specify what data stores are read and written to. The data stores that are
in use are: Customer Orders, Stock DB and Customer DB. According to the scenario description, the Customer Relations role contains the Customer DB data store, the Delivery Manager contains the Customer Orders data store and the Stock DB data store belongs to the Stock Manager agent. As such, these data stores are added to the appropriate swim lanes in Figure 4.8.

![Diagram](image-url)

*Figure 4.8: Initial Action Map with Data Stores: Organize Delivery*

Furthermore, the correct data flow (i.e. access to data, and read/write access) has been ensured and is shown in Figure 4.8 by use of the data flow indicators.

### 4.5.3 Generalizing Action Maps

The purpose of this step is to add flexibility to the interaction by providing alternative success paths in the event of failures. This is done as per the original Hermes process (refer to Section 3.3.3 on page 76). That is, each action is systematically analyzed by the designer to determine if it can be achieved differently or if additional actions can be added.

In the integrated Hermes and Prometheus methodology, additional information can be obtained from the Prometheus scenario variations. The scenario variations can be used as a source of possible failures and associated remedial steps, which can be incorporated into the Hermes design. However, if the scenario variations are not well constructed or if scenario variations are lacking, the designer can use the original Hermes process to generalize the action maps.

In the case of the Order Book scenario, there are no success paths other than the one stated
in the scenario variation. The remainder of this section explains how the scenario variation is incorporated into the action map developed so far.

As the Prometheus scenario variations vary greatly depending on the domain, the agents involved and the actual interaction, there are no set guidelines for adding scenario variations to action maps. Incorporating scenario variations can be as simple as adding an action, or as complicated as requiring multiple actions and making changes to the interaction goal hierarchy.

In the case of the Order Book scenario (refer to Section 4.1), the variation states that if the book ordered is not available, steps 7–12 should be replaced with steps to add a pending order. This can be incorporated into the action map by providing two ways in which the Organize Delivery interaction goal can be achieved: (1) when the ordered book is available, the delivery order placed and processed (as depicted in Figure 4.8), and (2) when the ordered book is unavailable, a pending order is created. Once the book is available, the pending order is filled and the delivery is processed.

Note that in the current action map (refer to Figure 4.8), the availability of the ordered book is never explicitly queried; it is assumed to be part of the delivery options. In order to clarify matters, querying for the ordered book’s availability needs to be made explicit. This is done by adding two new actions at the start of the action map: Check Book Availability and Check Stock (refer to Figure 4.9). These two actions are used to determine how to arrange the delivery. Check Book Availability is used to query the Stock Manager about the availability of the ordered book. Check Stock is the action in the Stock Manager that replies to the query. If the ordered book is available, the delivery order is placed and processed. If the ordered book is not available, the Add Pending Order action is used to order the book (from the publishing firm). Once the book comes in (from the publishing firm), Process Newly Received Stock is triggered, the pending order is filled and the delivery is processed. The result of this step is shown in Figure 4.9.

### 4.5.4 Adding Failure Handling to Action Maps

This step is unchanged from the original Hermes. At this point, the design interaction should be flexible due to incorporating the scenario variations. However, this step attempts to further increase flexibility and robustness by identifying possible failures and using Hermes’ failure handling mechanisms to address them.

In the first part of this step, the action map is analyzed to determine where possible failures can occur. For each action that can fail, the possible failures and remedial action that can be taken to recover from the failure are summarized in Table 4.2.

For example (refer to Table 4.2), the Order Book action can fail if the book is out of print. In this case, one way of dealing with the failure is to suggest alternative titles or editions to the shopper.

The remedial steps are added to the action map by use of action retries and rollbacks (shaded action boxes in Figure 4.10 with numbers corresponding to those in Table 4.2). For example, it has been identified that the Order Book action could fail if a book is out of print, and the suggested
CHAPTER 4. INTEGRATING HERMES INTO PROMETHEUS

Figure 4.9: Action Map: Organize Delivery (with scenario variation)

<table>
<thead>
<tr>
<th>#</th>
<th>Action</th>
<th>Possible Failures</th>
<th>Remedial Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Order Book</td>
<td>Book out of print</td>
<td>Suggest alternative title or edition</td>
</tr>
<tr>
<td>2</td>
<td>Place Delivery Request</td>
<td>Invalid address</td>
<td>Get details from user and validate</td>
</tr>
<tr>
<td>3</td>
<td>Send Email</td>
<td>Email bounces</td>
<td>Use different medium to contact user (e.g. send mail via post)</td>
</tr>
</tbody>
</table>

Table 4.2: Possible Failures and Remedial Actions for Organize Delivery
remedial action is to suggest an alternative title. In Figure 4.10, this is achieved by the Process Book Out Of Print Message action, which leads to the Suggest Alternative or Similar Title action.

As stated in Section 3.3.4, adding action retries can lead to loops between actions, and it is important to ensure that there are no unintended endless loops. In Figure 4.10, there is a loop between Get User Address (an action retry) and Place Delivery Request. The intention is that the user is re-prompted for an address every time an invalid address is encountered. However, an additional action, Terminate Interaction, has been provided in the case the user cannot or does not wish to enter an address and would like to exit the interaction at that point.

In the last part of this step, rollbacks are added to the action map. In this particular case, all three failures are dealt with using action retries. However, the first failure (refer to Table 4.2) involves a possible rollback if the user decides to purchase the suggested alternative and the new book is more expensive than the previous book. The Rollback to Process Bank Transaction action has been added to the action map in Figure 4.10 to reflect this.

The result of this step is shown in Figure 4.10. The action map appears to be quite complex, however, it provides flexible and robust interactions, including dealing with the failure cases summarized in Table 4.2.

This step, adding failure handling to action maps, is intended to be iterative. That is, once failures have been identified and handled appropriately, the designer should apply this step again. This may lead to identification of failures which the failure handling has introduced. For example, in Figure 4.10, the Send By Post action was added to address failure to send an email. However, Send By Post could itself fail. This failure would typically be identified in the additional iteration, however, in this case, as it is unlikely enough that Send By Post will fail, it is not catered for.

4.6 Software Tool Support

Hermes has existing software tool support (refer to Section 3.8), as does Prometheus [Thangarajah et al., 2005; Padgham et al., 2005a]. For the integrated methodology to be pragmatic, the two software support tools must also be integrated. There are four ways in which the support tools can be integrated together:

1. integrate at the data level by transforming between data formats and use the support tools separately;
2. launch the Hermes tool from the Prometheus tool;
3. implement support for Hermes within the Prometheus tool; or
4. integrate Prometheus and Hermes tools together as plugins into one integrated development environment (IDE).

The first approach is the simplest one in terms of implementation. The two support tools are used independently from each other. Thus, when an agent system needs to be designed,
CHAPTER 4. INTEGRATING HERMES INTO PROMETHEUS

Figure 4.10: Final Action Map: Organize Delivery
the designer uses the Prometheus Design Tool (PDT). If the designer decides that a particular interaction should be goal-oriented, the designer then switches to the Hermes design tool.

The connection between the two support tools is done at the data level. That is, there will need to be a way in which data from one tool – most likely the Hermes tool – is transformed into data that is understood by the other tool – the Prometheus Design Tool. How well this approach works will depend on the representation of and the actual data which the support tools use. The integration at the data level could be as simple as creating a script to transform a particular data representation to another or to extract required data and present it in a particular representation which the other tool can use. In more complicated cases, the tools may need to be modified to include additional data which is required by the other tool.

Although this approach can be easily achieved by simple scripts (or XSL if XML is used to represent data in both tools), it has a number of disadvantages. It is the least pragmatic approach for the design of the integrated methodology as the designer will need to be knowledgeable with both tools and will need to keep on switching between the tools. As well as switching between the two support tools, the designer will also need to transform data from one representation to another. Although this can be automated by the use of scripts, it can take a significant amount of time, especially for large designs or designs in which the designer uses multiple iterations. Thus, although this integration approach is the easiest to implement, it is not very pragmatic for the designer.

The second approach is an improvement upon the first but requires more implementation work. In this approach, the designer still uses the two different support tools, however, the Prometheus Design Tool is able to launch the Hermes tool when required. Behind the scenes, the data transformation between the two tools are done automatically.

Although the support tools are more closely integrated, the designer will still need to be knowledgeable with both tools and will still need to switch between them.

The third approach is perhaps the most implementation intensive. In this approach, Hermes support is implemented into the Prometheus Design Tool, and the existing Hermes tool is not used. The implementor will need to know Hermes, Prometheus, and the integrated methodology described in this chapter in detail to be able to add Hermes support to the Prometheus Design Tool.

This approach is the best for the designer, who will be presented with one integrated development environment with a uniform graphical user interface. The designer will not need to worry about switching to different support tools or running scripts to perform data transformations between tools.

In regard to implementation, this is not the best approach. The actual implementation will require a lot of work. The maintenance of the software tool will also be difficult. If improvements are made to the existing Hermes tool, the integrated software tool will not benefit from it. Instead, the improvements will need to be re-implemented in the integrated software.
The fourth approach, which is a variation of the second approach, is a pragmatic compromise between being pragmatic for the designer and the amount of implementation work required. In this design, both support tools are implemented as plugins and are integrated together into an existing integrated development environment, such as Eclipse\(^2\).

The amount of work required implementation-wise will depend on the current state of the two support tools. If they are not implemented as plugins, they will need to be altered. However, the Hermes design tool is simply UMLet with additional palettes for Hermes, and UMLet can be run independently or as an Eclipse plugin. Thus, the Hermes design tool is already an Eclipse plugin. The Prometheus Design Tool is also available as an Eclipse plugin [Padgham et al., 2005a]. Thus, this approach does appear to be feasible.

From a software engineering standpoint, this is a good approach. Each plugin is modular and encapsulates the internal details of each methodology. The implementor will need to integrate both plugins with each other and into Eclipse. This will include integrating the data which the plugins use. Additionally, improvements to the individual plugins will cascade to the integrated development environment.

This approach is also good for the designer, who will be presented with one uniform integrated development environment. The designer will not need to worry about switching to different support tools or transforming data from one tool to another. Instead, the designer can concentrate solely on designing the interaction. However, one issue is that, although the designer has a uniform integrated development environment in Eclipse, the graphical user interfaces of the two plugins will be different. Furthermore, each of the plugins has its own data, and these need to be synchronized.

### 4.7 Summary

This chapter presented, in detail, how Hermes can be integrated with a full agent system design methodology, using Prometheus as an example. It explained and showed how the design aspect of the two methodologies can be merged to create an integrated methodology that is able to design full agent systems with goal-oriented interactions.

An overview of the new design process, which incorporated Prometheus and Hermes process steps, was presented. Existing Prometheus design artifacts can be used to enhance and better guide the Hermes designs. This was explained in detail for the development of Hermes’ interaction goal hierarchy and action maps.

Furthermore, a discussion on how existing Prometheus and Hermes software support tools could be integrated was also presented.

\(^2\)http://www.eclipse.org/
Chapter 5

Implementing Goal-Oriented Interactions †

This chapter describes the implementation aspects of Hermes, including guidelines on how to implement the Hermes design artifacts, and software support tools that are available to aid the interaction implementor.

The guidelines explain how a Hermes design is systematically mapped to an implementation. As Hermes is goal-oriented, it is natural to use an implementation platform that defines agents in terms of goals and plans. Apart from that restriction, Hermes designs are able to be implemented on any goal-plan agent platform, including those based on the Belief-Desire-Intention (BDI) model, such as JACK, Jadex, JAM, Jason, and others.

Although Hermes designs have only been currently implemented using Jadex, it is possible to implement Hermes on any of the aforementioned platforms. This is possible as the implementation scheme does not use any platform-specific features. Hermes artifacts, such as interaction goal hierarchies and action maps, are mapped to collections of goals and plans; generic features that are common to all goal-plan agent platforms.

An overview of the implementation is shown in Figure 5.1, including the different plan types and their inter-connections. Agent interactions designed using Hermes are not centralized. As shown in Figure 5.1, agents (in this case Agent A and Agent B) have their own local copies of plans which are derived from the action maps created in the Hermes design process. Agents do not need to know about the entire interaction. Instead, local beliefs about their part in the interaction are sufficient and they coordinate through the interaction by sending each other messages to trigger appropriate plan execution (refer to Section 5.5 and Figure 5.4).

Although there are three different types of plans in a Hermes implementation, namely coordination, achievement, and interface plans, these plans are implemented as “normal” plans and

†Part of the work presented in this chapter has been previously published [Cheong and Winikoff, 2006b; 2009].
Figure 5.1: Implementation Overview
do not require any platform or Hermes specific features. The plans are named based on their purpose rather than actual plan type. For example, coordination plans are used to coordinate participating agents through the interaction. These plans are derived from interaction goals and are common to all agents involved in the interaction. Achievement plans are used by participating agents to achieve particular interaction goals. These plans are derived from actions and differ from agent to agent. Interface plans are derived from message descriptors and action maps, and are used to transform inter-agent messages into goals and events for intra-agent processing. For example, when a Customer agent proposes a price to a Merchant agent, the Merchant’s interface plan converts the message into a goal which is dispatched internally and the Merchant agent then tries to achieve it.

An automatic skeleton code generation tool has been created based on the developed Hermes implementation guidelines. The tool uses a syntactic representation of a Hermes interaction to produce Jadex skeleton code for coordination and achievement plans. Interface plans are not currently generated by the software tool and must be coded manually. The tool is also able to generate some of the required beliefs for the interaction.

In this chapter, the Hermes implementation scheme is firstly explained, including how interaction goal states are represented (refer to Section 5.1), and the development of interface, coordination, and achievement plans (refer to Sections 5.2, 5.3, and 5.4 respectively). In Section 5.5, a sample execution of a Hermes interaction is given to clarify the workings of the implementation artifacts. The implementation of designs which contain interaction parallelism is explored in Section 5.6 and the code generation tool, including the syntactic representation, is then presented in Section 5.7.

5.1 Interaction Goal State Representation and Beliefs

As can be seen in Figure 5.1, agent beliefs connect different plan types. The different plan types use the beliefs to pass information between themselves so they can coordinate the agents through the interaction. Beliefs are used to represent the states of interaction goals. The states are represented by a combination of three Boolean values per interaction goal: in, finished, and succeeded. The in belief indicates that the interaction goal is currently active. The finished belief is used to indicate whether the interaction goal has been completed, whilst succeeded indicates whether the interaction goal has been successful.

The interaction goal states and valid transitions between them are shown in Figure 5.2. The dashed circles represent intermediate states that have no conceptual meaning, but are necessary to change state from active to either succeeded or failed. The boolean tuple in parentheses show the values of the three beliefs in, finished, and succeeded, respectively.

In addition, for each interaction, each agent has a role which states the name of the agent’s role in the interaction (e.g. the Customer role in the e-commerce interaction). The role beliefs are generally used to determine which roles need to take action in each interaction goal. The initiator
role states which agent initiated the interaction. This is needed for interaction goals in which the initiator is inherited.

The interaction goal initiator beliefs are a series of beliefs which identify an initiator for each interaction goal. The initiator role is responsible for taking the initial action for its designated interaction role (see Algorithm 5.2, page 131), which will cause the other agents involved to take responsive actions and achieve the interaction goal.

The interaction goal retries are used to flag that their respective interaction goals are being retried. This is important as the agents will then try to achieve a different outcome than that achieved previously so that the interaction can proceed successfully. Thus, to be able to do this, an achievement plan must be able to detect whether it is being retried. If so, it will need to produce a different solution and in order to do so, it must keep a log of previously proposed solutions. For more details on the action retry failure handling mechanism, refer to Section 5.4. The remainder of the beliefs are interaction specific, including beliefs that are based on data stores from the action maps.

A summary of the agent’s beliefs, along with examples, is shown in Table 5.1. The examples are presented as key:value pairs with sample values for the e-commerce interaction.

The automatic code generation tool is able to create a number of beliefs, such as the role and initiator beliefs. Furthermore, sets of beliefs, such as the interaction goal initiators and the interaction goal states, can also be generated.

Other beliefs, such as the interaction goal retries and interaction specific beliefs (including data store beliefs), are not able to be automatically generated. However, it is envisioned that interaction goal retries and data store beliefs may be able to be generated if the syntactic representation provides further information.

This additional information can be obtained by expanding the design artifacts. For example, in the interaction goal hierarchy, the designer could mark interaction goals which are allowed to be retried and state the name of the interaction goal retry beliefs. Furthermore, data store beliefs, including interaction specific beliefs, can be added in a data store descriptor form in
CHAPTER 5. IMPLEMENTING GOAL-ORIENTED INTERACTIONS

![Table 5.1: Belief Structure and Examples](image)

which all beliefs are specified. This design information can then be included in the syntactic representation of the design artifacts and used by the software support tool to generate code. However, a disadvantage of adding all this information is that it would result in cluttering the design artifacts.

### 5.2 Interface Plans

Interface plans are used to map incoming messages to goals for an agent to achieve. For example, if a Merchant sends a message to a Customer requesting payment, the Customer’s interface plans will map that message to a goal which the Customer is to achieve.

In some cases, interface plans also change the agent’s beliefs. For example, if a Merchant is not currently involved in an interaction and a Customer sends it a RequestPrice message, the Merchant’s interface plan will trigger the Merchant’s coordination plan corresponding to the root interaction goal of the interaction by setting its in belief to true. The interface plan will then post a RequestPrice goal which will trigger one of the Merchant’s achievement plan. In this manner, interface plans are used to initiate interactions between agents.

The code generation tool is currently unable to generate interface plans as the mapping between
messages and goals is not captured in the design artifacts and must be manually coded by the programmer.

5.3 Coordination Plans

When implementing Hermes interactions on goal-plan agent platforms, such as Jadex, coordination and achievement details are implemented within plans. However, as coordination details are quite different to achievement details, and serve a different purpose, we choose to implement them separately.

In the design of Hermes interactions, coordination details are encapsulated in the interaction goals and the interaction goal hierarchy. This design artifact contains information about what needs to be done. It also includes details such as who the initiators of the interaction goals are (i.e. when a role needs to start the action sequence to achieve an interaction goal) and what sequence the interaction goals need to be achieved in, if any. Such details are implemented as coordination plans.

Achievement information is captured in the action maps. These describe how to achieve particular interaction goals. This includes what actions from one role trigger other actions in other agents, and failure handling techniques such as action retry and rollback. These details are implemented as achievement plans.

As both coordination and achievement plans are essentially just ordinary plans, there is no difference between the two in regard to the agent platform used for implementation. However, as these two types of plans serve different purposes, the structure of the plans will be different. As such, they are implemented separately as coordination and achievement plans; a clear and logical division for the programmer.

In this section, coordination plans are explained in detail, whilst achievement plans are explained in the succeeding section.

Coordination plans, derived from interaction goals, are a common set of plans that guide the agents through their interactions. That is, each agent involved in the interaction will have these plans in their plan library. There are two variations of coordination plans, compound and atomic. Compound coordination plans are based on compound interaction goals, i.e. those that are composed of other interaction goals, such as Trade, Agree, and Exchange (refer to Figure 3.4), whilst atomic coordination plans are derived from interaction goals that are not composed of other interaction goals (i.e. atomic interaction goal), such as Determine Availability, Negotiate Details, Negotiate Price, Transfer Goods, Payment, and Send Receipt.

Compound coordination plans are involved with coordination between themselves and other coordination plans. Atomic coordination plans, however, deal with coordination between themselves and achievement plans.

In our methodology for implementation of a Hermes interaction, there is one coordination plan for each interaction goal. Algorithm 5.1 presents an example of a generic compound coordination
plan. As this is a generic coordination plan, it is assumed that it is derived from an interaction goal named IG.

Algorithm 5.1  Generic (Sequential) Compound Coordination Plan for Interaction Goal IG

Require: in[IG] == true

1: terminate = false
2:
3: while moreChildIGs() and not terminate do
4:     // Get beliefs for next IG
5:     ChildIG = nextChildName()
6:
7:     // Coordination
8:     in[ChildIG] = true
9:     waitFor(finished[ChildIG] and not in[ChildIG])
10:    if not succeeded[childIG] then
11:       terminate = true
12:    end if
13: end while
14:
15: if all Child IGs succeeded then
16:    succeeded[IG] = true
17: end if
18:
19: // Synchronization (with other Coordination plans)
20: finished[IG] = true
21: in[IG] = false

When an agent is created, all of its three beliefs about each interaction goal state are set to false. In regard to Algorithm 5.1, these following beliefs are false before the algorithm begins: in[IG], finished[IG] and succeeded[IG]. To trigger a compound coordination plan, its in belief is set to true. In the case of Algorithm 5.1, it is when the in[IG] belief changes to true (as shown by the Require statement).

When a compound coordination plan is executed, its first step is to begin the achievement of its sub-coordination plans (i.e. child interaction goals) in the specified order\(^1\). Algorithm 5.1 shows an interaction in which all child interaction goals are to be achieved in sequence (denoted by the while loop). As such, the IG coordination plan begins by retrieving the name of the next child interaction goal to be achieved (line 5). The coordination plan then sets the in belief of the

\(^1\)In this example, a sequential order is assumed. Refer to Section 5.6 for the implementation of interaction parallelism.
child interaction goal to \text{true} (line 8), which allows the child interaction goal’s coordination plan to execute. The IG coordination plan then waits until the child interaction goal is achieved (line 9), and then attempts to achieve the following child interaction goal if the current one has been achieved successfully (lines 10–12).

When all its child interaction goals have been successfully achieved, the IG coordination plan sets its succeeded belief to \text{true} (line 15–17). The last part of the compound coordination plan is to synchronize itself with the other coordination plans. That is, it sets its \text{in} and \text{finished} beliefs (in this case \text{in}[IG] and \text{finished}[IG]) to \text{false} and \text{true} respectively to signal its completion.

Algorithm 5.2 is an example of a generic atomic coordination plan. As with compound coordination plans, atomic coordination plans are triggered when their \text{in} beliefs change to \text{true} (refer to Require statement in Algorithm 5.2).

\begin{algorithm}
\caption{Generic Atomic Coordination Plan for Interaction Goal IG (If succeeded[IG] is \text{true}, finished[IG] will also be \text{true} as the last time the plan executed, it would have been successful and the plan would have finished appropriately.)}
\begin{algorithmic}[1]
\Require in[IG] == true
1: \If not succeeded[IG] \then
2: \If role == initiator \then
3: \hspace{1em} dispatch(new triggerInitialActionGoal())
4: \hspace{1em} \EndIf
5: \EndIf
6: \hspace{1em} \Statex // Synchronisation (with Achievement plans)
7: \hspace{1em} waitFor(finished[IG])
8: \hspace{1em} in[IG] = false
\end{algorithmic}
\end{algorithm}

The first step of an atomic coordination plan is to execute the initial action (in the relevant action map) in an attempt to achieve itself. However, before that action is triggered, the atomic coordination plan ensures that it has not already been achieved (refer to line 1). This is important in situations where rollbacks are issued – there is no need to achieve a coordination plan that is already achieved. Furthermore, the atomic coordination plan ensures that only the initiator of this interaction goal begins the interaction (refer to line 2). If these conditions are met, the coordination plan triggers the initial action by dispatching the appropriate goal (line 3).

Once the goal has been dispatched, the atomic coordination plan must wait until the action (which is likely to trigger a series of other actions) completes. When the series of actions is completed, the finished belief of the current interaction goal will be set to \text{true}. Thus, as part of its synchronization with the achievement plans (implementations of the actions), the atomic coordination plan waits for its finished belief to be set to \text{true} (line 8). The last part of the atomic coordination plan is to set its \text{in} belief to \text{false}, signifying that it has been completed.
The current version of the code generation tool is able to generate full code for compound coordination plans, and partial code for the atomic coordination plans which must be augmented with application-specific code. Refer to Listings A.1 and A.2 in Appendix A for examples of automatically generated coordination plans. For more details on the generation of coordination plans, refer to Section 5.7.2.

5.4 Achievement Plans

Achievement plans, based on actions from the action maps, are used by the interacting agents to accomplish the individual agent steps necessary to achieve a particular interaction goal. These will differ from one interaction to another. Therefore, achievement plans usually contain interaction-specific steps.

Algorithm 5.3 presents an example of a generic achievement plan. For an achievement plan to begin execution, it must be triggered by an appropriate goal event, as shown by the Require statement in Algorithm 5.3. Once triggered, the achievement plan must ensure that it is in the correct context, i.e. its interaction goal is active, before beginning to execute (refer to line 2). When the achievement plan is in the correct context, it executes interaction-specific code (lines 4 and 5).

**Algorithm 5.3 Generic Achievement Plan**

**Require:** actionTriggerGoalEvent

1: // Synchronisation (with Coordination plan)
2: waitFor(in[IG])
3: 
4: // Achieve IG (application specific)
5: ...
6: if action achieves IG then
7:  succeeded[IG] = true // Action achieves IG
8: end if
9: 
10: // Finish IG, only done if action is final
11: // Synchronisation (with Coordination plan)
12: if action is final then
13:  finished[IG] = true
14: end if

If the achievement plan represents an action that achieves the interaction goal (i.e. a final action that terminates with success), then the interaction goal’s succeeded belief is set to true (lines 6–8). Furthermore, achievement plans representing final actions have a synchronization
section which sets the **finished** belief of the interaction goal to **true** signaling the completion of the interaction goal (lines 10–14).

The implementation of action failure handling mechanisms, *termination* and *action retry*, are simple and straightforward. When an action fails (i.e. an achievement plan fails), there are two options: terminate the interaction or attempt to recover by retrying the action with different parameters.

This first option, termination, is the simplest. In such a case, the programmer will need to add actions to request termination and to terminate the interaction.

For example, in Figure 3.7 (on page 77), the *Customer* will terminate the interaction if the *Merchant* rejects its proposed price. To implement this termination, the programmer will have to equip the *Customer* with a **Request Termination** action. This will allow the *Customer* to send a message to the *Merchant* to terminate the interaction.

The *Merchant* will need to be equipped with a **Terminate On Request** action, which upon receiving the *Customer*'s request will terminate the interaction and reply to the *Customer*. In turn, the *Customer* will require a **Terminate** action, which it will use to end the interaction. The **Terminate** action ends the interaction by setting the three interaction goal state beliefs (i.e. **in**, **finished**, and **succeeded**) of each interaction goals to **false**.

This sequence of **Request Termination**, **Terminate On Request** and **Terminate** actions can be quite easily added to the action map, however, to avoid unnecessarily cluttering the diagram, the **Terminate Interaction** action on Figure 3.7 is understood to represent this sequence of actions. This is similar to the sequence used to specify rollback, as depicted in Figure 5.3.

Although effective, the termination of an interaction is not always desirable. In that situation, an action retry might be more appropriate. An action retry is implemented by adding an action to form a loop in the sequence of actions. For example, the **Re-propose Price** action in Figure 3.9 is an action retry. It allows the *Customer* to retry the **Propose Price** action with different parameters, in this case a higher price.

It is important to ensure that the action retry does not force the interaction in an endless loop. Thus, an escape condition must be provided for every action retry. If the *Customer* cannot propose a higher price, it can end the action retry loop by use of the **Terminate Interaction** action (refer to Figure 3.9).

Implementing the rollback failure handling mechanism, which addresses interaction goal failure, is more complicated than implementing terminations or action retries. Algorithm 5.4 is an example of a rollback achievement plan in which a *Customer* is rolling back from the **Negotiate Price** interaction goal to the **Negotiate Details** interaction goal as per the action map in Figure 3.10 (on page 83). The comments (in bold) present a general plan for rolling back with the code showing how the *Customer* rolls back in this particular example.

In general, a rollback is implemented by “saving” the interaction in a particular state and re-starting the entire interaction. The “saving” of the interaction is done by setting the interaction
Algorithm 5.4 Customer Rollback Plan (from Negotiate Price to Negotiate Details)

Require: rollbackGoalEvent

1: // Synchronise (with Coordination plan)
2: waitFor(in[IG])
3:
4: // 1. Terminate current IG
5: succeeded[IG] = false
6: finished[IG] = true
7: in[IG] = false
8:
9: // 2. Wait for apex IG to terminate, e.g. Trade
10: waitFor(finished[topIG] and not in[topIG])
11:
12: // 3. Set appropriate beliefs to re-start interaction
13: // to begin at desired IG (shortcut)
14: // 3.1. Reset current IG beliefs
15: finished[IG] = false
16:
17: // 3.2. Set beliefs of IG to begin next interaction from (shortcut),
18: // e.g. Negotiate Details
19: succeeded[startIG] = false
20: finished[startIG] = false
21: in[startIG] = true
22:
23: // 3.3. Set beliefs for “retry” attempt
24: retryNegDetails = true
25:
26: // 4. Re-start interaction, set “in” belief of apex interaction goal to “true”
27: in[topIG] = true
28:
29: // 5. Notify relevant agents
goal to which the agent wishes to roll back to be attempted next. This is done by setting its in belief to true and both its finished and succeeded beliefs to false, which essentially flags the interaction goal as active, but not yet completed (lines 17–20 in Algorithm 5.4). Thus, it will be attempted next (unless there are other active but uncompleted interaction goals preceding it).

In Algorithm 5.4, the agent must firstly ensure that it is in the correct context (i.e. its current interaction goal is active) before it can carry out the rollback. In this case, the Customer must wait until the Negotiate Price interaction goal is active (lines 1 and 2).

The agent then terminates the current interaction goal by setting its in and succeeded beliefs to false and its finished belief to true. This will cause the interaction to fail, which the agent waits for (lines 9 and 10).

After the interaction has failed, the agent sets the appropriate beliefs to re-start the interaction at the desired interaction goal (this is specific to the particular rollback) and then re-starts the interaction.

In Algorithm 5.4 the Customer does not need to notify any agents (line 28) as it is the last agent to roll back (refer to Figure 5.3). However, in the case of the Merchant in Figure 5.3, it would have to notify the Customer that it has completed its rollback so that the Customer can then begin its rollback.

As with terminations, a similar sequence of actions is, by convention, understood by the Rollback to Negotiate Details action in Figure 3.10 (on page 83). That is, the Customer will request that the Merchant roll back to Negotiate Details. The Merchant will roll back and reply to the Customer, who will then also roll back to the Negotiate Details interaction goal. Refer to Figure 5.3 for a diagrammatical depiction of this sequence of actions.

The code generation tool is currently only able to generate skeleton code for achievement plans. As termination, action retry and rollbacks are simply actions, the code generation tool is able to produce the skeleton code for these failure handling mechanisms, but the programmer is required to add code specifying what the actions should do (such as to which interaction goal a specific rollback should return to). Refer to Listing A.3 in Appendix A for an example of an automatically generated skeleton achievement plan. For more details on the generation of skeleton achievement plans, refer to Section 5.7.2 on the skeleton code generation tool.

5.5 Sample Execution

To better understand the interaction between different plan types, a simple example execution is provided in this section. In this example, based on the e-commerce scenario used in Chapter 3, the Customer is attempting to purchase a monitor at the maximum price of $100 with the following colour preferences: red, blue, yellow, and green and the Merchant is selling blue and yellow monitors at the minimum prices of $110 and $100 respectively. In this situation, for a successful sale to occur, the Merchant must sell a yellow monitor to the Customer at $100.
Figure 5.3: Rollback Sequence
The following demonstrates how such an interaction executes on an implementation based on a goal-oriented Hermes design. Figure 5.4 presents the initial execution steps graphically.

The interaction begins with the Customer receiving a request to start the interaction. This is handled by its interface plan, Handle Requests, which flags a boolean belief (i.e. inTrade) to start the interaction. The Customer then enters the Trade interaction goal, then the Agree interaction goal, and then Determine Availability interaction goal (based on the interaction goal hierarchy, refer to Figure 3.4, page 67).

The Determine Availability coordination plan executes and triggers its achievement plan, Request Availability, which executes and sends a message to the Merchant, enquiring about the availability of a monitor.

The message is received by the Merchant’s interface plan and is converted into a checkAvailability goal which triggers the Merchant’s Check Availability plan. Although the Check Availability plan is triggered, it does not execute as it is waiting for the Merchant to enter the Determine Availability interaction goal. Since the Merchant has not started the Trade interaction, when it receives the message from the Customer, its Handle Requests interface plan sets its inTrade belief to true and starts the interaction (at the Trade interaction goal). The interaction then moves into the Determine Availability interaction goal through the Agree interaction goal. The Check Availability plan then executes and sends a message to the Customer, informing it that there are monitors available. The Determine Availability interaction goal is then successfully achieved for the merchant.

The Customer’s interface plan, Handle Requests, handles the message and converts it into a goal for internal agent processing. The Determine Availability interaction goal is successfully achieved for the Customer. It moves into the Negotiate Details interaction goal and its Propose Details achievement plan is triggered. The Propose Details plan sends a message to the Merchant to request a red monitor. As the Merchant does not have red monitors, it sends a rejection message to the Customer. The Customer’s Re-propose Details plan is triggered (after the message is converted to a goal by the Customer’s interface plan). The Re-propose Details plan, which is based on an action retry, then creates a new goal to trigger the Propose Details plan to send a message requesting a blue monitor.

As the Merchant sells blue monitors, it returns a positive reply to the Customer and moves into the Negotiate Price interaction goal. The Customer receives the message and also moves into the Negotiate Price interaction goal.

The negotiations over the price of the monitor proceed similarly to the negotiation of the colour of the monitor. When the Merchant rejects the Customer’s highest price of $100 (as the Merchant’s minimum is $110), the Customer’s Re-propose Price plan triggers the Request Rollback To Negotiate Details plan (refer to Figure 3.10, page 83), which sends a rollback request to the Merchant. The Merchant then uses its Rollback To Negotiate Details plan to return to the Negotiate Details interaction goal and notifies the Customer that it has successfully rolled back.
CHAPTER 5. IMPLEMENTING GOAL-ORIENTED INTERACTIONS

Figure 5.4: Sample Execution
CHAPTER 5. IMPLEMENTING GOAL-ORIENTED INTERACTIONS

The Customer then executes its Rollback To Negotiate Details plan to return to the Negotiate Details interaction goal.

The Customer and the Merchant re-negotiate the colour of the monitor and settle on yellow. The interaction then proceeds to the negotiation of the price and is able to terminate successfully.

In practice, a better way to implement this interaction would be for the Merchant to send the Customer a list of monitors and prices, however, this example better illustrates the process and mechanics of a Hermes implementation.

5.6 Implementing Parallelism

Although very similar, there are some differences in the implementation of parallel designs compared to the typical sequential Hermes designs. While the majority of the work on implementing parallelism is left as future work, in this section, implementation details of some specific cases and associated issues are described. This includes the implementation of parallel interaction goals (Section 5.6.1), parallel actions (Section 5.6.2), and homogeneous instance parallelism (Section 5.6.3). For a detailed description of designing parallel interactions, refer to Section 3.7.

Parallel interaction goals are a coarse level of parallelism in goal-oriented interactions as they allow large portions of the interaction (i.e. entire action maps) to be executed in parallel. In contrast, parallel action sequences are a finer level of parallelism as they allow sequences of actions to be executed in parallel within action maps. The combination of both parallel interaction goals and action sequences allow for many possibilities of parallelism in goal-oriented interactions. These possibilities are further increased with homogeneous instance parallelism, a type of parallelism that specifies what the different instances of the same agent type can perform within an interaction.

5.6.1 Implementing Parallel Interaction Goals

As interaction goals have a one-to-one mapping to coordination plans, only coordination plans are affected by parallel interaction goals. More specifically, only compound coordination plans are affected as only compound interaction goals can be designated as the start and end point of parallelism in interaction goal hierarchies. That is, they coordinate the parallel execution of their sub-goals. A generic example of a compound interaction goal with two atomic parallel child interaction goals is shown in Figure 5.5.

The algorithm for a generic parallel compound coordination plan is shown in Algorithm 5.5. This algorithm is very similar to its sequential counterpart (refer to Algorithm 5.1, page 130). Both algorithms have a coordination and synchronization section. The synchronization sections are exactly the same. The only difference between the two types of plans lies in the coordination section.

As expected, the sequential coordination plan achieves its sub-goals one after another whereas the parallel coordination plan achieves its sub-goals concurrently. In the parallel coordination plan
all sub-goals are started sequentially (i.e. one immediately after the other) as shown in steps 2 and 3 in Algorithm 5.5 but they all run in parallel. Although Algorithm 5.5 shows only two sub-goals being achieved in parallel, it is possible to achieve more interaction goals in parallel.

Algorithm 5.5 waits for all interaction goals to complete before progressing. This may be inefficient in certain situations as it might be desirable to abort all other parallel interaction goals if one of them fails. For example consider a manufacturing example in which the required parts are sought out in parallel. If it is not possible to obtain one particular part, then it may be desirable to abort interaction goals which seek out the other parts in parallel. This can be achieved by customizing the coordination plan.

### 5.6.2 Implementing Parallel Actions

Unlike parallel interaction goals, the implementation of parallel actions is significantly more complex. This is especially true for the termination of parallel action sequences (i.e. merging multiple parallel action sequences into a single action sequence).

Typically, but not always, the actions executed in parallel will be from different agents. Thus, initiating and terminating parallel action sequences will involve sending, receiving and waiting for messages in parallel.

An abstract example of parallel action sequences (based on Figure 3.16, page 95) is shown in Figure 5.6. Although an abstract example is used, the implementation details are based on actual interactions in which parallel actions were implemented.

Initiating a parallel sequence (i.e. splitting a single action sequence into multiple parallel action sequences) is trivial as it is simply a matter of one agent sending multiple messages to different
CHAPTER 5. IMPLEMENTING GOAL-ORIENTED INTERACTIONS

Figure 5.6: Parallel Actions
Algorithm 5.5 Generic (Parallel) Compound Coordination Plan

Require: in[IG] == true

1: // Coordination
2: in[childIG01] = true
3: in[childIG02] = true

4:
5: waitFor((finished[childIG01] and not in[childIG01]) and
6: (finished[childIG02] and not in[childIG02]))

7:
8: if succeeded[childIG01] and succeeded[childIG02] then
9:   succeeded[IG] = true
10: end if
11:
12: // Synchronization (with other Coordination plans)
13: finished[IG] = true
14: in[IG] = false

agents at the same time (if the parallelism is with different agents) or dispatching multiple internal goals (if the parallelism is within the same agent) or a combination of both. For example, in Figure 5.6, Agent-A simply needs to send messages to both Agent-B and Agent-C to initiate the parallel action sequences. This, in effect, is the implementation of the parallel split.

Terminating parallel action sequences is more difficult, especially as some parallel action sequences may fail whilst others succeed. There can be multiple terminations (i.e. parallel merges) to parallel action sequences. Typically, there will be at least two; one for successful merges and the other for failure. The Process Success action in Figure 5.6 is an example of a successful parallel merge and is executed when all parallel action sequences have been successfully achieved, whilst the Process Failure action is an example of a failed parallel sequence and is executed when one or more parallel action sequences fail. It is important to note that these actions are executed after the parallel merge and do not in themselves merge the parallel action sequences.

Intermediate achievement plans (which are not shown on the action maps) are required to implement the parallel merge (denoted as parallel bars with incoming and out-going directed lines on the action maps). The implementation of the successful parallel merge (which occurs prior to the Process Success action) is shown in Figure 5.7. As depicted, the Process Success achievement plan occurs after the parallel merge and is independent of the parallelism. Process 01 Success and Process 02 Success are two intermediate achievement plans to handle the individual parallel action sequences. The intermediate plans are used to implement the synchronization point at the end of the parallel action sequences.

The intermediate achievement plans handle the individual parallel action sequences by pro-
cessing the responses and setting appropriate beliefs to terminate the parallelism, after which the Process Success achievement plan is executed. Algorithm 5.6 shows an example of an intermediate achievement plan to handle successful parallel action sequences.

As the Process Success achievement plan is only to be executed after all parallel sequences have terminated successfully, the intermediate achievement plans use two beliefs to signify the end of the successful parallelism. For example, Process 01 Success (refer to Algorithm 5.6) will set both sequence01Terminated and sequence01Success to true (lines 6 and 7), and Process 02 Success will also set similar beliefs to true. The Process Success achievement plan will only execute when all parallel action sequences are achieved successfully. That is, it is triggered when the condition sequence01Success and sequence02Success becomes true.

Algorithm 5.6 Intermediate Achievement Plan: Process 01 Success

Require: actionTriggerGoalEvent

1: // Synchronization (with Coordination plan)
2: waitFor(in[IG])
3: 
4: // Application specific code
5: 
6: sequence01Success = true
7: sequence01Terminated = true
8: 
9: // Synchronization (with Coordination plan)
10: succeeded[IG] = true
11: finished[IG] = true

The merging of failed parallel sequences is implemented as shown in Figure 5.6. As with the merging of successful parallel sequences, merging failed parallel sequences will require intermediate achievement plans, namely Process 01 Failure and Process 02 Failure. Similarly to their “success” counterparts, Process 01 Failure and Process 02 Failure set two beliefs to acknowledge the failure
and termination of their parallel action sequences. For example, Process 01 Failure will set the beliefs sequence01Terminated and sequence01Failure to true and Process 02 Success will also set similar beliefs to true.

![Figure 5.8: Merging Failed Parallel Action Sequences](image)

The Process Failure achievement plan will only be executed if all parallel sequences have terminated and at least one of them failed. That is, it is triggered when the condition (sequence01Failure or sequence02Failure) and (sequence01Terminated and sequence02Terminated) becomes true. There are a variety of ways to handle successes and failures of parallel sequences\(^2\), however, this is the only one currently implemented in our work.

### 5.6.3 Implementing Homogeneous Instance Parallelism

Given that each agent’s logical control is separate, i.e. agent instances execute in parallel (or concurrently) to each other, the implementation of homogeneous instance parallelism is not particularly difficult. Furthermore, there are some commonalities between homogeneous instance parallelism and action parallelism. In this section, the implementation of homogeneous instance parallelism is explained based on the auction action map with homogeneous instance parallelism in Figure 3.17 (in Chapter 3, page 98).

To begin homogeneous instance parallelism, actions in multiple agent instances must be triggered in parallel. For example, in Figure 3.17 the Auctioneer starts the parallelism by triggering the Valuate Item action from multiple Bidders in parallel. Triggering an action in another agent is implemented as one agent sending a message to another. Thus, as the Auctioneer is to trigger actions in multiple instances of the Bidder agent type, in terms of implementation, this translates to the Auctioneer sending multiple messages in parallel to multiple Bidders. In this particular case, the Auctioneer needs to trigger the same action (i.e. Valuate Item) in the different instances of the Bidder. However, there are situations in which different actions need to be triggered in different instances.

\(^2\)As previously mentioned, there are four options to handle failure in parallelism. Only one option is presented here, for the other three options, refer to [Age, 2005a]
One example in which different actions are to be triggered in the different instances of the Bidder in parallel is when the Auctioneer is to declare a winner (refer to Figure 3.17). The Auctioneer needs to trigger the winning Bidder’s Register Won action whilst it needs to trigger the other Bidders’ Register Lost actions in parallel. In terms of design, a combination of action parallelism and homogeneous instance parallelism notations are used. However, in terms of implementation, this simply results in the Auctioneer sending different messages in parallel. The Auctioneer sends a single winning message to the winning Bidder and triggers its Register Won action whilst the Auctioneer also sends a single unsuccessful message to each of the remaining Bidders to trigger their Register Lost actions in parallel.

Merging the parallelism may be more complicated than initiating it, in which case intermediate achievement plans can be used to synchronize the parallel sequences before the interaction progresses into a single sequence.

In parts of the interaction where there are different possible replies, such as between the Bidders using their Place No Bids and Place Bid actions, and the Auctioneer using its Collect Bids action, there needs to be an intermediate achievement plan between each possible reply type and the action which merges the parallelism. That is, the Auctioneer has a Collect No Bids intermediate achievement plan to handle the messages from the Bidders’ Place No Bids actions and a Collect Placed Bids intermediate achievement plan to handle the Bidders’ messages from their Place Bid action.

These intermediate achievement plans will register the number of unique Bidders and their replies. The Collect Bids achievement plan will then be triggered when all Bidders have replied.

5.7 Implementation Software Support Tool

The main purpose of the automatic code generation tool is to facilitate and ease the implementation process. As the implemented plans have a common structure, it is possible to automatically generate the implementation code. This then frees the programmer to concentrate on the more interesting and complicated parts of the implementation.

To ensure that the automatic code generation is independent of the design support tool, a syntactic representation of Hermes interactions is required. The automatic code generation tool is able to accept a syntactic representation of a Hermes design, which is separate from the design tool (or any design tool implemented in the future) and transform it into agent platform code.

A syntactic representation of Hermes interactions is firstly presented, followed by a description of the automatic code generation tool.

5.7.1 Syntactic Representation of Hermes Goal-Oriented Interaction

The syntactic representation of Hermes interactions is defined in XML. XML was chosen as it is able to capture and represent interactions adequately and it can be easily manipulated by XSLT
to automatically generate the required code.

Creating a distinct syntactic representation for Hermes interactions has a number of advantages; the foremost being a separation between the design tool and the automatic code generation tool. This approach allows the design tool to use any type of internal representation desired and, as long as it is able to generate output in the defined XML representation, the design can be used to automatically produce implementation code.

The syntactic representation can also be seen as a standard representation for Hermes interaction tools. If future software support tools are created for Hermes, they can be compatible by conforming to the XML syntactic representation.

The syntactic representation of Hermes interactions is constrained by an XML schema. Rather than explain the XML schema, we instead present a less verbose grammar which represents the rules enforced by the XML schema and a sample interaction. The XML schema is presented in Listing B.1 in Appendix B.

Listing 5.1 presents the grammar for Hermes interactions whilst Listing 5.2 shows a sample Hermes interaction. As the first line of Listing 5.1 shows, an interaction has a `<name>`, and three main sections, `<agents>`, `<igh>`, and `<ams>`. 

```
1 <interaction> ::= INTERACTION <name> <agents> <igh> <ams>
2 <name> ::= variable
3 <location> ::= variable
4 <beliefType> ::= variable
5 <initialValue> ::= variable
6 <agents> ::= AGENTS <agent> <agentList>
7 <agentList> ::= <agent> | <agent> <agentList>
8 <agent> ::= AGENT <name> ROLE <name>
9 <igh> ::= IGH <ig>
10 <ig> ::= IG <name> INITIATOR <name> | IG <name> INITIATOR <name> <igs>
11 <igs> ::= <mode> <igList> END-IGS
12 <mode> ::= SEQUENTIAL | PARALLEL
13 <ams> ::= ACTION-MAPS <am>
14 <am> ::= ACTION-MAP <name> ACHIEVES-IG <name>
15 <roles> ::= ROLES <role> <roleList>
16 <roleList> ::= <role> | <role> <roleList>
17 <role> ::= ROLE <name>
18 <actions> ::= ACTIONS <actionList>
19 <actionList> ::= <action> | <action> <actionList>
20 <action> ::= ACTION <name> TYPE <type> OWNER <name> <data> <rollback> <loadCode>
21 <type> ::= INDEPENDENT | CAUSED | FINAL-CAUSED | FINAL-INDEPENDENT
22 <data> ::= READS-FROM <name> <dataList>
23 <write> ::= WRITES-TO <name> <dataList>
24 <dataList> ::= <name> | <name> <dataList>
25 <rollback> ::= ROLLBACK <name> | λ
26 <loadCode> ::= LOAD-CODE <location> | λ
27 <causalities> ::= CAUSALITIES <causalityList>
28 <causalityList> ::= EMPTY | <causality> <causalityList>
29 <causality> ::= CAUSALITY TYPE <causalityType> ORIGIN <name> DESTINATION <name> LABEL <name>
30 <causalityType> ::= SINGLE | PARALLEL-MERGE | PARALLEL-SPLIT | BROADCAST-MERGE | BROADCAST-SPLIT
```
The first section, AGENTS, simply lists all the agents involved in the interaction and the roles that they undertake. The IGH section represents the interaction goal hierarchy of the interaction. It lists all the interaction goals along with hierarchical details and the order in which they need to be achieved (as described in Section 5.3).

The IGH section is followed by an ACTION-MAPS section, in which all action maps are described. The action map descriptions are composed of five parts: ACHIEVES-IG, ROLES, ACTIONS, CAUSALITIES, and DATASTORES. ACHIEVES-IG states which interaction goal this action map is attempting to achieve. The ROLES section lists all the roles involved in this particular action map. The ACTIONS section lists all actions involved in the action map.

Actions must belong to a role involved in the action map (its OWNER) and must also have a TYPE, which can be one of INDEPENDENT, CAUSED, FINAL-CAUSED, or FINAL-INDEPENDENT (refer to line 28 of Listing 5.1).

The DATA section of an ACTION is optional and when present it lists all the datastores that the ACTION requires access to. The type of access the ACTION requires is specified by READS-FROM and WRITES-TO. It is possible to have both read and write access to the same datastore.

ROLLBACKs can be used to specify to which interaction goal, if any, an ACTION rolls back to. Specifying ROLLBACKs for ACTIONs is optional.

The optional LOAD-CODE section of an ACTION allows the support tool to incorporate action-specific code in its code generation. That is, the implementor can write action-specific code (in this case Java snippets as the generated code is for the Jadex agent platform) in a separate file which the code generation tool will load, read, and add to the code it generates. This allows for (one-way) re-generation of action-specific code in plans without overwriting changes made.

The CAUSALITIES section of an action map lists all causalities in the action map. Causalities must have an ORIGIN and a DESTINATION, both of which are action names. Causalities also have a TYPE, which is one of SINGLE, PARALLEL-MERGE, PARALLEL-SPLIT, BROADCAST-MERGE, or BROADCAST-SPLIT (refer to lines 43 and 44 of Listing 5.1).

Currently, only the SINGLE type is supported. The other types are provisional values to be used in the future to support automatic generation of parallel code.

The last section of an action map definition is the DATASTORES section, which lists all the datastores present on the action map. A DATASTORE, like an ACTION, must belong to an AGENT (its OWNER). DATASTOREs are a list of BELIEFS which require a name, type, and initial value (refer to line 52 in Listing 5.1).
CHAPTER 5. IMPLEMENTING GOAL-ORIENTED INTERACTIONS

A sample interaction using the grammar described in Listing 5.1 is shown in Listing 5.2.

```
INTERACTION E-Commerce
AGENTS
  AGENT Anna ROLE Customer
  AGENT Bob ROLE Merchant
IGH
  IG Trade INITIATOR initiator
    SEQUENTIAL
      IG Agree INITIATOR initiator
        SEQUENTIAL
          IG DetermineAvailability INITIATOR initiator
          IG NegotiateDetails INITIATOR initiator
          IG NegotiatePrice INITIATOR initiator
        END-IGS
      IG Exchange INITIATOR initiator
        SEQUENTIAL
          IG TransferGoods INITIATOR Merchant
          IG Payment INITIATOR Customer
          IG SendReceipt INITIATOR Merchant
        END-IGS
    END-IGS
ACTION-MAPS
  ACTION-MAP DetermineAvailabilityActionMap
    ACHIEVES -IG DetermineAvailability
    ROLES
      ROLE Customer
      ROLE Merchant
    ACTIONS
      ACTION ProposePrice
        TYPE INDEPENDENT
        OWNER Customer
    CAUSALITIES
      CAUSALITY TYPE SINGLE
      ORIGIN ConsiderPrice
      DESTINATION CompleteInteractionGoal
      LABEL Price Accepted
```

Listing 5.2: Sample Hermes Interaction

For an example of an XML input file, refer to Appendix B. As the XML input file is quite verbose, as many XML documents are, the file is not shown in its entirety. Instead, representative parts of the file are displayed. Furthermore, the file has been split into two separate figures to facilitate its display. Listing B.2 shows the main XML input file, including the interaction goal hierarchy definition. Listing B.3 shows only one partial action map definition, which should normally be together with Listing B.2.

5.7.2 Skeleton Code Generation Tool

An overview of the skeleton code generation tool is presented in Figure 5.9. The code generation tool takes an XML representation of a Hermes goal-oriented interaction, as described in Section 5.7.1, and produces multiple files of Jadex skeleton code in a structured directory hierarchy. The generation of code is done by an XSL transformation.

To generate the skeleton code, the XSL transformation files and the XML input file are run in an XSLT processor such as Saxon. Currently, the code generation tool is only able to generate
Jadex code, however, to produce skeleton code for a different agent platform is a matter of creating an XSL transformation for the desired agent platform.

The code generation tool is composed of two primary parts. The first is an XML schema (refer to Listing B.1 in Appendix B) which is used to validate and enforce the structure of the XML input file, as explained in Section 5.7.1.

The second part of the code generation tool is what performs the actual XSL transformation. This part of the code generation tool is divided into three smaller logical components: ADF Generator, Coordination Plan Generator, and Achievement Plan Generator.

The ADF Generator creates Jadex-specific files called Agent Definition Files (ADF). The Coordination Plan Generator and Achievement Plan Generator generate the code for the coordination and achievement plans respectively.

In Jadex, every agent is defined by an ADF, which is used to specify the contents of the agent. An ADF for a Hermes agent is composed of three main sections to define the agent’s beliefs, goals and plans. These sections state what beliefs a particular agent has, what type of goals and events it can handle, and what plans the agent has at its disposal. The ADF also specifies the conditions and contexts under which a plan is applicable.

The code generation tool is not able to generate single agent goals, although it can generate beliefs and plans. In regard to plan generation, only coordination and achievement plans can be
generated; interface plans are not generated. Single agent goals and interface plans are not able to be generated as the design artifacts do not contain these (implementation) details. In practice, this means that the programmer will need to implement interface plans. This will include accepting messages and dispatching appropriate single agent internal goals.

Generated beliefs include those about the interaction, such as the agent’s role in the interaction, and beliefs about the initiators of the interaction goals. Also generated are beliefs for state representation of interaction goals (i.e. the in, finished, and succeeded beliefs).

Although the grammar shown in Listing 5.1 allows for representation of datastores, the current version of the code generation tool does not generate these beliefs. This is left as future work and it is not anticipated that it will be difficult.

The Coordination Plan Generator uses the details from IGH section of the input file (refer to Listing 5.2). This includes information such as the interaction goal names, decomposition (parent-child) relationships, and the sequence in which the interaction goals are to be achieved.

As can be expected, the produced compound and atomic coordination plans follow the algorithms outlined in Algorithms 5.1 and 5.2 respectively.

Currently, the code generation tool is able to produce the complete code for compound coordination plans. However, only skeleton code is produced for atomic coordination plans as the code to trigger the initial action requires the use of goals, which the tool is unable to generate. Thus, the code generation tool produces a comment to remind the programmer to add the required code. Refer to Listings A.1 and A.2 in Appendix A for examples of generated coordination plans.

Achievement plans are produced by the Achievement Plan Generator, which gathers the required details from the ACTION-MAPS section of the input file (refer to Listing 5.2). The majority of the required information is obtained from the ACTIONS section. The CAUSALITIES section is currently not used, however, it is there as a placeholder for future versions. Furthermore, it is required to present an adequate representation of a Hermes interaction.

In its current state, the code generation tool is only able to produce skeleton code for the achievement plans (refer to Listing A.3 in Appendix A, an example of a generated achievement plan). This is primarily due to the nature of the achievement plans; the core of the achievement plans require application-specific details. These details are not available in the XML representation of Hermes interactions as they are to represent design artifacts, and not implementation details. It is possible to modify the syntactic representation to allow for such details, however, the syntactic representation is intended to be used as both a clean representation of Hermes design diagrams and machine-understandable code from which implementation code can be generated. That is, the focus of the syntactic representation is on design artifacts rather than implementation details. As such, adding implementation-specific details, which will clutter the XML input files, is not desirable.

Although the tool can only generate skeleton code for the achievement plans, it guides the programmer to add required code by use of instructions and examples (refer to Listing A.2 from
Appendix A on page 175) which are specific to individual achievement plans. Furthermore, the programmer is able to fill in the code that the tool cannot generate in a separate file which can be loaded while the skeleton code is being generated. This allows the programmer-specified code to be incorporated with the skeleton code created by the tool in the generation of code.

Currently, the code generation tool does not produce interface plans, which map messages to goal events. However, the tool can be improved to produce these. This would involve capturing additional information, such as which messages map to which goal events.

One possibility is to improve the design software support tool and syntactic representation of Hermes interactions to allow the designer to specify messages based on the message descriptors. To allow for the generation of interface plans, the improvements should also allow the designer to specify mappings between messages and goal events, including which data item from messages map to which goal parameters.

The initial messages of the interaction will need to set the in belief of the top-most interaction goal to true so that the interaction begins. Initial messages can be determined from the action maps or, alternatively, the designer can set an option on messages to generate such code.

Although it is possible to generate the interface plans, only skeleton code can be produced as, in some cases, the implementor will need to make changes to the code. For example, an interface plan might need to dispatch different goal events depending on the data values contained in messages or interaction-specific beliefs, such as customer-name, may need to be set before dispatching a goal. It is possible to modify the interaction syntax to allow for these, however, this would clutter design by incorporating both design and implementation details, which is not desirable.

5.8 Summary

An explanation of how Hermes goal-oriented design can be implemented was provided in this chapter. Although Hermes interactions have only been implemented on the Jadex agent platform, it is possible to implement the interactions on any goal-plan platforms as the implementation scheme does not use any platform-specific features.

The implementation scheme explained how interaction goal states can be represented using beliefs. It also showed how the interaction goals can be mapped to coordination plans which are used to coordinate the agents throughout their interactions. Actions, from the action map design artifacts, can be mapped to achievement plans, which the agents use to achieve their interaction goals. Furthermore, failure handling mechanisms, such as action retries and rollbacks, were explained as well.

The implementation of parallelism in design artifacts was also explained. This included the implementation of the three types of parallelism: interaction goal parallelism, action parallelism and instance parallelism.

A prototype software support tool for implementing Hermes interactions was also discussed. This included the development of a syntactic representation of Hermes design artifacts, such
as interaction goal hierarchies and actions maps, and of the prototype tool. The prototype tool generated skeleton code for the Jadex agent platform given an interaction in such a representation. The code generation tool can be extended to produce skeleton code for other agent platforms.
Chapter 6

Evaluation

The previous chapters have presented Hermes: a methodology that is intended to be pragmatic, and was developed to increase flexibility and robustness in interactions. In this chapter, an experimental evaluation of Hermes is presented to assess how well Hermes succeeds in meeting its aims. The evaluation compared Hermes against the message-centric interaction design aspect of Prometheus. It is both qualitative and quantitative, and it covers design notation, design process and guidance given to the interaction designer.

This chapter firstly describes the evaluation objectives and the metrics used in the evaluation. The experimental procedure used to carry out the evaluation is then explained. Finally, an analysis of the results obtained is presented.

6.1 Evaluation Objectives and Metrics

The evaluation is a qualitative and quantitative comparison of goal-oriented versus message-centric interaction design, with Hermes and Prometheus being representatives of the respective approaches. The main objective of the evaluation is to determine the advantages and disadvantages of Hermes’ goal-oriented interactions over Prometheus’ message-centric interactions. As the key aim of Hermes is to allow for greater flexibility and robustness in interactions, the evaluation criteria included two metrics to measure flexibility and robustness of interactions.

The metrics chosen were selected to measure the key expected differences between Hermes and Prometheus. Obviously, the most notable difference between the two is that Hermes is goal-oriented whilst Prometheus is message-centric. This difference leads to a number of key differences between the two approaches.

One difference concerns the design notation, which leads to different types of “slicing”, i.e. how the interaction design is represented as a collection of design artifacts. In Prometheus, the interaction protocol shows the inter-agent processes, i.e. the interactions between the agents, and
the process diagrams – typically one per agent per interaction protocol – depict the intra-agent processes, i.e. the processes within the agents. Hermes, however, uses action maps which can be seen as the equivalent of Prometheus interaction protocols and process diagrams as they capture both inter- and intra-agent processes. That is, Hermes “slices” per interaction goal (i.e. one action map per atomic interaction goal) while Prometheus “slices” per agent (i.e. one process diagram per agent and interaction protocol).

Although slicing per agent is intuitive, it is disadvantageous in that a process diagram displays the internal processing of the entire interaction for a single agent. The processes on a single process diagram may appear to be disjoint, however, when the process diagrams for all interacting agents concerning a particular interaction protocol are placed together, the diagrams make perfect sense. It is then clear why and when a particular agent will execute a particular internal process as an effect of the other agents’ internal processes and message exchanges.

One advantage of this approach is that in large projects, detailed design (and implementation) can be easily divided. Agents can be developed separately based on the interface specified in the protocols.

In contrast to Prometheus, Hermes slices per interaction goal. In effect, Hermes divides the interaction into smaller, logical, coherent and modular parts. This approach captures both the inter- and intra-agent processes in one artifact and keeps them both in context. This appears to be a more natural way to slice the interaction and might be expected to provide less scope for missing steps in the processing (e.g. scenario steps and messages). However, the division of labour is not as clear as in Prometheus.

To assess the difference between how well these approaches reduce the scope for missing steps in the processing (e.g. scenario steps and messages), the scenario coverage metric is used. This metric simply counts how many of the scenario steps are included in the design.

Compared to Prometheus designers, Hermes designers attend directly to the agents’ actions rather than their message exchanges. Thus, in regard to designing alternatives, Prometheus designers are likely to think of different messages that agents can send in response to incoming messages from other agents whereas Hermes designers are more likely to think about alternative actions that agents can carry out in response.

It is expected that this will lead Hermes designs to have more alternatives than Prometheus. The flexibility metric is used to measure the number of ways in which an interaction can be completed. This involves counting the number of different paths through the interaction. Appendix I provides a detailed explanation of calculating the flexibility of an interaction.

Prometheus guides designers to consider alternatives, such as alternative messages that can be sent in response to an incoming message or alternative ways in which an interaction or part of it can be achieved. Hermes, however, also guides the designer to identify possible failure points and consider how they can be handled. As Prometheus deals with messages, i.e. the effects of the

---

1 There may also be process diagrams not related to interaction protocols but only to internal tasks.
Metric | Measure
---|---
Scenario Coverage | Number of scenario steps included in design.
Flexibility | Number of different paths through interaction.
Robustness | Number of scenario variations included in design.
Design Time | Time taken to create design (including refinements).

Table 6.1: Summary of Metrics

agents’ actions, only the effects of the failures will be apparent. These effects do not necessarily identify the failures themselves. In contrast, as earlier stated, Hermes deals directly with the agents’ actions, i.e. the cause of the failures. Thus, it is expected that this will more clearly identify the failures.

Therefore, it is expected that Hermes will better guide the designer to identify and deal with failures than Prometheus. The robustness metric is used to measure this difference. It is a measure of how well an interaction can handle foreseeable failures (measured as the number of scenario variations covered).

From our own experience using Hermes and Prometheus, we found that Hermes designs were more involved to create than Prometheus designs. Thus, the design time metric is used to determine how much time is required to create a design using a particular approach.

For a methodology to be pragmatic, it should be both effective and efficient. The effectiveness of the methodologies is measured by the scenario coverage, flexibility and robustness metrics, while the efficiency is measured by the design time metric. These metrics are summarized in Table 6.1 and further described in Section 6.3 as part of the results analysis.

6.2 Experimental Procedure

Participants with varying agent design experience were sought for the evaluation, in which they were to design interactions using a common scenario. Approximately half of the participants created designs using the Hermes approach, whilst the remainder created Prometheus designs. As the participants’ skill and experience varied, they were distributed into the two groups equally, in regard to both skill and experience. This was done by classifying the participants into three bands: novice, intermediate and expert. The classifications were based on the participants’ responses from a pre-evaluation questionnaire, which enquired about their knowledge, skill and experience with agent-oriented software engineering. The pre-evaluation questionnaire can be found in Appendix D.

For each band, participants were randomly allocated to groups if the number of participants in each group for that band were equal. If the number of participants in the groups for that band were uneven, the new participant was added to the group with the least participants to ensure that
each group had the same amount of participants. The final allocation of participants to groups is shown in Table 6.2.

There were two more participants in addition to those shown in Table 6.2, Participants 9 and 11. Participant 11 did not submit any evaluation material, was not classified into any band or group, and was not included in the evaluation. Participant 9 was classified as “intermediate” and allocated into the Hermes group. However, from the partial design submitted, it was clear that Participant 9 did not follow the Hermes process and the resulting design did not make sense as a Hermes design. This led to difficulties in applying some of our metrics to the design and inclusion into the evaluation would require that we impose our own interpretation of the design and alter it. Thus, Participant 9’s design was omitted from the evaluation. For more details on Participant 9’s design, refer to Appendix H.

<table>
<thead>
<tr>
<th></th>
<th>Hermes</th>
<th>Prometheus</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novice</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Participant:</td>
<td>8</td>
<td>Participants: 1, 12</td>
<td></td>
</tr>
<tr>
<td>Intermediate</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Participants:</td>
<td>3, 5, 10</td>
<td>Participants: 4, 6, 13, 14</td>
<td></td>
</tr>
<tr>
<td>Expert</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Participants:</td>
<td>2, 7</td>
<td>Participant: 15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>7</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 6.2: Experimental Groups and Bands

The evaluation scenario was designed with enough complexity to allow satisfactory evaluation of the methodologies, but simple enough so that the participants could complete the designs in 2–4 hours. To ascertain the correct level of complexity, a number of pilot tests were carried out on colleagues until the correct complexity was obtained.

The evaluation scenario concerned three agents in a meeting scheduling situation. The three agents played the roles of Organizer, Participant and Venue Manager. In the scenario, the Organizer is to schedule a meeting with the Participant at a mutually appropriate time and then to organize a venue for the meeting with the Venue Manager on the required date and time, and at an acceptable cost. The evaluation scenario given to the participants can be found in Appendix C.

As part of the evaluation, the participants were provided with the appropriate training manual depending on which group they were in. Tool support was also provided. Hermes participants were provided with UMLet with custom palettes provided for Hermes whereas Prometheus par-

---

2The participant numbers do not correspond to any particular order.
3The pilot test participants did not participate in the evaluation nor were their results included in the evaluation.
4http://www.qec.lfs.tuwien.ac.at/~auer/umlet
participants were given UMLet with custom palettes for Prometheus\(^5\) and Michael Winikoff’s AUML tool\(^6\) [Winikoff, 2005] was also provided\(^7\).

The participants were asked to create their designs using the provided software tools and to also time their designs. Once they completed their designs, the participants were required to fill in a post-evaluation questionnaire which enquired about their opinions in regard to design notation, design process and the resulting artifacts. As participants only used one methodology (either Prometheus or Hermes), they were not asked to compare the two methodologies in the questionnaire. The Prometheus and Hermes post-evaluation questionnaires are provided in Appendices E and F respectively. At the end of the evaluation, the participants’ designs, along with the time taken to create them, and the questionnaires were collected for analysis. The author analyzed the questionnaires and applied metrics to the participants’ designs.

### 6.3 Results

As previously mentioned, the evaluation carried out was both qualitative and quantitative. The qualitative aspect of the evaluation involved analyzing the responses to open-ended questions from the participants’ post-evaluation questionnaire. The quantitative part of the evaluation involved analyzing Likert scale responses to questions from the post-evaluation questionnaire and the application and comparison of the metrics mentioned in Section 6.1 to the participants’ submitted designs.

The evaluation required participants with varying degrees of experience with agent system design, who were difficult to obtain, and compared to medical or psychological studies, a much smaller number of participants were recruited. Although the number of participants was small, this is typical when evaluating methodologies, and it was sufficient to obtain statistical significance with an appropriate statistical test. In the following paragraph, we describe the chosen statistical test and argue its appropriateness for our small sample size.

The exact Wilcoxon rank sum test [Devore, 2004] was selected for all statistical significance testing in this evaluation as it is a 2-sample test which combines both samples together. More importantly, unlike other statistical tests, such as the 2-sample T-test and Z-test, the Wilcoxon test can be used for both small and large sample sizes. Furthermore, as the exact Wilcoxon rank sum test is a non-parametric test, it does not require normally distributed samples. Thus, the exact Wilcoxon rank sum test is appropriate for this evaluation, which has small sample sizes that are not normally distributed.

Although the post-evaluation questionnaire was informative, it is not used to evaluate the work. It is used to gain better insight on the participants’ view of their own design. Instead,

---

\(^5\)UMLet was used for Prometheus instead of the Prometheus Design Tool in order to reduce the difference between tools.

\(^6\)http://www.cs.rmit.edu.au/~winikoff/auml

\(^7\)The Prometheus Design Tool did not support AUML at the time the evaluation was carried out.
the conclusions are drawn based on the analysis of the designs themselves. An analysis of the post-evaluation questionnaire is presented next, followed by an analysis of the application of the metrics to participants’ designs.

6.3.1 Post-Evaluation Questionnaire Analysis

The post-evaluation questionnaire (refer to Appendices E and F) enquired about:

- the designs which the participants created
- the design process which the participants followed to create their designs

As for the designs which the participants created, the questionnaire enquired about characteristics such as flexibility, robustness, simplicity, understandability, ease of design and design speed (refer to Appendices E and F for the complete list). This was an attempt to determine the participants’ opinions about their own designs.

Participants were presented with a scale similar to that shown in Table 6.3 to enquire about their opinions on various aspects of the interactions they designed.

<table>
<thead>
<tr>
<th>Flexible</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Rigid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robust</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>Frail</td>
</tr>
</tbody>
</table>

*Table 6.3: Flexibility and Robustness Scales from Post-Evaluation Questionnaire*

The questionnaire results were tested for statistical significance, however, only responses to one of the questions resulted in statistical significance. This question enquired about the participants’ perceptions about how easy it is to follow the interaction they have created. We present the results for the responses to this question along with responses to three other questionnaire items, namely flexibility, robustness, and “overall ease of use” of the methodologies. Although statistical significance was not obtained for any of these three questionnaire items, we report and interpret the results as they form an important part of the evaluation. For a list of p-values for responses to all the questions, refer to Appendix G.

The participants’ responses in regard to how flexible and robust they believed their designed interactions to be were tested and did not result in statistical significance (p-values of 0.7488 and 0.9260 were obtained for flexibility and robustness respectively). Similarly, the responses to the question enquiring about the overall ease of use of the methodologies did not produce statistically significant results (a p-value of 0.1562 was obtained). There was no difference between the two groups regarding how flexible and robust they *perceived* their designs to be, the metrics used to measure flexibility and robustness did indicate that there was a significant difference (refer to Sections 6.3.3 and 6.3.4).

The only questionnaire item which was statistically significant (a p-value of 0.0087 was obtained) was an enquiry about how easy participants felt their designed interactions were to follow.
All participants, whether using Prometheus or Hermes, believed their designs were easy to follow (refer Figure 6.1). There is little variation in the participants’ responses as the standard deviation for Hermes participants is 0.516 whilst Prometheus participants had a slightly lower standard deviation of 0.488. Prometheus participants responded more favourably as their responses were either “2” or “3” while responses from Hermes participants ranged from “1” to “3”. This resulted in a higher mean and median for Prometheus (2.714 and 3 respectively) than for Hermes (1.667 and 2 respectively).

![Figure 6.1: Designed Interaction is Easy to Follow (negative values represent “Hard to Follow” whilst positive values represent “Easy to follow”)](image)

From the participants’ responses to queries about their designed interactions, there were no significant differences between the remainder of the characteristics enquired about. Histograms of each characteristic enquired about are available in Appendix K whilst raw data plots of each characteristic are available in Appendix J.

In regard to the design process, the characteristics enquired about were: how well guided the process was, ease of use, speed, simplicity, effort required, how easy it was to learn and expressiveness (refer to Appendices E and F for the complete list). Although none of the related questionnaire items resulted in statistical significance, both Hermes and Prometheus participants generally felt that their respective design processes were easy to use, fast, simple, easy to learn and expressive. They also felt that the model and diagrams used were intuitive. Refer to Appendix K for the histograms and Appendix J for the raw data plots of these characteristics.
The questionnaire also contained a number of open-ended questions. These questions enquired about how easy it is to understand the various design artifacts of the two methodologies. The Prometheus questionnaire also had a question enquiring if the process diagrams were useful in identifying missing variations of failures. In the case of Hermes, there was a question enquiring if the action maps were useful for the same purpose.

Most Hermes participants agreed that the interaction goal hierarchy and the action maps they had created were easily understood and made positive comments such as “It [the interaction goal hierarchy] gives a simple breakdown of the interaction goals” (Participant 3) and “… they [the action maps] allowed me to focus on the given IG, without thinking about the whole interaction as a whole” (Participant 2). Participant 5 expressed some difficulties with the complexity of the action maps:

“Adding loops and alternate steps due to variation or failure recovery/rollback makes the maps harder to read. Annotations may be required to ensure someone doesn’t get lost interpreting the action map.”

Prometheus participants agreed that, overall, their designed interactions were easily understood. Specifically, they all felt that their interaction diagrams and process diagrams were easily understood. However, a number of Prometheus participants believed that their interaction protocols were not easily understood. For example, Participant 13 stated that it was “not very easy [to understand the interaction protocol created], in fact, it is quite complex as several non-simple protocols are nested”. Participant 1 expressed difficulty in the creation of the protocol diagram: “I found it difficult to put in the loops and decision points that were required if the action was refused. I ended up with too many alternative boxes and ‘go to’s”, although, Participant 15 did point out that “… the protocol diagram is the most useful of the three classes of diagrams ...”.

Half of the Hermes participants believed that the action maps were useful to identify failures or variations in the interactions while two of the seven Prometheus participants thought the process diagrams were useful to identify failures and variations.

The post-evaluation questionnaire also allowed the participants to provide comments. In the comments, Participant 5 mentioned that the action maps were particularly useful in identifying endless loops in the interactions: “... it [the action map] did help catch endless loops”. This was a problem which was present in some of the Prometheus designs submitted. For example, Figure 6.2 shows part of Participant 4’s protocol diagram. From the protocol diagram, it can be seen that the alternative box caters for two situations: when a common availability between the Organizer and the Participant is found and when it is not found. If a common availability is found, the interaction progresses whereas if a common availability is not found, then the AUML goto specifies that the Organizer and Participant should try to find another available time. In the case of repeated failure, it is possible that the Organizer and Participant endlessly try to find a common time. A second exit condition needs to be placed in this loop. That is, the alternative box should cater for a third situation in which all the potential dates and times for the meeting
have been exhausted. This problem is also present in other parts of the interaction, such as trying to find an available room and a valid credit card.

![Protocol Diagram](image)

*Figure 6.2: Participant 4’s Protocol Diagram (Partial)*

Participant 15 expressed that splitting the interaction per agent, like in Prometheus, means that they become “devoid of context”:

“... in these [the process diagrams] it is less easy to see the overall sequence of operations ... splitting up the interactions like this [per agent] means they become devoid of context: you can’t see, from looking at the process diagram, that the reason a ‘cost of room’ message is received is that the agent sent a ‘check room cost’ message earlier on in the sequence, and hence is waiting to find out the answer.”

Participant 1 agreed with Participant 15:

“It’s difficult to show the whole process here [in the process diagram] ... [It] would have been good to have a diagram that showed similar things [to the internal agent processes], but with the other agents included. It was ambiguous who the message was going to ...”

Some of the difficulties mentioned by the Hermes participants included that the temporal dependency symbol is the reverse of the causality symbol, which causes confusion. Participant 3 stated: “Using the same (but reversed) symbol/icon for IGH dependency and AM causality/action flow is confusing”. Participant 8 also ascertained the difficulty:

“The IGH appears straight forward [sic], although their [sic] is a little confusion based on the direction of the arrows, as this appears counter-[intuitive] to what most people would perceive the arrows to mean. However, once someone has used the methodology, it makes sense.”
Another problem is that the data flow between action maps is sometimes not very clear, as Participant 2 stated: “... it was not clear how the required data is passed between the action maps”.

### 6.3.2 Scenario Coverage

The scenario coverage metric is a metric we have developed to determine how well a designed interaction covers the steps in the given scenario. It is used to indicate how well each methodology guides the designer to ensure that all scenario steps are accounted for and is measured simply as the number of scenario steps covered in the resulting interaction.

Table 6.4 shows the scenario coverage of the submitted designs. From the table, it can be seen that the scenario coverage for Prometheus designs varies from 10 to 14, however, the Hermes designs all received a perfect scenario coverage of 14 (as there were 14 steps in the given evaluation scenario). This indicates that the scenario coverage is better for Hermes than Prometheus. This is confirmed by a p-value of 0.04895 from an exact Wilcoxon rank sum test, which shows statistical significance in the greater number of scenario steps present in Hermes designs compared to Prometheus designs.

The most commonly missed scenario steps were:

- Step 4: Determine Room Size
- Step 6: Check Funds
- Step 12: Reserve Room
- Step 14: Set Meeting

These steps are the finer steps in the scenario and can be easily overlooked if the designer does not carefully go through the scenario step-by-step and check for consistency between the scenario and the design artifacts.

### 6.3.3 Flexibility

The robustness and flexibility metrics we developed are two of the most important metrics as they are both the main focus of the evaluation. Flexibility is defined as the total number of different possible message sequences which lead to the (successful or unsuccessful) completion of an interaction. It is measured by determining the total number of possible variations (or “paths”) in the designs.

The number of possible paths through the given scenario depends on three variables:

- \( m \), the number of alternative times and dates available for the meeting;
- \( r \), the number of alternative rooms available for the meeting to take place in; and
<table>
<thead>
<tr>
<th>Participant #</th>
<th>01</th>
<th>04</th>
<th>06</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>Σ</th>
<th>02</th>
<th>03</th>
<th>05</th>
<th>07</th>
<th>08</th>
<th>10</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step #</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>14</td>
<td>10</td>
<td>14</td>
<td>11</td>
<td>10</td>
<td>14</td>
<td>-</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>-</td>
</tr>
</tbody>
</table>

*Table 6.4: Scenario Coverage Results*


- \( c \), the number of alternative credit cards available to pay for the rooms.

As the number of paths depend on these variables, the flexibility of a submitted design is modelled as a function of \( m \), \( r \) and \( c \). Although the flexibility metric is defined simply, it can be quite complex to calculate the flexibility of an interaction. A detailed explanation of how to determine the flexibility of a given interaction is given in Appendix I. To compare the flexibility of the designs, a range of (reasonable) values were assigned to the three variables. When these values are substituted into the variables, the flexibility functions produce flexibility values which can then be compared. Table 6.5 shows flexibility values for some selected reasonable variable values. Figure 6.3 displays plots of the data from Table 6.5.

Figure 6.3 (note the logarithmic scale) and Table 6.5 show that, in general, the average flexibility for Hermes designs is always much greater than Prometheus designs. Participant 7’s design was the most flexible Hermes interaction, while Participant 13’s design was the most flexible Prometheus interaction. However, the flexibility produced by that particular Hermes design greatly exceeds the most flexible Prometheus design obtained.

For example, when \( m \), \( r \) and \( c \) all take the value of 3, Participant 7’s Hermes design is more than 150 times more flexible than Participant 13’s Prometheus design. When \( m \) and \( r \) are increased to 4 and \( c \) remains at 3, the Hermes design is 1,000 times more flexible than the Prometheus design.

<table>
<thead>
<tr>
<th>Participant #</th>
<th>( m = 3, r = 3, c = 3 )</th>
<th>( m = 4, r = 4, c = 3 )</th>
<th>( m = 7, r = 7, c = 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 (Prometheus)</td>
<td>240</td>
<td>620</td>
<td>8,160</td>
</tr>
<tr>
<td>04 (Prometheus)</td>
<td>60</td>
<td>155</td>
<td>2,040</td>
</tr>
<tr>
<td>06 (Prometheus)</td>
<td>4</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>12 (Prometheus)</td>
<td>12</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>13 (Prometheus)</td>
<td>2,655</td>
<td>13,598</td>
<td>1,613,326</td>
</tr>
<tr>
<td>14 (Prometheus)</td>
<td>16</td>
<td>25</td>
<td>64</td>
</tr>
<tr>
<td>15 (Prometheus)</td>
<td>16</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>Average</td>
<td>429</td>
<td>2,063</td>
<td>231,951</td>
</tr>
<tr>
<td>02 (Hermes)</td>
<td>488</td>
<td>1,256</td>
<td>16,448</td>
</tr>
<tr>
<td>03 (Hermes)</td>
<td>387</td>
<td>1,170</td>
<td>31,707</td>
</tr>
<tr>
<td>05 (Hermes)</td>
<td>164</td>
<td>255</td>
<td>648</td>
</tr>
<tr>
<td>07 (Hermes)</td>
<td>405,872</td>
<td>14,269,655</td>
<td>598,072,990,808</td>
</tr>
<tr>
<td>08 (Hermes)</td>
<td>1,140</td>
<td>5,445</td>
<td>236,160</td>
</tr>
<tr>
<td>10 (Hermes)</td>
<td>385</td>
<td>1,121</td>
<td>20,481</td>
</tr>
<tr>
<td>Average</td>
<td>68,123</td>
<td>2,379,817</td>
<td>99,666,717,570</td>
</tr>
<tr>
<td>P-Values</td>
<td>0.01632</td>
<td>0.01748</td>
<td>0.02564</td>
</tr>
</tbody>
</table>

**Table 6.5: Flexibility Results**
CHAPTER 6. EVALUATION

Figure 6.3: Graphs of Flexibility Values (note logarithmic vertical scale)
Further increasing the values of $m$ and $r$ to 7 while keeping $c$ at 3 results in the Hermes design being 365,000 times more flexible than the Prometheus design.

The least flexible Prometheus design was obtained from Participant 6, whilst the least flexible Hermes came from Participant 5. However, Participant 5’s design was at least 40 times more flexible than its Prometheus counterpart.

Table 6.5 also shows the p-values from exact Wilcoxon rank sum tests for the selected variable values. As can be seen, statistical significance, indicating that the greater flexibility in the Hermes designs compared to the Prometheus designs is not due to chance or randomness, was obtained for all shown values. In fact, statistical significance was obtained for all test values of $m$, $r$ and $c$, many of which are not presented in Table 6.5.

It is interesting to note that although responses from the participants do not indicate a difference in their perceptions of the flexibility of their design (refer to Section 6.3.1), there is a statistically significant difference when measured objectively. This would seem to indicate that a designer’s perception of flexibility of his or her own design is not a useful measure.

6.3.4 Robustness

The robustness metric is defined as how well the interaction is able to persevere through failure. It measures the designed interactions’ resilience to failure. In the evaluation scenario given to the participants, only one scenario variation was provided, however, the scenario was designed such that there were a total of nine variations which could be foreseen (with different degrees of difficulty in identifying them). Table 6.6 presents the possible variations and the participants’ success at identifying them.

Robustness is measured as the total number of scenario variations which were identified (not including the provided variation). Therefore, the robustness metric provides a comparison of not only how well the Hermes and Prometheus methodologies help the designers identify possible failures in the interaction, but also how well they help the designers to determine scenario variations (i.e. alternative paths to achieve the interaction).

The robustness results are shown as the total in Table 6.6. The robustness values for Hermes designs are generally higher than the Prometheus designs. In fact, the highest robustness value for Prometheus (3) is equal to the lowest robustness score for Hermes. Furthermore, two Prometheus designs only allowed for the provided scenario variation (and scored 0), whilst the worst Hermes design allowed for 3 variations in addition to the provided variation.

The highest robustness value of 7, belonged to a Hermes design. As previously mentioned, there were 9 possible variations (including the provided variation) for the given scenario, however, one of the unprovided variation, Step 4: Cannot determine room size, was particularly difficult to handle it. Hence, identification of variations and handling the variations are strongly correlated.
### Table 6.6: Possible Variations

<table>
<thead>
<tr>
<th></th>
<th>Prometheus</th>
<th></th>
<th></th>
<th></th>
<th>14</th>
<th>15</th>
<th>Σ</th>
<th>02</th>
<th>03</th>
<th>05</th>
<th>07</th>
<th>08</th>
<th>10</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 2:</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td>√</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cannot find suitable date/time</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 4:</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cannot determine room size</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 6:</td>
<td></td>
<td>√</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not enough funds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 7:</td>
<td></td>
<td>√</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Room is not available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 8:</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Room is no longer available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 10:</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Credit card list is empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 11:</td>
<td></td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Credit card is declined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 13:</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Participant no longer available</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Possible Variations
identify and none of the designers allowed for it. This may indicate that the particular variation was too difficult to find or too trivial to consider.

An exact Wilcoxon rank sum test was performed on the robustness values. The test yielded a p-value of 0.001748, which indicates that the greater robustness of the Hermes designs compared to the Prometheus designs is statistically significant.

Statistical significance was not obtained with respect to the participants’ perceptions of the robustness of their designs (refer to Section 6.3.1), however, the objective measures show that there is a significant difference. As with flexibility, this indicates that their perception of their own designs is not a useful measure.

### 6.3.5 Design Time

The design time metric indicates the time taken to create an interaction design for the given scenario. It is measured in minutes from the time the participants start creating the design until the design is completed (including refinements). The amount of time spent on each of Hermes’ sub-phases was also measured, but this information was not useful for comparison, it was collected to help improve Hermes.

The design times, refer to Table 6.7, show that the time taken to create a Hermes design is significantly higher than for a Prometheus design. The shortest time taken to create a Prometheus design was 45 minutes while the longest time was 240 minutes. The fastest Hermes design time was 145 minutes and the longest time was 320 minutes. An exact Wilcoxon rank sum test on the design times produced a p-value of 0.006993, affirming that the increase in the time required to create a Hermes design compared to creating a Prometheus design is statistically significant.

### 6.4 Summary

Overall, the results of the evaluation showed that the goal-oriented approach of Hermes resulted in interactions which were many times more flexible than the message-centric approach of Prometheus. The Hermes interactions were also significantly more robust and had better scenario coverage than Prometheus designs. However, the improved flexibility, robustness and scenario coverage come at a cost: more time is required to create a Hermes design than a Prometheus
Interestingly, designers did not perceive their designs as being different with respect to flexibility and robustness, indicating that designer perception is not a very useful measure for these characteristics. We also found that Prometheus participants were more likely to think that their designs were easy to follow than Hermes participants.

Our results indicate that Hermes is successful in leading to interactions that are flexible and robust. However, more time is required to create these interactions and the notation for the design is not as easy to follow as AUML.
Chapter 7

Conclusion and Future Work

In this thesis, an alternative to the traditional message-centric approach to agent interaction design has been investigated. A key aspect of this alternative approach is the use of a fundamental intelligent agent concept, goal-orientation, to develop agent interactions. As part of the work, a methodology for the design of goal-oriented agent interactions, called Hermes, was developed. The three main parts of the methodology, the design process, its integration with full agent system design methodologies, and the implementation of the design on goal-based agent platforms, are summarized next, followed by a summary of the empirical evaluation carried out.

A notation for modeling goal-oriented interactions was explicitly developed as part of the Hermes methodology. This notation allows interaction designers to represent agent interactions in design artifacts such as interaction goal hierarchies and action maps. Hermes also provides processes and heuristics to guide the designer in developing the agent interactions and creating the design artifacts. A prototype design software tool which supports the Hermes design process has also been developed.

As the Hermes methodology only caters for agent interactions, and not entire agent systems, guidelines to integrate Hermes with full agent system design methodologies have been provided. These guidelines have been used in the integration of Hermes into the Prometheus methodology. As additional information from Prometheus artifacts is available in the integrated methodology, the interaction designer can be better guided through the process of designing goal-oriented interactions. The designer is also able to get a holistic view of where the interactions belong and what their purposes are in the entire system as opposed to developing isolated interactions.

The Hermes methodology also provides guidelines for mapping its design artifacts, such as interaction goal hierarchies and action maps, into an implementation on goal-based agent platforms. This includes guidelines on how to represent interaction goals, and mapping rules between design artifacts and implementation artifacts such as coordination and achievement plans. Furthermore, it is possible to partially automate these guidelines and rules. This has been shown through the development of a prototype implementation software support tool. The tool takes a syntactic rep-
representation of Hermes design artifacts and produces skeleton code for the Jadex agent platform. The tool can be modified to produce code for other goal-based agent platforms.

An empirical evaluation which compared goal-oriented interaction design against message-centric interaction design was carried out. This was achieved by comparing the Hermes interaction design process with the Prometheus interaction design process. The comparison involved providing two groups of participants with a common scenario and asking participants from one group to create Prometheus interaction designs whilst asking the other group of participants to create Hermes interaction designs. The participants’ designs and questionnaires were then collected and analyzed.

The comparison was based on four metrics: scenario coverage, flexibility, robustness, and design time. The scenario coverage metric measures the number of scenario steps that the designers incorporated into their designs. It indicates how well the design methodologies guide the designers.

Flexibility measures the total number of possible paths through the designed interactions and measures the different ways in which the agents can achieve this particular interaction.

Robustness is measured as the number of scenario variations the designers identified in their designs and measures the designed interactions’ resilience to failure and how well the methodologies help the designer to identify possible failure points.

Design time is measured as the time taken to create a design from start to finish, including refinements to the designs.

The results of the evaluation indicated that scenario coverage was significantly better for Hermes than Prometheus. Hermes designs were also significantly more flexible and robust than their Prometheus counterparts. However, it took significantly longer to create the Hermes designs compared to Prometheus design creation.

The four metrics used were designed specifically for this evaluation and they measure what we believe to be important aspects of agent interaction design and methodologies. However, this work can be expanded to include metrics which measure other dimensions of agent interaction design. These may include a metric to measure the degree of inconsistencies in the designs created. This is motivated by the concept that a good methodology should assist in detecting and/or avoiding inconsistencies. Additionally, the four metrics used in this work focused solely on the design aspect of the methodology. Further metrics could be devised to measure aspects of implementation, and the mapping of design to implementation.

The only item from the post-evaluation questionnaire which is statistically significant is the participants’ perception of how easy they believed their designs were to follow. Both Hermes and Prometheus participants felt their designs were easy to follow, however, Prometheus participants responded more favourably than Hermes participants and there is a slightly lower standard deviation, and a higher mean and median for Prometheus. Although the participants’ responses to the remainder of the post-evaluation questionnaire items are not statistically significant, they suggested that Hermes participants felt positively about the flexibility and robustness of their de-
signed interactions while Prometheus participants felt less strongly about these characteristics of their designed interactions. Furthermore, Hermes participants felt that the Hermes design process is well guided. There were no clear differences about other aspects of the two design processes. The participants felt that both the Prometheus and Hermes design processes are easy to use, fast, simple, easy to learn and expressive. They also felt that the model and diagrams used were intuitive.

These results indicate that Hermes is most suitable for interactions that are complex, dynamic and failure-prone. However, in simpler and less failure-prone interactions, message-centric interaction design would be more appropriate.

### 7.1 Research Aims and Contributions Revisited

As stated in Section 1.2 (page 7), the main aim of the work is the development of a methodology for designing agent interactions that is congruent with the intelligent agent paradigm and which should be pragmatic for practicing software engineers. The specific aims of the thesis were to:

- develop a more pragmatic approach to specifying and designing flexible and robust agent interactions; and
- develop guidelines to implement designs developed using the new design methodology on any goal-based agent platform.

The empirical evaluation carried out provides evidence that Hermes meets the first aim as the goal-oriented designs were significantly more flexible and robust than message-centric designs. These results are primarily due to the combination of Hermes’ goal-oriented approach, its design processes, and heuristics. Furthermore, the evaluation also showed that Hermes is pragmatic as the methodology was successfully used by the evaluation participants, amongst whom were a number of novices and undergraduate students with little experience in designing agent interactions.

To address the second aim, the Hermes methodology provides guidelines and rules which map the design to an implementation on any goal-based agent platform. These guidelines and rules include how to represent interaction goals, and how to map interaction goals and actions to coordination and achievement plans respectively. The mapping process explains how the design artifacts can be mapped to plans that can be implemented on any goal-based agent platform, i.e., no platform specific features are used, thus, any goal-based agent platform can be used for the implementation.

### 7.2 Limitations and Future Work

The previous section has highlighted the strengths of Hermes. Another strength of Hermes is that it is pragmatic: it has been integrated with existing full agent system design methodologies and has software support tools for the design and implementation processes. However, in order
to be pragmatic for practicing software engineers, Hermes is not formal. Specialized techniques such as modal and temporal logics, which are used in other alternative approaches to message-centric interactions, were avoided. An effect of this is that Hermes interactions, unlike some other approaches such as social commitments, are not verifiable.

Although message sequences are emergent in Hermes, the interacting agents are limited to static interaction goal hierarchies as defined by the designers. The agents are not able to dynamically create interaction goal hierarchies for interactions. However, by using failure handling mechanisms such as rollbacks, a great degree of flexibility can be added to interactions. It is not clear that greater flexibility is required or practically advantageous.

The parallel support in Hermes is not as developed as its sequential counterpart. Although explicitly specified parallelism is not often used in agent interactions, it does play an important part in certain specific interactions. Hermes parallelism will need to be developed beyond what is presented in this thesis to make it more practical. This includes improving failure handling mechanisms, such as rollbacks, in interaction parallelism.

The prototype software support tools, although useful, will need to be further developed. This involves enhancing the tools to support the design and implementation processes. The tools also need to be improved to support interaction parallelism, generation of interface plans, and different causality types. Furthermore, constraint checking can also be implemented to ensure that interaction designers are creating valid Hermes designs.

To increase usability, the two support tools should be consolidated into a single tool. The consolidated tool can be further integrated into software support tools for methodologies that Hermes is integrated with. For example, as Hermes has been integrated with Prometheus, integrating the consolidated Hermes support tool with the Prometheus Design Tool (PDT) would ease the design process for interaction designers designing agent systems with the Prometheus methodology.

It may be worthwhile to investigate using Hermes for specialized types of agent interactions, such as teamwork. As goal-oriented interactions are used to coordinate agents through interactions, they may be useful in situations such as teamwork where coordination also plays a crucial part.

Although Hermes is designed to produce more flexible interactions with more execution paths than message-centric interactions, large numbers of executions paths are not always necessarily beneficial. In fact, if the produced interactions are too flexible, it may be difficult to verify the correctness of the design, and there may be difficulties in determining whether the interaction is well designed.

As the work in the thesis is targeted at the implementation level, and not the business level, there is no discussion of business models and processes as part of the methodology. This is important for the adoption of Hermes for real-world applications and is left as future work.

Additionally, the work in this thesis is set in the agents context, and thus focusses on agent-oriented methodologies. However, there are a number of methodologies developed by the non-agent community that may be worthwhile investigating to further improve Hermes. These include
the Rational Unified Process (RUP) [Jacobson et al., 1999], the V-Model [IABG Information Technology, 2008], and eXtreme Programming (XP) [Beck, 1999].

The RUP has been used in a number of AOSE methodologies, such as RAP/AOR, ADELFE, and MESSAGE [Wagner and Taveter, 2004], and it will be interesting to determine whether integration with Hermes would be worthwhile. In comparison with the V-Model, there is little testing/validation in the Hermes methodology as this has not been the focus of the work. This is a prominent aspect of the V-Model and is lacking in Hermes. The area of testing agent systems has received some interest of late and, if added to Hermes, will make it a more “solid” methodology. Some interesting issues would be how to carry out the various tests (e.g., “unit testing”, “integration test”, etc) on the design and implementation of Hermes interactions. It may also be useful to investigate the use of automated testing, which has been added to some AOSE methodologies, such as Prometheus [Zhang et al., 2007] and also integrated with software support tools [Zhang et al., 2008]. Some of the ideas of XP are orthogonal to Hermes, and could be adopted as potential improvements. For instance, pair-programming (or pair-designing) could be easily adopted in Hermes. The use of action message sequence diagrams can be seen as a design-level equivalent to test-driven development, and could be further developed, and used in a more XP-like manner. On the other hand, XP tends to focus on code and eschew the use of design artifacts, which is somewhat at odds with Hermes (or any design methodology).

Despite these limitations, Hermes has met its aims: the results of the empirical evaluation carried out are very positive and show that Hermes is a viable and pragmatic approach for the development of flexible and robust agent interactions.
Appendix A

Automatically Generated Plans

```java
package edu.rmit.cs.e-commerce.plans.coordination;
import jadex.model.MCondition;
import jadex.runtime.RBeliefbase;
import jadex.runtime.RGoal;
import jadex.runtime.ThreadedPlan;

public class Trade extends ThreadedPlan {

    /* 
     * Plan body 
     */
    public void body() {
        RBeliefbase beliefBase = getBeliefbase();
        // Achieve Interaction Goals in sequence.
        // Start Agree
        beliefBase.setFact("inAgree", new Boolean(true));
        waitFor(createCondition("beliefbase.inAgree == false && beliefbase.finishedAgree == true"));
        if (((Boolean)beliefBase.getFact("agreeSuccess")).booleanValue()) {
            beliefBase.setFact("inExchange", new Boolean(true));
            waitFor(createCondition("beliefbase.inExchange == false && beliefbase.finishedExchange == true"));
            if (((Boolean)beliefBase.getFact("exchangeSuccess")).booleanValue()) {
                beliefBase.setFact("tradeSuccess", new Boolean(true));
            }
        }
        // Synchronize with other Coordination plans
        beliefBase.setFact("inTrade", new Boolean(false));
        beliefBase.setFact("finishedTrade", new Boolean(true));
    }
}
```

Listing A.1: Generated Compound Coordination Plan
public class NegotiatePrice extends ThreadedPlan {
    /* Plan body */
    public void body() {
        RBeliefbase beliefBase = getBeliefbase();
        // Do not re-achieve IG if outcome is already successful
        if (!((Boolean)beliefBase.getFact("negotiatepriceSuccess"))
            .booleanValue()) {
            // Proactively start action if agent is initiator of this interaction
            if (((String)beliefBase.getFact("negotiatepriceInitiator"))
                .equals(((String)beliefBase.getFact("role")))) {
                // TODO:
                // Trigger the initial action, i.e. the relevant Achievement Plan.
                // If you use a goal to trigger the Achievement plan (as shown below),
                // remember to add its definition to the Agent Definition File.
                // Example:
                // =========
                // Trigger requestProvider Achievement plan
                RGoal goal = createGoal("getUpdate");
                dispatchTopLevelGoal(goal);
                /*
                */
            }
            // Synchronize with Achievement plans
            beliefBase.setFact("inNegotiatePrice", new Boolean(true));
            waitFor(createCondition("beliefbasefinishedNegotiatePrice",
                MCondition.IS_TRUE));
            beliefBase.setFact("inNegotiatePrice", new Boolean(false));
        }
    }
}

Listing A.2: Generated Atomic Coordination Plan

package edu.rmit.cs.e-commerce.plans.achievement.negotiateprice;
import jade.core.AID;
import jade.lang.acl.ACLMessage;
import jade.GoalEventFilter; import jade.model.MCondition;
import jade.runtime.RAbstractGoal; import jade.runtime.RBeliefbase;
import jade.runtime.RGoalEvent; import jade.runtime.ThreadedPlan;

public class ProposePrice extends ThreadedPlan {
    public static GoalEventFilter getGoalEventFilter() {
        // TODO: Return the goal which this plan will handle.
        // For example, if this plan was to handle the "match" goal:
        // return new GoalEventFilter("match");
        }

    public void body() {
        // Synchronize with Coordination plan
        waitFor(createCondition("beliefbase.inNegotiatePrice",
            MCondition.IS_TRUE));
        RBeliefbase beliefBase = getBeliefbase();
        RAbstractGoal goal = ((RGoalEvent) getInitialEvent()).getGoal();
        String provider = (String) goal.getParameter("provider");
        // TODO:
        // Messages might need to be sent to agents in the interaction.
        // If so, message templates will appear below. You will then need to
        // fill in the following messages and ensure they are sent. If no message
        // templates appear, then goal will need to be created and dispatched.
        // Follow the instructions below.
        String msgContents;
        String receiver;
// Send message to Merchant to trigger ConsiderPrice Achievement plan.
msgContents = ""; // TODO: Fill in message contents.
receiver = "Merchant"

// TODO: Replace ACLMessage.REQUEST with appropriate message type.
ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
msg.addReceiver(new AID(receiver, AID.ISLOCALNAME));
msg.setContent(msgContents);
sendMessage(createMessageEvent(msg));

Listing A.3: Generated Achievement Plan
Appendix B

Syntactic Representation Artifacts

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- Define an "ig" element type -->
<xs:complexType name="ig">
  <xs:sequence>
    <xs:element name="Initiator" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="InteractionGoals" minOccurs="0" maxOccurs="1">
      <xs:complexType>
        <xs:choice>
          <xs:element name="Sequential">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="InteractionGoal" type="ig" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="Parallel">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="InteractionGoal" type="ig" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="Any">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="InteractionGoal" type="ig" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:element name="Interaction">
  <xs:complexType>
    <xs:sequence>
      <!-- Interaction Goals must have a name -->
      <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```
<!-- AGENTS element -->
<xs:element name="Agents">
  <xs:complexType>
    <xs:sequence>
      <!-- A minimum of 2 agents are required for an interaction to take place -->
      <xs:element name="Agent" minOccurs="2" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Role" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- INTERACTION GOAL HIERARCHY element -->
<xs:element name="InteractionGoalHierarchy">
  <xs:complexType>
    <xs:sequence>
      <!-- A minimum of 1 interaction goal is required for an interaction -->
      <xs:element name="InteractionGoal" type="ig" minOccurs="1" maxOccurs="unbounded">
        <xs:element name="Roles">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Role" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- ACTION MAPS element -->
<xs:element name="ActionMaps">
  <xs:complexType>
    <xs:sequence>
      <!-- A minimum of 1 action map is required for an interaction -->
      <xs:element name="ActionMap" minOccurs="1" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <!-- Must have exactly 1 AchievesIG element -->
            <xs:element name="AchievesIG" type="xs:string" minOccurs="1" maxOccurs="1"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- ACTIONS element -->
<xs:element name="Actions">
  <xs:complexType>
    <xs:sequence>
      <!-- A minimum of 1 action is required for an interaction -->
      <xs:element name="Action" minOccurs="1" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <!-- Type can only be one of the enumerated values below -->
            <xs:element name="Type" minOccurs="1" maxOccurs="1" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:enumeration value="Final Caused"/>
<xs:enumeration value="Final Independent"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="Owner" type="xs:string"
minOccurs="1" maxOccurs="1"/>
<xs:element name="LoadCodeFromFile" type="xs:string"
minOccurs="0" maxOccurs="1"/>
</xs:sequence>
<!-- Action must be given a name -->
<xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
<!-- CAUSALITIES element -->
<xs:element name="Causalities"
minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element name="Causality" minOccurs="0" maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="Type" minOccurs="1" maxOccurs="1">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="Single"/>
<xs:enumeration value="ParallelSplit"/>
<xs:enumeration value="ParallelMerge"/>
<xs:enumeration value="BroadcastSplit"/>
<xs:enumeration value="BroadcastMerge"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element name="Origin" type="xs:string" />
<xs:element name="Destination" type="xs:string" />
<xs:element name="Label" type="xs:string"
minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<!-- Data Stores element -->
<xs:element name="DataStores"
minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence>
<xs:element name="DataStore"
APPENDIX B. SYNTACTIC REPRESENTATION ARTIFACTS

Listing B.1: XML Schema for Syntactic Representation

```xml
<Interaction
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation='../SyntacticRepresentation/hermes.xsd'
name='E-Commerce'>
  <!-- Roles -->
  <Agents>
    <Agent name='anna'><Role>Customer</Role></Agent>
  </Agents>
</Interaction>
```

Listing B.2: XML Syntactic Representation, Part 1: Interaction Goal Hierarchy

```xml
<Agent name="Bob"><Role>Merchant</Role></Agent>

<!-- Interaction Goal Hierarchy -->
<InteractionGoalHierarchy>
  <InteractionGoal name="Trade"><Initiator>initiator</Initiator>
    <InteractionGoals>
      <Sequential>
        <InteractionGoal name="Agree"><Initiator>initiator</Initiator>
          <InteractionGoals>
            <Sequential>
              <InteractionGoal name="DetermineAvailability">
                <Initiator>initiator</Initiator>
              </InteractionGoal>
              <InteractionGoal name="NegotiateDetails">
                <Initiator>initiator</Initiator>
              </InteractionGoal>
              <InteractionGoal name="NegotiatePrice">
                <Initiator>initiator</Initiator>
              </InteractionGoal>
            </Sequential>
          </InteractionGoals>
          <InteractionGoal name="Exchange"><Initiator>initiator</Initiator>
            <InteractionGoals>
              <Sequential>
                <InteractionGoal name="TransferGoods">
                  <Initiator>Merchant</Initiator>
                </InteractionGoal>
                <InteractionGoal name="Payment">
                  <Initiator>Customer</Initiator>
                </InteractionGoal>
                <InteractionGoal name="SendReceipt">
                  <Initiator>Merchant</Initiator>
                </InteractionGoal>
              </Sequential>
            </InteractionGoals>
          </InteractionGoal>
        </Sequential>
      </InteractionGoals>
    </InteractionGoal>
  </InteractionGoal>
</InteractionGoalHierarchy>

<!-- Action Maps -->
<ActionMaps>
  <ActionMap name="NegotiatePrice">
    <AchievesIG>NegotiatePrice</AchievesIG>
    <Roles>
      <Role>Customer</Role>
      <Role>Merchant</Role>
    </Roles>
    <Actions>
      <Action name="ProposePrice">
        <Type>Independent</Type>
        <Owner>Customer</Owner>
      </Action>
      <Action name="Re-proposePrice">
        <Type>Caused</Type>
        <Owner>Customer</Owner>
      </Action>
      <Action name="RollbackToNegotiateDetails">
        <Type>Final Caused</Type>
        <Owner>Customer</Owner>
      </Action>
      <Action name="CompleteInteractionGoal">
        <Type>Final Caused</Type>
      </Action>
    </Actions>
  </ActionMap>
</ActionMaps>
```
```xml
<Owner>Customer</Owner>
<LoadCodeFromFile>CompleteInteractionGoal.xml</LoadCodeFromFile>
</Action>
...
</Actions>
<Causalities>
<Causality>
<Type>Single</Type>
<Origin>ConsiderPrice</Origin>
<Destination>CompleteInteractionGoal</Destination>
<label>Price Accepted</label>
</Causality>
</Causalities>
</ActionMap>
</ActionMaps>
```

Listing B.3: XML Syntactic Representation, Part 2: Action Maps
Appendix C

Evaluation Scenario

C.1 Background

C.1.1 System Description

In an agent-based calendar system, agents autonomously manage their users’ calendar. The agents are able to schedule, reschedule and cancel their users’ meetings based on the users’ availability and preferences.

C.1.2 Interaction Description

In the particular scenario to be developed we are interested in creating an interaction between two users’ agents to schedule a meeting and book a venue for the meeting. The agents’ ability to reschedule and cancel meetings is not considered. That is, two agents will determine when to schedule a meeting for their two users without rescheduling or canceling currently existing meetings, and the organizing agent will book a venue for the meeting.

A user will instruct his agent to schedule a meeting and supply the agent with the following information:

- A list of potential dates and times (ordered by preference) during which the meeting can occur
- The participant who is to attend the meeting

The agents will then schedule the meeting at the most appropriate time for the users and the organizing agent will then book a venue for the meeting.

C.2 Agent Type and Roles

There are two agent types involved in the interaction, Meeting Manager and Venue Manager. The Meeting Manager is responsible for managing its user’s meetings. In the case of our scenario, this
involves requesting and scheduling meetings at appropriate times (i.e. during its user’s preferred availability times) and booking venues for meetings. Note that the scenario given in the next section does not show the functionalities within the agents as in our training manual. Instead, we have given the agent roles to facilitate the creation of the interaction design.

The Meeting Manager can take on two roles: Organizer and Participant. When in the Organizer role, it attempts to schedule a desired meeting for its user and book a venue for it. When in the Participant role, it attempts to select the best time to schedule a meeting (requested by another Organizer) for its user.

The Meeting Manager contains the following data:

- **Preferred Meeting Dates and Times** – A list of potential dates and times (ordered by preference) during which the user wishes the meeting to take place (this is only applicable when the Meeting Manager takes on the role of Organizer).

- **Timetable Data** – Represents the user’s timetable. A working day starts at 08:30 and concludes at 17:30, with one hour timeslots for meetings.

- **Funds** – The amount of money which the user is willing to spend to book meeting venues.

- **Credit Card List** – A list of credit card details for a number of different cards that the user has. These credit cards are to be used to book venues for meetings.

- **Room Sizes** – A list of different room sizes (small, medium and large).

The Venue Manager is responsible for managing venues. This involves answering queries about room costs and availability. The Venue Manager contains the following data:

- **Room Costs** – The costs of rooms of different sizes (small, medium and large).

- **Room Availability** – The availability of all the rooms at the venue.

### C.2.1 Notes

For simplicity, the following are assumed:

- Durations of meetings and room bookings are always one hour long. Availability times in the timetables are set in one hour slots.

- There is no failure when reading data from a database.

- There is no failure in the network (e.g. messages disappear, etc.)
**APPENDIX C. EVALUATION SCENARIO**

**Name:** Schedule Meeting  
**Description:** A meeting is scheduled between two Meeting Managers and a venue is booked by the organizing Meeting Manager and the Venue Manager.  
**Trigger:** Percept: Schedule Meeting Request

### Steps:

<table>
<thead>
<tr>
<th>Step</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Description</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Percept</td>
<td>Schedule Meeting Request</td>
<td>Organizer</td>
<td>User requests a meeting to be organized with a given participant and specifies a list of preferred meeting dates and times during which the meeting can occur.</td>
<td>Produces: Preferred Meeting Dates and Times, and Meeting Participant</td>
</tr>
<tr>
<td>2</td>
<td>Goal</td>
<td>Select Meeting Date and Time</td>
<td>Organizer</td>
<td>Organizer selects a date and time for the meeting.</td>
<td>Uses: Preferred Meeting Dates and Times, and Timetable</td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Determine Participant Availability</td>
<td>Participant</td>
<td>Participant determines its availability.</td>
<td>Uses: Timetable</td>
</tr>
<tr>
<td>4</td>
<td>Goal</td>
<td>Determine Room Size</td>
<td>Organizer</td>
<td>Organizer determines the size of the room required for the meeting for the number of participants.</td>
<td>Uses: Room Sizes</td>
</tr>
<tr>
<td>5</td>
<td>Goal</td>
<td>Check Room Cost</td>
<td>Venue Manager</td>
<td>Venue Manager checks the cost of the given room size for the date and time provided.</td>
<td>Uses: Room Costs</td>
</tr>
<tr>
<td>6</td>
<td>Goal</td>
<td>Check Funds</td>
<td>Organizer</td>
<td>Organizer determines if it has enough funds to pay for room cost.</td>
<td>Uses: Funds</td>
</tr>
<tr>
<td>7</td>
<td>Goal</td>
<td>Check Room Availability</td>
<td>Venue Manager</td>
<td>Venue Manager checks if a room of specific size is available for a given date and time.</td>
<td>Uses: Room Availability</td>
</tr>
</tbody>
</table>

*Figure C.1: Schedule Meeting Scenario (Steps 1–7)*
### APPENDIX C. EVALUATION SCENARIO

<table>
<thead>
<tr>
<th>Step</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Description</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Goal</td>
<td>Book Room</td>
<td>Organizer</td>
<td>Organizer requests a room of a specific size to be booked for a specific date and time.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Goal</td>
<td>Request Credit Card Details</td>
<td>Venue Manager</td>
<td>Venue Manager requests credit card details for payment.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Goal</td>
<td>Select Credit Card</td>
<td>Organizer</td>
<td>Organizer selects a credit card for payment.</td>
<td>Uses: Credit Card List</td>
</tr>
<tr>
<td>11</td>
<td>Goal</td>
<td>Process Payment</td>
<td>Venue Manager</td>
<td>Venue Manager processes Organizer’s payment.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Goal</td>
<td>Reserve Room</td>
<td>Venue Manager</td>
<td>Venue Manager reserves room for the meeting.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Goal</td>
<td>Confirm Meeting</td>
<td>Organizer</td>
<td>Confirms with the Participant the meeting details (sets meeting in timetable and sends room details).</td>
<td>Produces: Timetable</td>
</tr>
<tr>
<td>14</td>
<td>Goal</td>
<td>Set Meeting</td>
<td>Participant</td>
<td>Participant sets meeting in timetable</td>
<td>Produces: Timetable</td>
</tr>
</tbody>
</table>

**Variation:** Step 3: Participant is not available. Repeat steps 2-3.

*Figure C.2: Schedule Meeting Scenario (Steps 8–14)*
Appendix D

Pre-Evaluation Questionnaire

1. Which of the following best describes you currently?

- [ ] Undergraduate Student
- [ ] Postgraduate Student (Masters by course work)
- [ ] Postgraduate Student (Masters by research)
- [ ] Postgraduate Student (Ph. D.)
- [ ] Staff (Academia)
- [ ] Software Developer (Industry)
- [ ] Other, please specify

2. Rate your software system design ability (not including agent-oriented):

- [ ] Very Bad
- [ ] Bad
- [ ] Average
- [ ] Good
- [ ] Very Good

3. Approximately how long have you been designing software systems (not including agent-oriented)?

Years: ______________ Months: ______________
APPENDIX D. PRE-EVALUATION QUESTIONNAIRE

4. How many software systems have you designed (not including agent-oriented)?

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1–5</td>
<td>6–10</td>
<td>11–20</td>
<td>21–29</td>
<td>30 +</td>
</tr>
</tbody>
</table>

Briefly describe the software systems you have designed (e.g. small university assignment, industry project, research project, etc.).


5. Rate your familiarity with intelligent agents:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Unfamiliar</td>
<td>Unfamiliar</td>
<td>Average</td>
<td>Familiar</td>
<td>Very Familiar</td>
<td></td>
</tr>
</tbody>
</table>

6. Rate your familiarity with the following agent concepts:

(a) BDI

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Unfamiliar</td>
<td>Unfamiliar</td>
<td>Average</td>
<td>Familiar</td>
<td>Very Familiar</td>
<td></td>
</tr>
</tbody>
</table>

(b) Goals and Plans

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Unfamiliar</td>
<td>Unfamiliar</td>
<td>Average</td>
<td>Familiar</td>
<td>Very Familiar</td>
<td></td>
</tr>
</tbody>
</table>

7. Rate your agent-oriented system design ability:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Bad</td>
<td>Bad</td>
<td>Average</td>
<td>Good</td>
<td>Very Good</td>
<td></td>
</tr>
</tbody>
</table>

8. Approximately how long have you been designing agent-oriented systems?

Years: ______________ Months: ______________
9. How many agent-oriented systems have you designed?

[ ] 1–5  [ ] 6–10  [ ] 11–20  [ ] 21–29  [ ] 30 +

Briefly describe the agent-oriented systems you have designed (e.g. small university assignment, industry project, research project, etc.).

10. Rate your familiarity with Agent UML (AUML):

[ ] Very Unfamiliar  [ ] Unfamiliar  [ ] Average  [ ] Familiar  [ ] Very Familiar

11. Approximately how long have you been using AUML?

Years:  ______________  Months:  ______________

12. How many projects have you used AUML with?

[ ] 1–5  [ ] 6–10  [ ] 11–20  [ ] 21–29  [ ] 30 +

Briefly describe the projects with which you have used AUML (e.g. small university assignment, industry project, research project, etc.).
13. Rate your familiarity with Prometheus:

[ ] Very Unfamiliar [ ] Unfamiliar [ ] Average [ ] Familiar [ ] Very Familiar

14. Approximately how long have you been using Prometheus?
   Years: _______________ Months: _______________

15. How many systems have you designed with Prometheus?

[ ] 1–5 [ ] 6–10 [ ] 11–20 [ ] 21–29 [ ] 30 +

Briefly describe the systems you have designed with Prometheus (e.g. small university assignment, industry project, research project, etc.).


16. Rate yourself as an agent system designer:

[ ] Novice [ ] Intermediate [ ] Expert
17. Please provide any additional information that you feel will give us a better understanding of your background as a system designer. We are particularly interested in your experience as a system designer, the projects (type and size) you have designed and the methodologies that you have used.
Appendix E

Prometheus Post-Evaluation Questionnaire

1. Overall, I found the methodology easy to use.

[ ] [ ] [ ] [ ] [ ]
Strongly Disagree Disagree Neither Agree nor Disagree Agree Strongly Agree

2. Using the following rating sheet, please select the number that most closely matches your feeling about the interaction which you have created.

Flexible\(^1\) [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Rigid
Robust\(^2\) [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Frail
Simple [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Complex
Easy to understand [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Hard to understand
Easy to design [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Hard to design
Fast to design [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Lengthy to design
Easy to follow [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Difficult to follow

193
3. Using the following rating sheet, please select the number closest to the term that most closely matches your feeling about the design process of the methodology you have used.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Well guided</td>
</tr>
<tr>
<td>2</td>
<td>Easy to use</td>
</tr>
<tr>
<td>1</td>
<td>Fast</td>
</tr>
<tr>
<td>0</td>
<td>Simple</td>
</tr>
<tr>
<td>1</td>
<td>Does not require a lot of effort</td>
</tr>
<tr>
<td>2</td>
<td>Easy to learn</td>
</tr>
<tr>
<td>3</td>
<td>Conducive to thought process</td>
</tr>
<tr>
<td>2</td>
<td>Expressive</td>
</tr>
<tr>
<td>1</td>
<td>Model/diagram intuitive</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. How easy to understand do you think your interaction diagram is?

5. How easy to understand do you think your protocol diagram is?
6. How easy to understand do you think your process diagrams are?

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

7. Were your process diagrams useful in identifying missing variations or failures?

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

8. Did you have to modify your protocol diagram after creating your process diagrams? If so, what changes did you have to make?

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________
9. Please add any comments in the space provided that you feel will help us to evaluate/improve the methodology. We would especially appreciate your input on the following:

- Comments in regard to the design process.
- Comments in regard to the produced interaction.
Appendix F

Hermes Post-Evaluation Questionnaire

1. Overall, I found the methodology easy to use.

[ ] [ ] [ ] [ ] [ ]
Strongly Disagree Disagree Neither Agree nor Disagree Agree Strongly Agree

2. Using the following rating sheet, please select the number that most closely matches your feeling about the interaction which you have created.

Flexible\(^1\) [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Rigid
Robust\(^2\) [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Frail
Simple [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Complex
Easy to understand [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Hard to understand
Easy to design [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Hard to design
Fast to understand [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Lengthy to understand
Easy to follow [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Difficult to follow
3. Using the following rating sheet, please select the number closest to the term that most closely matches your feeling about the design process of the methodology you have used.

- **Well guided**: [3] [2] [1] [0] [1] [2] [3] Not well guided
- **Easy to use**: [3] [2] [1] [0] [1] [2] [3] Hard to use
- **Fast**: [3] [2] [1] [0] [1] [2] [3] Lengthy
- **Simple**: [3] [2] [1] [0] [1] [2] [3] Complex
- **Does not require a lot of effort**: [3] [2] [1] [0] [1] [2] [3] Requires a lot of effort
- **Easy to learn**: [3] [2] [1] [0] [1] [2] [3] Hard to learn
- **Conducive to thought process**: [3] [2] [1] [0] [1] [2] [3] Not conducive to thought process
- **Expressive**: [3] [2] [1] [0] [1] [2] [3] Inexpressive
- **Model/diagram intuitive**: [3] [2] [1] [0] [1] [2] [3] Model/diagram unintuitive

4. How easy to understand do you think your interaction goal hierarchy is?


5. How easy to understand do you think your action maps are?


6. Were your action maps useful in identifying missing variations or failures?

7. Did you have to modify your interaction goal hierarchy after creating your action maps? If so, what changes did you have to make?
8. Please add any comments in the space provided that you feel will help us to evaluate/improve the methodology. We would especially appreciate your input on the following:

- Comments in regard to the design process.
- Comments in regard to the produced interaction.
Appendix G

P-Values for Questionnaire Responses

<table>
<thead>
<tr>
<th>Questions/Criteria</th>
<th>P-Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall ease of use</td>
<td>0.1562</td>
</tr>
</tbody>
</table>

*Table G.1: P-Values for Questionnaire Responses about Participants’ Perceptions of the “overall ease of use” of the Methodologies Employed*

<table>
<thead>
<tr>
<th>Questions/Criteria</th>
<th>P-Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible</td>
<td>0.7488</td>
</tr>
<tr>
<td>Robust</td>
<td>0.9260</td>
</tr>
<tr>
<td>Simple</td>
<td>0.3228</td>
</tr>
<tr>
<td>Easy to understand</td>
<td>0.5874</td>
</tr>
<tr>
<td>Easy to design</td>
<td>0.2896</td>
</tr>
<tr>
<td>Fast to design</td>
<td>0.2022</td>
</tr>
<tr>
<td>Easy to follow</td>
<td>0.0087</td>
</tr>
</tbody>
</table>

*Table G.2: P-Values for Questionnaire Responses about Participants’ Perceptions of their Created Design*
## APPENDIX G. P-VALUES FOR QUESTIONNAIRE RESPONSES

<table>
<thead>
<tr>
<th>Questions/Criteria</th>
<th>P-Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well Guided</td>
<td>0.8164</td>
</tr>
<tr>
<td>Easy to use</td>
<td>0.5629</td>
</tr>
<tr>
<td>Fast</td>
<td>0.5431</td>
</tr>
<tr>
<td>Simple</td>
<td>0.3228</td>
</tr>
<tr>
<td>Does not require a lot of effort</td>
<td>0.4796</td>
</tr>
<tr>
<td>Easy to learn</td>
<td>0.3864</td>
</tr>
<tr>
<td>Conducive to thought process</td>
<td>0.4907</td>
</tr>
<tr>
<td>Expressive</td>
<td>0.6329</td>
</tr>
<tr>
<td>Model/diagram intuitive</td>
<td>0.5886</td>
</tr>
</tbody>
</table>

*Table G.3: P-Values for Questionnaire Responses about Participants’ Perceptions of the Methodology Used*
Appendix H

Participant 9’s Hermes Design

In the evaluation comparing Hermes’ goal-oriented approach against the message-centric interaction design aspect of Prometheus, Participant 9, who was classified as “intermediate” and allocated into the Hermes group, submitted a partial design which resulted in difficulties in its analysis. This led to the exclusion of the submission from the evaluation. In this appendix, this design is discussed in detail.

Participant 9’s interaction goal hierarchy and action map are shown in Figures H.1 and H.2 respectively.

![Figure H.1: Participant 9’s Interaction Goal Hierarchy (re-produced)](image)

From the submitted design, it is clear that the participant did not follow the Hermes process and the resulting Hermes design is not sensible. The first problem with the design is that there
is only one action map for the entire interaction when there should be one action map per atomic interaction goal. The submitted action map appears to be an attempt to cover the entire interaction, which is incorrect as action maps should be focused on achieving their atomic interaction goal.

![Diagram](image)

*Figure H.2: Participant 9’s Action Map (re-produced)*

The control flow in the action map is odd. For example, after the *Select Meeting Date and Time* action, either the *Check Availability of Participant* or the *Check Availability of Venue* is executed. According to the control flow, once one of these is executed, the other is not executed at all. Logically, both of these should be executed at some point in the interaction. The control flow is also incorrect at certain points, for example, final actions trigger other actions (final actions should not trigger any other actions). Furthermore, some of the actions in the action map do not appear to correspond to the steps in the scenario.

Although Participant 9’s submission was not included in the evaluation, we discuss the application of the metrics to the design and compare the results with the rest of the submissions next.

The *scenario coverage*, *robustness*, and *design time* metrics were able to be applied directly to Participant 9’s design. The *scenario coverage* (refer to Table H.1) and *robustness* (refer to Table H.2) values for Participant 9’s design are 7 and 1 respectively. The *design time* was 70 minutes and the *flexibility* values are shown in Table H.3.

Although three of the metrics were able to be applied directly to Participant 9’s design, the *flexibility* metric was not able to be directly applied due to the incorrect control flows which
lead to insensible flexibility graphs (refer to Appendix I). Thus, to sensibly apply the flexibility metric, Participant 9’s action map was modified slightly (shown in Figure H.3). It is important to note that the changes in Figure H.3 are our interpretation of Participant 9’s design. As this is our interpretation of the design, if it is to be included in the evaluation, it would invalidate it. Thus, in order to investigate Participant 9’s design, these results are separately presented from the evaluation in Chapter 6.

![Figure H.3: Participant 9’s Modified Action Map](image)

Compared to the results of the other submitted Hermes design, Participant 9’s design is clearly the weakest by far. In terms of scenario coverage, Participant 9’s design only covers 7 of the 14 steps whilst all other Hermes designs cover the scenarios perfectly and Prometheus designs range from covering 10 to 14 steps (refer to Table 6.4 on page 163).

Participant 9’s design also has the lowest flexibility values by far compared to other Hermes designs (refer to Table 6.5 on page 164). It has the same flexibility values as Participant 6’s Prometheus design, which has the lowest flexibility values in the entire set of submitted designs. For example, for the $m = 7$, $r = 7$, $c = 3$ scenario, the lowest Hermes flexibility value is 648 whilst Participant 9’s corresponding flexibility value is 8.

In regard to robustness values, Participant 9’s design again has a low value of 1 compared to other Hermes designs, which range from 3 to 7 (refer to Table 6.6 on page 167). The robustness value of Participant 9’s design is more comparable to Prometheus designs which have robustness values ranging from 0 to 3.

The design time for Hermes designs range from 145 to 320 minutes, whilst Prometheus design
### Table H.1: Scenario Coverage for Participant 9’s Design

<table>
<thead>
<tr>
<th>Scenario Step #</th>
<th>Participant 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>12</td>
<td>X</td>
</tr>
<tr>
<td>13</td>
<td>✓</td>
</tr>
<tr>
<td>14</td>
<td>✓</td>
</tr>
<tr>
<td>Total</td>
<td>7</td>
</tr>
</tbody>
</table>

### Table H.2: Robustness Results for Participant 9’s Design

<table>
<thead>
<tr>
<th>Variations</th>
<th>Participant 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2: Cannot find suitable date/time</td>
<td>✓</td>
</tr>
<tr>
<td>Step 4: Cannot determine room size</td>
<td>X</td>
</tr>
<tr>
<td>Step 6: Note enough funds</td>
<td>X</td>
</tr>
<tr>
<td>Step 7: Room is not available</td>
<td>X</td>
</tr>
<tr>
<td>Step 8: Room is no longer available</td>
<td>X</td>
</tr>
<tr>
<td>Step 10: Credit card list is empty</td>
<td>X</td>
</tr>
<tr>
<td>Step 11: Credit Card is declined</td>
<td>X</td>
</tr>
<tr>
<td>Step 13: Participant is no longer available</td>
<td>X</td>
</tr>
<tr>
<td>Total</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table H.3: Flexibility Results for Participant 9’s Design

<table>
<thead>
<tr>
<th></th>
<th>$m = 3, r = 3, c = 3$</th>
<th>$m = 4, r = 4, c = 3$</th>
<th>$m = 7, r = 7, c = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participant 9</td>
<td>4</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

*Table H.3: Flexibility Results for Participant 9’s Design*
times range from 45 to 240 minutes (refer to Table 6.7 on page 168). Participant 9’s design time of 70 minutes is the shortest time for a Hermes design and is also less than all Prometheus designs except for that of Participant 6 (52 minutes) and Participant 14 (45 minutes).

When Participant 9’s design is not included in the evaluation, statistical significance (i.e. a p-value lower than 0.05) is obtained for all metrics. However, the inclusion of Participant 9’s design results in statistical significance not being obtained for the scenario coverage and flexibility metrics (refer to Table H.4). This is due to the large difference between the values obtained from Participant 9’s design compared to the other designs and indicates that an evaluation with a larger number of participants may be required for the scenario coverage and flexibility metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Excluding Participant 9</th>
<th>Including Participant 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario Coverage</td>
<td>0.04895</td>
<td>0.1329</td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m = 3, r = 3, c = 3$</td>
<td>0.01632</td>
<td>0.06935</td>
</tr>
<tr>
<td>$m = 4, r = 4, c = 3$</td>
<td>0.01746</td>
<td>0.06731</td>
</tr>
<tr>
<td>$m = 7, r = 7, c = 3$</td>
<td>0.02564</td>
<td>0.08625</td>
</tr>
<tr>
<td>Robustness</td>
<td>0.001748</td>
<td>0.006119</td>
</tr>
<tr>
<td>Design Time</td>
<td>0.006993</td>
<td>0.02652</td>
</tr>
</tbody>
</table>

Table H.4: P-values with and without Participant 9’s Design
Appendix I

Calculating Interaction Flexibility

The flexibility of an interaction is defined by a flexibility value, which expresses the number of ways in which the interaction can be completed (successfully or unsuccessfully). This section explains how to determine the flexibility values of both Prometheus and Hermes interactions.

In order to determine flexibility values, flexibility functions must be first established and then evaluated. The main principle in determining the flexibility functions is to identify all possible paths through the interactions. As Prometheus and Hermes follow different approaches to interactions, the process for determining the flexibility functions is slightly different for each.

In the case of Prometheus, the messages of the protocol diagram are numbered, as in Figure I.1, an example of a Prometheus interaction protocol. To better visualize the paths through the interaction, the messages are placed end-to-end to create a graph as shown in Figure I.2. Note that the numbered directed lines (representing the messages of the interaction) correspond to the numbered messages in Figure I.1. The nodes are also labelled, although the labelling is arbitrary and does not correspond to anything from the interaction protocol.

![Prometheus Protocol Diagram]

*Figure I.1: Prometheus Protocol Diagram*
AUML Alternatives are captured as “splits” in the graph. For example, at the end of the second message, Credit Card Details, there are two possible replies from the Vendor: message 3, Payment Made, or message 4, Credit Card Declined. In the graph, the node after message 2 (node C) has two possible paths, one that follows the message 3 sequence and another which follows the message 4 sequence.

One issue is loops, which can be created by AUML continuations that go back to an earlier point in the interaction. If we treat these naively then the interaction will have an infinite number of possible paths, making comparison difficult. However, in practice, jump backs are usually constrained. For example, when message 4, Credit Card Declined, is sent, the interaction returns to a previous point and message 2, Credit Card Details, is sent again (as per the Make Payment AUML continuation on the protocol diagram). This jump back represents the selection of an alternative credit card, and an agent will have a finite (and usually small!) number of these. Thus we deal with loops by having a variable representing the number of times that the loop can be taken (e.g. the number of different credit cards, or the number of available meeting slots). We depict arcs which “consume” an option as dotted edges labelled with the relevant counter. For example, in Figure I.2 the arc from E back to B is dotted and labelled with a $c$, where $c$ represents the number of credit cards available, i.e. this arc can be taken at most $c$ times.

For Hermes interactions, the actions from the action maps are labelled as in Figure I.3. Although not strictly necessary, the messages between actions can be numbered. Similarly to Prometheus, a graph is created by placing messages end-to-end and, in the case of Hermes, ensuring that the message flow between each node is correct. Note that nodes in the graph correspond directly with actions from the action map. Although this particular Hermes interaction is the same as the previous Prometheus interaction, the resulting graph is slightly different but will yield the same flexibility values as the Prometheus graph.

The Hermes graph is shown in Figure I.4. Note that message 4, Credit Card Declined is marked by the variable $c$, which represents the number of alternative credit cards available. As with the Prometheus interaction, this path can only be taken a limited number of times, depending on the number of credit cards available.

The simple action map used in this example does not contain a rollback action. Rollback
actions are similar to AUML continuations in that they return the interaction to a previous point. As such, in the graph they are modelled in the same fashion as the AUML continuations: a dashed line limited by a variable which returns the interaction to a previous node.

Figure I.3: Hermes Action Map

Figure I.4: Hermes Interaction Message Paths

Once the graphs of the interactions have been created, the flexibility functions are then determined. Determining the flexibility functions is dependent solely on the graph and is not affected by which approach, Prometheus or Hermes, is used. Instead of deriving flexibility functions for both the graphs in Figures I.2 and I.4, we present a more complicated interaction which was submitted as part of the evaluation and proceed to use it as the example for the remainder of this section.

Figure I.5 shows Participant 14’s protocol diagram and Figure I.6 shows its graph. Note that Figure I.6 contains an arc that is not numbered. This arc does not correspond to a message from Figure I.5; rather, it corresponds to a choice being made: it has been added to distinguish between two states in the protocol that are not separated by a message. After message 2, Determine
Participant Availability, has been sent the interaction can either send message 3, Participant Not Available, or it can continue with obtaining a room. However, the label Reselect Room Size occurs before the next message. We need to add the additional (unlabelled) arc to distinguish between the state before the choice and the state after selecting the second option, but before sending message 4. If we do not do so then after message 7, Not Enough Funds, the interaction in the graph would allow either message 4 or message 2, which is not what the AUML protocol specifies.

To ease the creation of the flexibility functions, the graphs can sometimes be compressed into smaller graphs. This is done by compressing straightforward message sequences, such as messages 8–11 into one message, and the nodes are labelled. Splits, such as the nodes between the unnumbered message, messages 2 and 3 are retained. Merges, such as the node between messages 1, 2 and 3 are also kept. The compressed graph, from which the flexibility functions can be established, is shown in Figure I.7.

To determine the flexibility, the number of alternative message sequences must be determined. As the number of message sequences will vary depending on the values of the variables, a function, dependent on those variables, is determined for the interaction. This function is based on small, node-to-node functions.

For each node, a function which counts the number of ways in which the node can reach its successor node is determined. When these are aggregated we obtain a function which counts the number of paths from the first node to the end of the interaction.

The function is expressed in the following notation:

\[ \text{Node}_{m,r} \]

where Node = a particular node in the graph
\[ m = \text{number of alternative meetings available} \]
\[ r = \text{number of alternative meeting rooms available} \]

End nodes, i.e. nodes that have no out-going arcs, are counted simply as 1. For example, since \( E \) is at the end of the interaction, there is only one way in which it can reach the end of the interaction (it is already there), that is \( E_{m,r} = 1 \).

For other nodes, i.e. non-end nodes, the number of ways of reaching the end of the interaction is the sum of the number of ways of reaching the end from its successors. For each successor, the number of ways of reaching the end of the interaction will be the sum of its successor and so on.

For example, for the node \( A \), as \( B \) is its only successor, the function is \( A_{m,r} = B_{m,r} \). Similarly, as \( B \) has only one successor, \( C \), its function is: \( B_{m,r} = C_{m,r} \). Node \( C' \) also has only one successor, \( D \), its function is \( C'_{m,r} = D_{m,r} \).

If an arc is a loop, when that path is taken, the value of the loop variable is decreased by 1. For example, the node \( C \) has two successors, \( C' \) and \( B \). However, the arc leading from \( C \) to \( B \) has a loop variable, \( m \). Therefore the function for \( C \) is \( C_{m,r} = C'_{m,r} + B_{m-1,r} \) if \( m > 0 \), and
Appendix I. Calculating Interaction Flexibility

Figure I.5: Participant 14’s Protocol Diagram

Figure I.6: Participant 14’s Interaction Message Paths
is $C_{m,r} = C'_{m,r}$ if $m = 0$. Similarly, the function for $D$ is $D_{m,r} = E_{m,r} + C'_{m,r-1}$ if $r > 0$ and $D_{m,r} = E_{m,r}$ if $r = 0$.

The complete set of flexibility functions for Participant 14’s interaction is thus:

$$
A_{m,r} = B_{m,r} \\
B_{m,r} = C_{m,r} \\
C_{m,r} = C'_{m,r} + B_{m-1,r}, \text{ if } m > 0 \\
C_{m,r} = C'_{m,r}, \text{ if } m = 0 \\
C'_{m,r} = D_{m,r} \\
D_{m,r} = E_{m,r} + C'_{m,r-1}, \text{ if } r > 0 \\
D_{m,r} = E_{m,r}, \text{ if } r = 0 \\
E_{m,r} = 1
$$

To determine the flexibility values of the interaction, the flexibility functions are implemented and reasonable values for the variables (in this case, $m$ and $r$) are selected and substituted into the function for the start node, i.e. $A$.

Below we show an example evaluation of how the flexibility value for Participant 14’s interaction is determined for $m = r = 1$.

$$
A_{1,1} = B_{1,1} \\
= C_{1,1} \\
= C'_{1,1} + B_{0,1} \\
= D_{1,1} + C_{0,1} \\
= (E_{1,1} + C'_{1,0}) + C'_{0,1} \\
= (1 + D_{1,0}) + D_{0,1} \\
= (1 + E_{1,0}) + (E_{0,1} + C'_{0,0}) \\
= (1 + 1) + (1 + D_{0,0}) \\
= (1 + 1) + (1 + E_{0,0}) \\
= (1 + 1) + (1 + 1) \\
= 4
$$
Appendix J

Raw Data Plots of Participant Responses

![Figure J.1: Overall Ease of Use](image-url)
APPENDIX J. RAW DATA PLOTS OF PARTICIPANT RESPONSES

Figure J.2: Designed Interaction is Flexible

Figure J.3: Designed Interaction is Robust
APPENDIX J. RAW DATA PLOTS OF PARTICIPANT RESPONSES

Figure J.4: Designed Interaction is Simple

Figure J.5: Designed Interaction is Easy to Understand
Figure J.6: Designed Interaction was Easy to Design

Figure J.7: Designed Interaction was Fast to Design
APPENDIX J. RAW DATA PLOTS OF PARTICIPANT RESPONSES

Figure J.8: Designed Interaction is Easy to Follow

Figure J.9: Design Process is Well Guided
APPENDIX J. RAW DATA PLOTS OF PARTICIPANT RESPONSES

Figure J.10: Design Process is Easy to Use

(a) Designed Interaction – Easy to Use (Prometheus)

(b) Designed Interaction – Easy to Use (Hermes)

Figure J.11: Design Process is Fast

(a) Designed Interaction – Fast (Prometheus)

(b) Designed Interaction – Fast (Hermes)
APPENDIX J. RAW DATA PLOTS OF PARTICIPANT RESPONSES

Figure J.12: Design Process is Simple

Figure J.13: Design Process Does Not Require a Lot of Effort
Figure J.14: Design Process is Easy to Learn

Figure J.15: Design Process is Conducive to Thought Process
Figure J.16: Design Process is Expressive

Figure J.17: Model/Diagrams Used in Design Process Intuitive
Appendix K

Histograms of Participant Responses

Figure K.1: Overall Ease of Use
Figure K.2: Designed Interaction is Flexible

Figure K.3: Designed Interaction is Robust
Figure K.4: Designed Interaction is Simple

Figure K.5: Designed Interaction is Easy to Understand
Figure K.6: Designed Interaction was Easy to Design

Figure K.7: Designed Interaction was Fast to Design
APPENDIX K. HISTOGRAMS OF PARTICIPANT RESPONSES

Figure K.8: Designed Interaction is Easy to Follow

Figure K.9: Design Process is Well Guided
Figure K.10: Design Process is Easy to Use

Figure K.11: Design Process is Fast
Figure K.12: Design Process is Simple

Figure K.13: Design Process Does not Require a Lot of Effort
APPENDIX K. HISTOGRAMS OF PARTICIPANT RESPONSES

Figure K.14: Design Process is Easy to Learn

Figure K.15: Design Process is Conducive to Thought Process
APPENDIX K. HISTOGRAMS OF PARTICIPANT RESPONSES

Figure K.16: Design Process is Expressive

Figure K.17: Model/Diagrams Used in Design Process Intuitive
Bibliography


M. P. Singh. Social and psychological commitments in multiagent systems. In AAAI Fall Symposium on Knowledge and Action at Social and Organizational Levels, Monterey, California, USA, November 1991.


