CORE
Metadata, citation and similar papers at core.ac.uk
Provided by OpenGrey Repository

UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

An Incremental Process for the Development of Multi-Agent Systems in Event-B

by

Elisabeth Ball

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

August 2008

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Elisabeth Ball

A multi-agent system is a group of software or hardware agents that cooperate or compete to achieve individual or shared goals. A method for developing a multi-agent system must be capable of modelling the concepts that are central to multi-agent systems. These concepts are identified in a review of Agent Oriented Software Engineering methodologies.

The rigorous development of complex systems using formal methods can reduce the number of design faults. Event-B is a formal method for modelling and reasoning about reactive and distributed systems. There is currently no method that guides the developer specifically in the modelling of agent-based concepts in Event-B. The use of formal methods is seen by some developers as inaccessible.

This thesis presents an Incremental Development Process for the development of multi-agent systems in Event-B. Development following the Incremental Development Process begins with the construction of informal models, based on agent concepts. The informal models relate system goals using a set of relationships. The developer is provided with guidance to construct formal Event-B models based on the informal design. The concepts that are central to multi-agent systems are captured in the Event-B models through the translation from the goal models. The Event-B models are refined and decomposed into specifications of roles that will be performed by the agents of the system. Two case studies illustrate how the Incremental Development Process can be applied to multi-agent systems. An additional aid to the developer presented in this thesis is a set of modelling patterns that provide fault-tolerance for Event-B models of interacting agents.

Contents

Declaration of Authorship **xiii**

Acknowledgements **xv**

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 3
1.3 Contribution Overview 3
1.4 Thesis Organisation 4

2 Event-B **7**
2.1 Introduction 7
2.2 Event-B 8
2.2.1 Safety and Liveness of the Event-B models 12
2.2.2 Refinement 13
2.2.3 Decomposition 14
2.2.4 Tools 15
2.2.5 Specifying Event-Based Decomposition 16
2.3 Related Methods 17
2.4 Temporal Logic 19
2.5 Process Algebra 19
2.6 Summary 20

3 Agent-Oriented Software Engineering **21**
3.1 Multi-Agent Systems 21
3.2 Survey of Agent-Oriented Software Engineering Methodologies 23
3.3 Evaluation of AOSE Methodologies 27
3.3.1 Multi-Agent System Concepts 28
3.3.2 Complexity Management 28
3.3.3 Formality 29
3.4 Modelling Techniques for Multi-Agent System Concepts 30
3.4.1 Agent Interaction and Coordination 30
3.4.2 Agent Goals 32
3.4.3 Agent Roles 34
3.5 The Use of the B-Method and Event-B in AOSE 35
3.6 Summary 36

4 Incremental Development Process: Stage One - Goal Elaboration **39**

4.1 Using the Multi-Agent System Concepts 40
4.2 Stage One 41
4.2.1 The Goal Diagram 42
4.2.2 Goal Elaboration 43
4.2.3 Constructing and Refining the Event-B Models 45
4.2.4 Endpoint Goals 56
4.2.5 One-to-Many Interactions 59
4.3 Summary 61

5 Incremental Development Process: Stage Two - Distributed Coordination 63
5.1 Communicating Goals 64
5.2 Broadcast Communication 64
5.3 Role Allocation 65
5.4 Allocating Resources 66
5.5 Refining the Event-B Models 67
5.5.1 Broadcast Communication 68
5.5.2 Role Allocation 69
5.5.3 Resource Allocation 72
5.6 System Decomposition 78
5.7 Discussion on Patterns 81
5.8 Summary 82

6 Case Study : Query-If 85
6.1 Case Study 85
6.2 Stage One 86
6.3 Stage Two: Introducing Send and Receive 90
6.4 Stage Two: Introducing the Messaging Medium 92

7 Case Study : Contract Net 99
7.1 Case Study 99
7.2 Stage One 100
7.3 Stage Two 107

8 Fault-Tolerance Modelling Patterns for Multi-Agent Systems 115
8.1 Fault-Tolerance in Agent Interaction 116
8.2 Fault-Tolerance Patterns for Event-B Models of Multi-Agent Systems . . 117
8.3 Applying the Patterns 118
8.4 Initial Development Chain for the Contract Net 120
8.5 Timeout Pattern 124
8.6 Refuse Pattern 127
8.7 Cancel Pattern 130
8.8 Failure Pattern 132
8.9 Not-Understood Pattern 135
8.10 Related Work 138
8.11 Summary 139

9 Conclusion 141

9.1 Limitations . . . 143
9.2 Contributions . . . 143
9.3 Comparison of the Process to the Gaia and Tropos Methodologies . . . 145
9.4 Comparison of Goal Diagrams . . . 147
9.5 Comparison of Informal to Formal Model Translation . . . 149
9.6 Future Work . . . 151

A Query-If Case Study Event-B Models 153
A.1 Context . . . 153
A.2 m0 - Abstract Machine . . . 154
A.3 m1 - First Refinement . . . 156
A.4 m2 - Second Refinement . . . 159
A.5 Context 2 . . . 164
A.6 m3 - Third Refinement . . . 165
A.7 Context 3 . . . 172
A.8 m4 - Fourth Refinement . . . 174
A.9 m5 - Fifth Refinement . . . 181
A.10 Initiator - Component Model . . . 184
A.11 Participant - Component Model . . . 188
A.12 Middleware - Component Model . . . 191

B Contract Net Case Study Event-B Models 193
B.1 Context . . . 193
B.2 m0 - Abstract Machine . . . 194
B.3 m1 - First Refinement . . . 196
B.4 m2 - Second Refinement . . . 202
B.5 Context 2 . . . 213
B.6 m3 - Third Refinement . . . 213
B.7 Context 3 . . . 227
B.8 m4 - Fourth Refinement . . . 229
B.9 m5 - Fifth Refinement . . . 243
B.10 Initiator - Component Model . . . 250
B.11 Participant - Component Model . . . 258
B.12 Middleware - Component Model . . . 264
B.13 Example Proof Obligation . . . 266

C Fault-Tolerant Contract Net Case Study Event-B Models 275
C.1 Context . . . 275
C.2 m0 - Abstract Machine . . . 275
C.3 Context2 . . . 278
C.4 m1 - First Refinement . . . 279

Bibliography 293

List of Figures

2.1 Example Event-B Machine Structure 10
2.2 Notation Used to Represent Events 11
2.3 Event-B Context Structure 11
2.4 Comparing Event-Based and State-Based Decomposition 15
2.5 Model Relationships for Decomposition 17

4.1 A THEN Elaboration 44
4.2 An AND Elaboration 44
4.3 An XOR Elaboration 44
4.4 An OR Elaboration 44
4.5 How Goal Elaboration Can Relate to Event-B Refinement 46
4.6 Event-B Model for a Goal 47
4.7 Event-B Model for a THEN Elaboration 49
4.8 Alternative Representation of Goals in Event-B 50
4.9 AND-THEN Goal Elaboration 51
4.10 Event-B Model for an AND-THEN Elaboration 52
4.11 Event-B Model for an XOR Elaboration 53
4.12 OR-THEN Goal Elaboration 54
4.13 Event-B Model for an OR-THEN Elaboration 55
4.14 Goal Elaboration With an Endpoint Goal 56
4.15 Goal Elaboration Without an Endpoint Goal 56
4.16 Event-B Model for an XOR-THEN Elaboration with an Endpoint Goal . 58
4.17 Event-B Model for a Goal That Uses Broadcast Communication 59
4.18 Event-B Model for a Refinement That Uses Broadcast Communication . . 60

5.1 Goal Elaboration With Communicating Goals 64
5.2 Goal Elaboration With Broadcasting Goals 65
5.3 A Broadcast Goals Elaboration With a Reply 65
5.4 A THEN Elaboration With Role Allocation 66
5.5 Goal Diagram With Resource Allocation 66
5.6 Event-B Model for the Communicating Goals Elaboration 67
5.7 Event-B Model for Broadcast Goals Elaboration 69
5.8 Event-B Model With Role Allocation 71
5.9 Event-B Model for Directory Service Resource Refinement 73
5.10 Extended Context for Step One 74
5.11 State of the First Event-B Refinement for the Messaging Medium 74
5.12 Events of the First Event-B Refinement for the Messaging Medium 76
5.13 Further Extended Context for the Messaging Medium 77

5.14 Events From the Second Event-B Refinement for the Messaging Medium . 77
5.15 Synchronising the Refinement Model and the Abstract Component Models 79
5.16 Synchronising the Refinement Model and the Abstract Component Models 80

6.1 First Level of Goal Elaboration for Query-If 86
6.2 Query-If: Abstract Machine 87
6.3 Second Level of Goal Elaboration for Query-If 87
6.4 Query-If: First Refinement 89
6.5 Goal Model for Query-If 90
6.6 Query-If: Invariants of the Second Refinement 91
6.7 Query-If: Events of the Second Refinement (Part 1) 92
6.8 Query-If: Events of the Second Refinement (Part 2) 93
6.9 Query-If: State of the Third Refinement 93
6.10 Query-If: Example Events from the Third Refinement 94
6.11 Query-If: Extended Context for the Fourth Refinement 95
6.12 Query-If: State of the Fourth Refinement 96
6.13 Query-If: Example Events of the Fourth Refinement 96
6.14 Query-If: Synchronising of the Fifth Refinement and the Abstract Component Models 97

7.1 The First Level of Goal Elaboration for the Contract Net 100
7.2 Contract Net: Abstract Machine 102
7.3 The Second Level of Goal Elaboration for the Contract Net 103
7.4 Contract Net: Invariants of the First Refinement 104
7.5 Contract Net: Events of the First Refinement 106
7.6 Goal Model for Contract Net 107
7.7 Contract Net: State of the Second Refinement 108
7.8 Contract Net: Selected Events From the Second Refinement 109
7.9 Contract Net: Extract From the State of the Third Refinement 110
7.10 Contract Net: Example Events of the Fourth Refinement 111
7.11 Contract Net: Extended Context for the Fifth Refinement 111
7.12 Contract Net: State of the Fifth Refinement 112
7.13 Contract Net: Example Events of the Fifth Refinement 112
7.14 Contract Net: Synchronising the Sixth Refinement Model and the Abstract Component Models 113

8.1 Applying the Patterns to an Existing Model 118
8.2 Effect of Applying Patterns 119
8.3 Abstract Machine of the Initial Chain 121
8.4 Invariants of the Refinement of the Initial Chain 122
8.5 Events of the Refinement of the Initial Chain 123
8.6 Interaction Diagram for Timeout Pattern 125
8.7 Abstract Events for the Timeout Pattern in the Contract Net 125
8.8 Invariants for the Refinement of the Timeout Pattern in the Contract Net 126
8.9 Concrete Events for the Timeout Pattern in the Contract Net 127
8.10 Interaction Diagram for Refuse Pattern 128
8.11 Concrete Invariants for the Refuse Pattern in the Contract Net 129
8.12 Concrete Events for the Refuse Pattern in the Contract Net 129

8.13 Interaction Diagram for Cancel Pattern 130
8.14 Abstract Events for the Cancel Pattern in the Contract Net 131
8.15 Invariants for the Refinement of the Cancel Pattern in the Contract Net . 131
8.16 Concrete Events for the Cancel Pattern in the Contract Net 133
8.17 Interaction Diagram for Failure Pattern 134
8.18 Abstract Events for the Failure Pattern in the Contract Net 134
8.19 Invariants for the Refinement of the Failure Pattern in the Contract Net . 134
8.20 Concrete Events for the Failure Pattern in the Contract Net 135
8.21 Interaction Diagram for Not-Understood Pattern 136
8.22 Abstract Events for the Not-Understood Pattern in the Contract Net . . . 137
8.23 Concrete Invariants and Events for the Refinement of the Not-Understood Pattern in the Contract Net 137

B.1 Proof Obligation 267
B.2 Proof Information 268
B.3 Adding Hypotheses 269
B.4 Updated Goal 270
B.5 Running Prover 271
B.6 Discharged Proof Obligation 272

DECLARATION OF AUTHORSHIP

I, Elisabeth Ball, declare that the thesis entitled 'An Incremental Process for the Development of Multi-Agent Systems in Event-B' and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of the thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:

    1. Ball, E. and Butler, M. (2006) Using Decomposition to Model Multi-agent Interaction Protocols in Event-B. In Proceedings of FM'06 Doctoral Symposium, Available From: http://fm06.mcmaster.ca/11ElisabethBall.pdf, McMaster University, Hamilton, Canada.
    2. Ball, E. and Butler, M. (2007) Event-B Patterns for Specifying Fault-Tolerance in Multi-Agent Interaction. In Proceedings of MeMoT 2007 Methods, Models and Tools for Fault-Tolerance, Available From: http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/full-proceedings-final.pdf, Oxford, UK.

Signed:...

Date: ....

Acknowledgements

I would like to thank my supervisor Professor Michael Butler for his expertise and insight. Without his guidance I would not have been able to complete this thesis.

I would like to thank members of the DSSE research group for their support and advice. I am particularly grateful to my fellow students for their friendship and support.

I would like to thank those that have reviewed my work during my PhD as their feedback has been influential in the course of my work.

I have many things in my life to be grateful for, but perhaps the most important is Peter and the patience and understanding that he has shown me whilst I pursue my dreams. Without the support of my mother and my brother it would not have been possible for me to complete, or in fact begin, this work. Each member of my family and friends are an inspiration to me and have all helped to motivate and support me over the last few years. For that, and for everything else, I would like to thank them.

To my Father, who got me here

Chapter 1

Introduction

Computer systems are increasingly being required to solve distributed problems that exist in dynamic environments. Grid systems (De Roure et al. (2003)) are required to decompose submitted tasks to be able to make use of available computing power. Mobile communications devices are required to switch between heterogeneous networks. The software systems that are used to solve distributed problems need to be able to respond dynamically to change in both the problem and the environment.

Multi-agent systems are systems of distributed software entities that cooperate or compete to achieve individual or shared goals (Ferber (1999)). Agents encapsulate their behaviour and are motivated by their internal goals. The agents can individually respond, pro-actively and reactively, to changes in their environment (Jennings (2000)). The agent metaphor is one approach to creating software systems that are capable of solving distributed problems.

Multi-agent systems are complex (Cilliers (1998)). They need to be complex to be able to solve complex problems. Each agent acts autonomously according to their motivations. The agent's interactions are non-linear and can have a feedback affect on themselves. The system is capable of changing over time and each component does not have a complete awareness of the rest of the system. The development of multi-agent systems requires specialised software engineering methodologies for the application of such systems to become successful (Fisher and Wooldridge (1996)).

1.1 Motivation

The engineering of software systems can involve several phases of development (Pressman (2000)). Requirements analysis creates models of the problem domain and results in a specification of the tasks that the system must perform. The design phase takes the results of the analysis and models a system that will fulfill the requirements. The

implementation phase is when the software is written that complies with the design of the system. The software can then be tested against the requirements of the system to ensure that it performs as it was intended.

Engineering software is a difficult task with many problems that can arise, due to factors such as the complexity of the system and the change-ability of system requirements (Brooks (1987)). An ambiguous requirements specification can lead to the incorrect implementation of the system (Berry and Kamsties (2004)). Software designs can be inadequate and contain conceptual mistakes and subtle flaws that will lead to system failure (Jackson (2006)).

Testing software is not always adequate for discovering faults in system design (Jackson (2006)). The more complex a system becomes and the greater the number of possible interactions, the more susceptible it is to design faults (Rushby (1995)). The non-linear interactions that feedback into the system make it difficult to trace the source of any errors. Therefore, the complexity of the system will affect the amount of testing required. The more unpredictable the interactions of the system the greater the number of tests that will be required to provide adequate coverage. Relying on testing alone could lead to the execution of interactions that have not been performed by testing. This makes the use of rigorous design methods more important in the design of complex systems, including multi-agent systems.

Formal methods are the application of mathematics to model and verify software or hardware systems (Storey (1996)). The use of formal methods in software engineering can lead to a specification of a system that is unambiguous and can be formally verified to ensure it is consistent. Using a formal method to design systems has been found to reduce the number of design faults introduced into a software system and reduce the amount of testing the system requires (Hall (1996)). The design can be analysed for flaws before it is turned into programming code (Jackson (2006)).

Software engineering methods for developing multi-agent systems use agent concepts, such as organisations, agents, knowledge and motivation, as primary entities in models for analysis and design (Bordini et al. (2006)). Formal methods use mathematical notations to model software systems. This allows the models to be proven to be correct through formal verification. To be able to engineer multi-agent system using formal methods the agent concepts need to be expressed in the formal models.

Event-B is a mathematical approach for developing formal models of distributed systems that can be used to analyse and reason about the system (Abrial and Mussat (1998)). It is a methodology built on the theory of action systems (Back and Sere (1991)) that can be used to create models of reactive and parallel distributed systems. This makes it an appropriate formalism for modelling systems of distributed agents. Because of this, the focus of the work presented in this thesis is multi-agent systems rather than individual agents. Event-B has been used to model multi-agent systems with a focus on concepts

such as mobility and trust (Iliasov et al. (2006)), but there is currently no method that guides the developer specifically in the modelling of agent-based concepts using Event-B.

The mathematical basis of formal methods is seen as giving developers the perception that formal methods are inaccessible and require a large amount of training (Heitmeyer (1998); Hinchey and Bowen (1996); Hall (1990)). Improved tools and the use of 'lightweight' formal methods are considered to be two ways to make formal methods more accessible (Jones et al. (1996)). Another possible solution is to provide novice developers with expert advice (Hinchey and Bowen (1996)).

1.2 Objectives

The work presented in this thesis aims to create a method for developing multi-agent systems using Event-B. The method must allow fundamental aspects of agents to be captured. Creating an approach that can be seen as accessible would make the method useful to a greater number of developers.

This overall aim can be broken down into the following three objectives:

1. **Identify agent concepts.** There is currently no clear, widely accepted definition of what a software agent is and what the qualities are that it possesses. There are several concepts that can be used compose an agent and a multi-agent system. To be able to construct models of multi-agent systems the concepts that must be modelled need to be identified.

2. **Construct a method for modelling agent concepts in Event-B.** Event-B is a method for modelling reactive and distributed systems. This provides a useful basis for constructing models of multi-agent systems. This suitability can be increased by an additional method for modelling the identified agent concepts within the Event-B method.

3. **Make the method accessible.** There are advantages to using formal methods when developing software. When developing complex systems, like multi-agent systems, using rigorous methods can prevent errors that would be difficult to test for. The mathematical basis of formal methods can make them seem inaccessible to a developer. There are methods used in both formal methods and Agent-Oriented Software Engineering (AOSE) for managing the complexity of developing models. It is also possible to use informal techniques alongside the formal methods.

1.3 Contribution Overview

To achieve the objectives described above this thesis makes several contributions.

The key contribution of this thesis is the Incremental Development Process. The Incremental Development Process uses informal techniques to capture the system requirements using agent-based concepts and make modelling decisions about the functioning of the system. It describes how these informal models can be translated into Event-B models, integrating the agent concepts into the Event-B models.

Further contributions that have been made in the course of this work can be summarised as follows:

- The key concepts used in a selection of current AOSE methodologies have been identified. Methods used to manage the complexity of developing models of multi-agent systems have also been evaluated.
- The further contributions that can be identified separately of the Incremental Development Process include:
  - Guidelines for translating goal diagrams into Event-B models.
  - The use of the events and variables in Event-B to model the agents fulfilling their goals.
  - The use of decomposition for modelling agent roles in Event-B.
- This thesis includes two case study models that have the potential for re-use.
- The construction of the case study models led to the identification of patterns for fault-tolerance in multi-agent systems. Alongside the patterns further contributions are.
  - The identification of how the interactions of multi-agent systems can use fault-tolerance to increase the dependability of the agents.
  - An examination of how patterns can be applied in Event-B.

1.4 Thesis Organisation

This thesis describes the contributions outlined above and how they have been achieved. The remainder of this thesis is structured as follows:

Chapter 2 introduces Event-B. The notation and method used to develop formal models of systems using Event-B are described. The use of refinement in Event-B, that takes a model of an abstract interpretation of requirements to a more detailed model of the system, is described. There is a discussion of the decomposition methods that can be applied to Event-B models. The RODIN tool set that can be used in Event-B development is also described.

Chapter 3 provides more detail on multi-agent systems. A definition of individual agents is included and from that multi-agent systems are described. The chapter includes a survey of a selection of AOSE methods. Each of the methods is described and then they are all evaluated to find the common concepts required to model multi-agent systems, how the complexity of agent systems can be managed during the design process and how formal methods can be used to model multi-agent systems. The main outcome of the evaluation is the identification of the concepts that are needed in a method for modelling multi-agent systems using Event-B.

Chapters 4 and 5 describe the Incremental Development Process that is the main contribution of this thesis. Chapter 4 introduces Stage One of the Incremental Development Process. The chapter describes how goal models of a system can be constructed by refining abstract goals with a set of relationships. Guidance is then given for translating the goals and their relationships into the elements of an Event-B model. The translations can be applied as refinement steps to create an Event-B refinement tree that corresponds with the refinement of the goals.

Chapter 5 describes how the models of system goals constructed in Stage One can be developed using Stage Two of the Incremental Development Process to model the coordination of the agents in the system. The Incremental Development Process then continues by refining and then decomposing the system model into abstract component models of the agent roles required by the system.

Chapters 6 and 7 contain two case studies that have been developed using the Incremental Development Process. The case studies are based on multi-agent interaction protocols. The first case study provides an example of a three stage interaction between two agents. The second case study is of a multi-stage interaction between multiple agents.

Chapter 8 presents a further contribution of this thesis. A set of modelling patterns that extend Event-B models to include fault-tolerance for the interactions of a multi-agent system are described. One of the case studies developed in the previous chapters is used to show how the patterns can be applied.

Chapter 9 concludes this thesis. The contributions of the thesis are discussed. Limitations of the work, including the Incremental Development Process, are examined. Comparisons are provided between the Incremental Development Process and two of the surveyed AOSE methodologies, related work using goal diagrams and related work translating between informal and formal models. Possible directions for future work are described.

Chapter 2

Event-B

This chapter provides background knowledge on the Event-B formal method. The Event-B method is introduced with a description of the philosophy behind the method followed by an introduction to the constructs used for modelling systems in Event-B. How liveness and safety properties can be modelled and proven in Event-B is described. More detail is given on the refinement method used to develop Event-B models and the decomposition methods that can be applied to Event-B models. The tools that are used to develop the Event-B models used in this thesis are introduced followed by a description of how the tools have been used to apply event-based decomposition to the models. Further background is provided with a description and comparison of some related formal methods and an overview of temporal logics.

2.1 Introduction

Formal methods are the application of mathematics to model and verify software or hardware systems (Storey (1996)). Mathematically based languages can be used to write specifications of systems using precise rules. The specification can be interpreted unambiguously and can be formally verified to ensure consistency and correctness. Formal methods have been used to develop applications such as air traffic control (Hall (1996)), railway signaling (Behm et al. (1999)) and transaction processing systems (Houstan and King (1991)).

Formal verification involves the application of mathematical proofs to every possible behaviour allowed by a specification (Abrial (1996)). In a state-based specification the behaviour is a transformation of the system moving from one state to another. Proof obligations are generated using the specification and the language rules. These proof obligations then need to be discharged using properties of the specification in order to prove the correctness of the specification. The generation of proof obligations, and to some degree the discharging of proof obligations, can be automated.

Model Checking is another verification technique available for formal specifications (Clarke Jr. et al. (2000)). Model checking tools will take a specification and automatically construct a state-transition graph for the specification. The state-transition graph represents the set of states, the transitions between the states and the properties that are true in each state. The state-transition graph can then be searched to find states that are inconsistent with the specification and to ensure that the states in the graph are reachable. If no inconsistent states are found then the specification is considered to be consistent. This technique is only possible for finite state systems and requires a lot of computing resources.

2.2 Event-B

Event-B is a mathematical approach for developing formal models of systems (Abrial and Hallerstede (2006)). An Event-B model is constructed from a collection of modelling elements. These elements include *invariants*, *events*, *guards* and *actions*. The modelling elements have attributes that can be based on set theory and predicate logic. Set theory is used to represent data types and the manipulation of data. Logic is used to apply conditions on the data. The development of an Event-B model goes through two stages; *abstraction* and *refinement*. The abstract machine specifies the initial requirements of the system. Refinement is carried out in several steps with each step adding more detail to the system, generally, but not exclusively, in a top-down manner.

Reactive systems (Harel and Pnueli (1985)) are systems that continually respond to changes in their environment. The focus on atomic events in Event-B creates a representation of a reactive system (Jones (2005)). The model transitions are triggered by changes in the state of the model, which can represent the system's environment. The guard of an event represents the necessary conditions on the state of the system for the event to be triggered. When the guard is true the actions of the event may be executed, possibly changing the state and allowing another event to be triggered. The guard on an event will allow or prevent an event from occurring depending on the state of the model. When none of the guards are true the system is deadlocked. When more than one guard is true one of the events is chosen non-deterministically and executed.

Event-B is designed for modelling distributed systems (Abrial and Hallerstede (2006)). It implements the theory of discrete transition systems. Discrete transition systems, or action systems, model atomic actions that can be performed in parallel providing the actions do not affect the same state variables. Similarly to Event-B, action systems are state-based with each action, guarded by a set of conditions, performing updates on the state. Action systems also use stepwise refinement to refine an abstract model of system behaviour to a more concrete model (Back and Sere (1991)).

The model of atomic events in Event-B assumes that the same part of the state is not updated concurrently by separate events that could belong to separate processes. Without modelling concurrent execution Event-B models cannot model solutions to the problems that may arise due to concurrency. One method for specifying concurrency in Event-B is to model each update as a group of potentially interleaving atomic events (Edmunds and Butler (2008)). This allows the model to specify how concurrent execution can be dealt with by the system being modelled.

The difficulties in modelling concurrency in Event-B means that it is not always suitable for modelling all reactive distributed systems. Reactive systems, as with other system paradigms, can have concurrently executing components. In distributed systems the concurrent execution of processes is the norm (Coulouris et al. (2000)). Multi-agent systems normally include potentially concurrent processes. The Event-B models of multi-agent system will model system that is protected from the potential problems of concurrency. This is useful when modelling at a high level of abstraction where the mechanisms for coping with concurrency would not be modelled. The models created in this thesis model systems at a high level of abstraction. When refining the models to a more concrete level using Event-B the mechanisms for coping with concurrency could be modelled using interleaving events, as described above, or a different formal method that models concurrency, such as a process algebra, could be used. Reactive systems can also terminate and liveness properties in the Event-B method, which is discussed in detail below, are aimed at non-terminating systems. This problem can be overcome by not generating the proof obligations for the liveness properties.

Specifying a distributed system in Event-B takes a global approach. Rather than creating a specification for each component of the system it is modelled as a whole along with its environment. The model is closed in that it reacts only to changes in its internal state. Initially states are modelled abstractly with the events that describes the main goal of the system. Detail is added through refinement to describe the final distributed system. The ability to add new events and refine single events into multiple concrete events allows the functionality of the system to expand beyond that modelled in the abstract machine. Refinement ensures that the refined models are consistent with the abstract machine.

Event-B is intended to be extensible. The approach is designed so new modelling elements can be added without affecting the underlying method. Figure 2.1 shows how the basic modelling elements can be structured to form an Event-B machine. This first element is the context for the model. The context is an Event-B component that contains the static properties of the model. The model is a refinement and this is shown by the next modelling element that indicates the abstract machine *m0*. The third modelling element is a variable, *num*, that models the state of the machine. The type of the variable, $num \in \mathbb{N}$, is defined by the invariant modelling element that follows the variable modelling element. Most machines will have multiple variable and invariant elements.

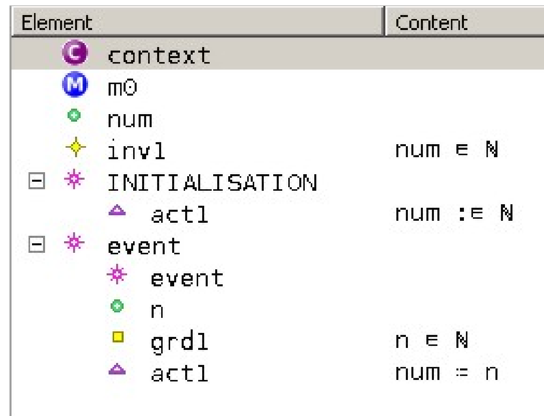

FIGURE 2.1: Example Event-B Machine Structure

Invariants are used to define necessary properties on a model. The modelling elements so far have specified the state of the system. The event modelling elements specify the actions that can be taken on the state. The first event is the *INITIALISATION* event that is present in all Event-B models. This event initialises the variables of the model. Because the model only has one variable there is only one action modelling element for the *INITIALISATION* event. The action contains a generalised substitution that non-deterministically assigns an element of the set of natural numbers to the state variable *num*. The next event has a refines clause, event variable, guard and action modelling elements. The refines clause states the name of the event in the abstract model that this event refines. The content of the guard is a predicate that restricts the value of the event variable, n, to an element of the set of natural numbers, $n \in \mathbb{N}$. The content of the action is a generalised substitution that assigns to the state variable *num* the value of the event variable n.

Notation can be used to construct models that present the modelling elements in a textual format. In this thesis the events of the Event-B models will be presented using the keywords `ANY`, `WHERE`, `THEN` and `END` to delimit the elements used. The event variables of an event will be written between `ANY` and `WHERE`. The guards of the event will be written between `WHERE` and `THEN` and the actions of the event will be written between `THEN` and `END`. Figure 2.2 shows the event taken from the machine in Figure 2.1 using this notation. It is intended that this notation will make the models more readable.

The static data properties of the model are specified in a context model and the properties that model the behaviour of the system are specified in the abstract model and refinements. Separating the constant properties of a model means that they can easily be replaced with a different set of constant properties and the model can be instantiated in a different context. This provides an opportunity for generic instantiation. An example would be a model of a sorting algorithm that can either sort a set of records or a set of integers. Reusing the model in this way prevents the need to re-prove the proof

```
ANY n WHERE
  n ∈ ℕ
THEN
  num := n
END
```

FIGURE 2.2: Notation Used to Represent Events

Element	Content
context0	
OBJECT	
objectId	
axm1	objectId ∈ OBJECT → ℕ

FIGURE 2.3: Event-B Context Structure

obligations for the model. Contexts can be extended by another context. The extension of contexts does not have to match the refinement chain of a development (Méteyer et al. (2005)). An example context model is shown in Figure 2.3. The first modelling element of the context specifies the name of the context that this context extends, *context*0. The next element is a carrier set that is labelled *OBJECT*. Carrier sets can either be deferred sets or they can be enumerated by constants. The context also contains a constant modelling element labelled *objectId*. The axiom modelling element contains a predicate that defines the constant *objectId* as a total function between the carrier set *OBJECT* and the set of natural numbers. This specifies that each object has an objectId.

Event-B models are verified by formal verification. The proof obligations generated for an Event-B model require it to be proven that the invariant conditions for the model are preserved by the events (Méteyer et al. (2005)). For the invariant conditions to be preserved the actions of the events cannot affect the state variables of the model in a way that will transform the model into a state where an invariant condition is false. The proof obligations generated can help the developer to understand the properties of a model. They can help to identify inconsistencies in the model. If a proof obligation cannot be discharged then there may be an inconsistency in the model between the invariant and the event that generated the proof obligation. Complex proof obligations can highlight complex properties in a model. Simplifying the properties will simplify the proof obligation and make the model more comprehensible. Model checking can be used to automatically check Event-B models for consistency and violation of invariant conditions (Leuschel and Butler (2003)).

The Event-B method begins with the specification of an abstract machine that describes the abstract requirements of a system. The model is refined to describe more of the requirements. The process of refinement ensures consistency with the initial abstraction whilst allowing the model to become complex enough to represent real systems. Refinement in Event-B will be described later in this chapter.

2.2.1 Safety and Liveness of the Event-B models

Safety and liveness are properties of a formal model that ensure the correct and continuous processing of the model. A safety property specifies that something bad will not happen and a liveness property specifies that something desirable will eventually happen (Lamport (1977)).

Safety in Event-B is specified through events and invariant conditions. For example, an event can be added to a model that will move the state of the model into a fail state when failure has occurred. Invariant conditions can be added to the model specifying that the model cannot be in a functioning state at the same time as a failure state. The proof of this invariant will ensure that the functioning events of the system cannot occur when the model is in the failure state.

Liveness in an Event-B model is based on the model being free from deadlock and divergence. Deadlock-freeness in an Event-B abstract machine can be ascertained by proving that the invariant of the model implies that at least one of the events is enabled.

When specifying a system it may be that the developer of a system would wish it to deadlock, for example, in a safety critical situation. A developer can ensure themselves of the enableness of desired events by animating the model and stepping through the model transitions. Once the desired liveness is ensured in the abstract machine it is important that any refinement does not impede the liveness property of the abstract machine.

The refinement of an abstract model in Event-B allows new events to be introduced. To be able to prove that the liveness property of the abstract model is upheld two properties of a refinement need to be proven: non-divergence and enableness.

The non-divergence property is required to prove that new events, once enabled, do not take control of the processing and prevent events that were enabled in the abstract model from occurring. To be able to prove this a variant is added to the model. The variant must be proven to decrease each time the new events occur and that it does not decrease below a base. This will prove that the new events will eventually be prevented from occurring as the variant cannot continue being decreased forever. Where non-divergence proves that the new events will eventually stop occurring enableness will prove that when the events in the abstract model were enabled in the abstract model they will also

be enabled in the refinement. To be able to prove enableness it must be proven that the guard of the abstract event implies the guard of the refined event or the guards of new events in the model: $grd(A) \Rightarrow grd(A') \vee grd(H)\ where\ H = all\ new\ events.$

2.2.2 Refinement

The refinement of a model is the process of adding more detail, in a stepwise manner, to a model. The refinement of an Event-B abstract machine can be carried out in several steps. More detail is added to the model at each step. Classic stepwise refinement proceeds in a top-down fashion (Back et al. (1998)). In large developments this is not always the case and refinement can be an iterative process.

The advantage of using refinement is that it allows the model to be analysed at an abstract level where the complexity is reduced (Abrial and Hallerstede (2006)). The detail of the model can be introduced over several steps with each step being available for separate analysis. A refinement model can be proven to be consistent with the abstract model using formal verification.

An Event-B model can be refined in several ways. The state variables of an Event-B model can be refined into a more concrete representation. New state variables can be introduced to the model to increase the detail of the system that can be modelled. The guards of the event can be strengthened to place further conditions on the event and state variables.

In Event-B the relationship between the variables in the abstract model and the more concrete variables in the refinement is described as a set of invariant conditions for the refinement. This set of invariant conditions is known as the *gluing invariant* (Abrial (1996)). The events in a refinement model must be proven to have the same affect as the events of the abstract model on the variables of the abstract model. The gluing invariant can be used to prove the correctness of the refinement with regard to the abstract model by specifying the relationship between the abstract variables and any refinements of the abstract variables. As with the other invariant conditions, the conditions specified in the gluing invariant have to be proven to be preserved for each event.

New events can be added to an Event-B refinement (Méteyer et al. (2005)). New events can only modify the variables new to the refinement model, otherwise, the refinement may become inconsistent with the abstract model.

A single event can be refined into more than one concrete event (Méteyer et al. (2005)). The guards and actions of the concrete events must be, as with single event refinement, a refinement of the guard and actions of the single abstract event. The guards of the concrete events can be stronger than the guard of the abstract event.

Refinement helps to reduce the burden on the developer by allowing an abstract model of the system to be analysed and the detail of the system to be introduced in a stepwise manner. The more detailed refinement models can become complex and difficult to analyse. This problem can be solved by decomposing the model into more manageable components.

2.2.3 Decomposition

Decomposition makes it possible to manage the complexity of the models that increases through the refinement process. Decomposition splits the model into separate components that model individual parts of the system. The developer can then further develop the components concentrating on each component separately. A further motivation when specifying distributed systems is to mimic the eventual physical distribution of the separate components. Decomposition is especially useful when modelling systems that contain complex subsystems, such as agents, as it can be re-applied at the different levels of the system hierarchy (Jennings (1993)). Another advantage of decomposition is that the components can be re-used (Pressman (2000)). If components are modelled to encapsulate a functionality and that functionality is separable from the development context then it is possible to re-use the component. Two forms of decomposition may be used in Event-B, *event-based* and *state-based*.

Event-based decomposition separates the model on its events. The variables are encapsulated in the separate components and the events or parts of the events that affect the variables are specified in that component model. The events that have been split will then need to be synchronised in order to recreate the functionality of the original system model. This can be achieved by exchanging inputs and outputs between the synchronised component events.

Event-based decomposition is based on a parallel composition method developed for action systems (Butler (1996)). In this method the different components contain shared events that can be hidden after composition. The components are able to synchronise on the events with one component taking as an input an output of the event in the synchronising component. This technique for event-based decomposition has been applied to Event-B in Jones (2006).

A different method for decomposition that can be used with Event-B is state-based decomposition (Abrial and Hallerstede (2006)). The variables are split between the different components with some variables being shared by the events in the different components. Events are added to components to simulate how the shared variables are used in the other components. The shared variables and events must be kept synchronised between the components during refinements. The system can be rebuilt into a single model later in refinement.

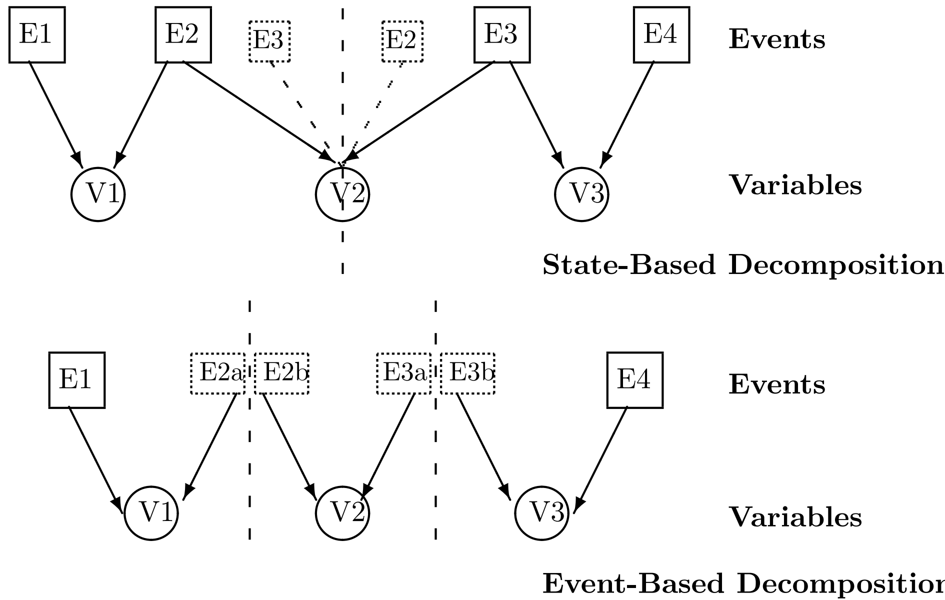

FIGURE 2.4: Comparing Event-Based and State-Based Decomposition

Figure 2.4 gives a diagrammatic comparison of the effect of using state-based and event-based decomposition. The state-based decomposition shows that variable **V2** has been split between the two components and the action of event **E3** that affects **V2** has been added to the first component and the action of event **E2** that affects **V2** has been added to the second component. The event-based decomposition shows that each variable has been placed in a separate component and the events that affect the components e.g. **E2** have been split into the parts that affect the different variables, e.g. **E2a** and **E2b**, and placed in the component that contains the affected variable.

Decomposition has currently not been implemented in the tool support that is available for Event-B. The tools available for the B-Method can be used to simulate event-based decomposition in Event-B by using parameters and the structuring mechanisms from the B-Method.

2.2.4 Tools

The RODIN platform is a tool environment that supports modelling using the Event-B method (Abrial et al. (2006)). The platform is built on the Eclipse platform, which allows it to be extended by plug-ins (Eclipse (2007)). Event-B machines and contexts can be created within projects. The models can be refined and the contexts can be extended. The proof obligations for the models are automatically generated. The theorem provers can be used to automatically discharge proof obligations and an interactive proof environment is provided for proof obligations that cannot be discharged automatically.

ProB is a model checker for B that has been made available as a plug-in for RODIN. It examines the reachable states of a model for consistency with the specification. The specification can also be animated using ProB (Leuschel and Butler (2005)). This allows

a user to step through the state space of the specification triggering the events and displaying the state of the model. This is useful for testing the specification for expected behaviour.

The RODIN platform has been used to develop the models presented as case studies in this thesis. Most of the models in this thesis were verified using the RODIN platform, some automatically and others interactively. An example of a proof obligation being discharged interactively using the RODIN platform can be found in Appendix B. The RODIN platform does not currently support decomposition. To be able to model and verify the event-based decomposition used in the case studies other software tools were used.

2.2.5 Specifying Event-Based Decomposition

Event-B is an evolution of the B-Method (Abrial (1996)) and most of the semantic rules for Event-B are already part of the B-Method. This means it is possible to use the tools developed for the B-Method to specify Event-B models, but care has to be taken that the specification remains within the Event-B rules as these are not enforced by the tools.

Several software tools are available to aid development using the B-Method (Butler et al. (2005)). The tools used to specify and verify the event-based decomposition, in this work, are B4Free project manager and logic solver (ClearSy (2005)) with the Click 'n' Prove interface (Abrial and Cansell (2003)). B4Free is a set of tools to help with the formal verification of a B project. The tools generate proof obligations and prove them either automatically or, when this is not possible, provide an environment for the developer to prove them interactively. The Click 'n' Prove interface makes B4Free more usable as a graphical interface that guides the user through specification and interactive proofs.

The main differences for Event-B are the use of guarded events over operations in the B-Method, the ability to introduce new events in refinement models, refining events into more than one event and the use of a context to specify the constant properties of a development. Guards for operations are available in the B-Method and so these can be used when specifying events. A context could be used in the same way as it would be in tools for Event-B. The *sees* structuring relationship is part of the B-Method and can be used to access the properties in the context machine. For the decomposition no new events are introduced and no events are refined into more than one event. Hallerstede (2007) provides a further discussion on the differences between Event-B and the B-Method.

To model event-based decomposition each component has to be modelled as a separate Event-B abstract machine. Each component model encapsulates the required variables and the events that affect them. The variables of the models have to be encapsulated

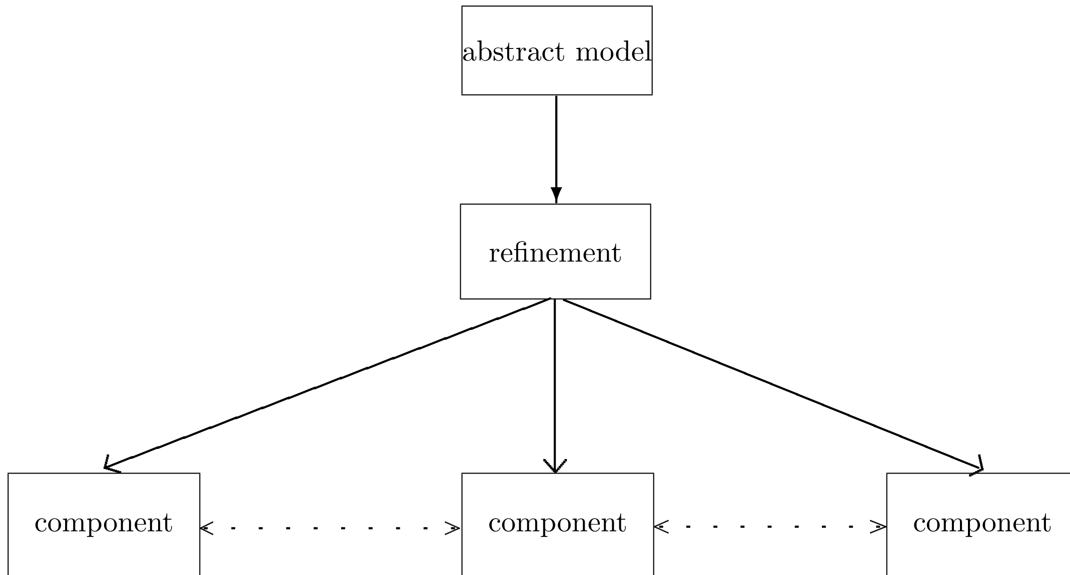

Figure 2.5: Model Relationships for Decomposition

within the separate components, which requires the variables to be refined to a point where they could be separated where necessary.

To be able to generate and discharge the proof obligations, that will prove that the components refine the abstract model, the components need to be synchronised. The B-Method has a structuring mechanism called *includes* that allows abstract machines to be included in another model. The events and variables of the included model can be accessed by the events of the including model. To model the decomposition a refinement can be produced that includes the component models and refines the abstract system model. The events of the component models can then be synchronised within the refining events of the synchronising refinement and parameters, used in the B-Method, can be used to pass values between the component events. The synchronised events can then be proven to refine the abstract model. B4Free and Click 'n' Prove can be used to perform this decomposition and prove that the synchronised components are a correct refinement.

Figure 2.5 show how the abstract component models are related to the refinement chain. The notation used is taken from Back (2005) with the filled arrowhead indicating refinement and the open arrowhead indicating dependency. The dashed lines used in the dependency relationships between the components indicates an implied dependency. The *includes* structuring relationship is used in the synchronising refinement model to create the dependencies.

2.3 Related Methods

There are several formal methods based on structured specifications using a notation of set theory and predicate logic. Below we summarise three of these methods and provide

a brief comparison to Event-B.

Z is a formal language for specifying software systems (Diller (1994)). A Z specification is modelled in a schema that contains declarations and formulas. The declarations specify variables and their types. The formulas specify both predicates on the variables and actions on the variables. The predicates and formulas are constructed using set theory and predicate logic.

Schemas in Z can be structured through schema inclusion where the name of one schema is included in the declarations of another. This allows large specifications to be incrementally constructed from component schemas. Schema inclusion can also be used to relate an abstract specification to a refinement with the formulas of the inclusion schema specifying the gluing invariant.

The Vienna Development Method (VDM) (Jones (1990)) is a method that creates modules that include declarations of variables, other state constructs and operations with pre and post conditions. Refinement of the modules is theoretically separated into data reification and operation decomposition. Data reification refines the abstract specification of data types into more structured representations. Operation decomposition occurs at the later stages of the refinement process and refines the abstract operations into a more algorithmic representation. The specification language used to construct VDM models includes programming constructs, such as while loops. The specification language is based on a 3-valued logic that allows a predicate to be true, false or undefined. Modules, or parts of modules, can be imported into other modules to construct a system model.

Event-B is an evolution of the B-Method. The B-Method is a formal method for specifying software programs (Abrial (1996)). Machines include declarations of variables, invariants, other state constructs and operations. Operations in the B-Method have pre and post conditions, which make their execution different from the guarded atomic events of Event-B. Refinement in the B-Method is carried out in the same way as with Event-B. Instead of decomposition, machines and refinements can be related using structuring mechanisms. The final refinement model, known as the implementation model, can include programming constructs, similarly to VDM modules.

VDM, the B-Method and Event-B are development methods that apply formal specification languages, whereas Z is a formal specification language that can be applied by following a development method. The use of modelling elements in Event-B allows the method to be extended to use other specification languages. Z, the B-Method and VDM do not offer extensibility in the same manner. The modelling elements used in Event-B also provide a structure within the model. Z schemas are separated into two parts. This provides a lot of flexibility, but at the cost of readability of the specifications. VDM and the B-Method provide more structure than that of Z with types, state and operations clauses in the specification and the operations include their pre and post conditions. Z,

VDM and the B-Method compose together models of components into a system model. Event-B abstractly models the entire system and decomposes it into components as it is refined. This approach should ensure that all aspects of the system are modelled. Event-B does not include programming constructs. This is because Event-B is developed for creating formal models of systems and then reasoning about them rather than creating formal models of software programs.

2.4 Temporal Logic

This section provides a brief overview of temporal logic, which is discussed later in this thesis.

Modal logic extends first-order logic with modal operators. Temporal logic is modal logic that uses temporal operators such as $\square$(always), $\diamond$(eventually), N(next) and U(until) to make statements about model states (Fagin et al. (2003)).

Temporal logics model several states and formulas that can be true or false in those states. There are several different temporal logics (Huth and Ryan (2004)). Two examples of temporal logics are Linear Time Temporal Logic (LTL) and Computational Tree Logic (CTL). LTL models paths in the future that can represent different possible futures. CTL is a Branching Time Logic that models the future as a tree-like structure and allows quantification over the different paths. CTL allows the different possible future paths to be analysed in more detail than LTL.

2.5 Process Algebra

This section provides an overview of process algebras, which are an example of a formal method that can be used to model multi-agent systems.

Process algebras model concurrent processes that interact through communication. A process algebra generally consists of two specifications, one that captures the design of the system and one that describes the high-level behaviour of the system. These specifications should be proven to be either equivalent or that the design refines the behaviour (Bergstra et al. (2001)).

CSP (Hoare (1985)) is an example of a process algebra. It consists of processes and events that are used to describe the behaviour of a system. For example, a specification of a process called *LIFT* that will recursively execute the events *up* and *down* could be specified as: $LIFT : up \rightarrow down \rightarrow LIFT$, where *up* must occur before *down*. This specification models a lift that will continually move between two floors.

The $\pi - calculus$ (Sangiorgi and Walker (2001)) is another example of a process algebra. It models mobile systems where the structure of the system can change over time. The processes interact through shared names. Names can be passed to processes allowing them to communicate with previously unknown processes and new names can be created.

Process algebras do not provide a record of state, as Event-B and other state transition systems do, as they mainly concentrate on communicating processes not shared variables.

Process algebras allow concurrent distributed systems, such as multi-agent systems, to be modelled succinctly. A multi-agent interaction protocol could be modelled in CSP as a few processes and events. However, the information state of an agent, e.g. its motivations and beliefs, would not be modelled as easily as with a state transition system.

Because of the strength of process algebras on modelling concurrent processes and state transition systems on modelling state there are several examples of work to combine both formalisms (Treharne and Schneider (1999); Butler and Leuschel (2005)). Using a combination of process algebra and Event-B would complicate the guidance given for translating the goal models in the Incremental Development Process.

2.6 Summary

Event-B is a formal method that allows formal verification of model properties through the discharge of proof obligations. Event-B is based on action systems and is suitable for modelling distributed systems.

Refinement and decomposition can be used to manage the complexity of modelling systems using formal methods. Refinement allows the detail of the models to be added in a stepwise manner. This refinement can then be formally verified. The decomposition techniques available for Event-B allow the model to be separated into component models reducing the complexity from that of a single model.

Event-B has the advantage of having tool support. Tools that support the B-Method can be used to verify event-based decomposition for Event-B models.

Chapter 3

Agent-Oriented Software Engineering

This chapter provides background on the field of Agent-Oriented Software Engineering (AOSE). The chapter begins with a description of agents and multi-agent systems. A survey of AOSE methodologies describes and evaluates a selection of methodologies that can be used to develop multi-agent systems. Following the survey, modelling techniques for the the concepts that are evaluated as the most common for multi-agent systems from the survey are examined. Related work on using Event-B to model multi-agent systems is also described.

3.1 Multi-Agent Systems

The way that software is designed has had to reflect the change from structured programming techniques to the object-oriented style of modern programming languages. The agent paradigm takes the modelling of software systems to a higher level of abstraction than object-orientation. The methods used to model multi-agent systems need to be capable of modelling their main concepts. To find the best approach for modelling multi-agent systems in Event-B, current approaches, using both formal and informal methods, need to be reviewed.

There is currently no clear, widely accepted definition of what a software agent is and what the qualities are that it possesses. The level of abstraction at which the agent model is based is one that can be ascribed intentional properties (Wooldridge and Jennings (1995)). For this reason software agents are often seen as intelligent agents. These intelligent agents are given properties that represent beliefs, desires and other intentional states. These states can be represented using logical notations. Some suggest that it is necessary for an agent to have an intentional system (Franklin and Graesser

(1996),Wooldridge and Jennings (1995)). Other definitions suggest that an agent need not be autonomous nor possess any form of intelligence before they can be described as agents (Luck and d'Inverno (1995a)). Attributes, such as autonomy and intention, are seen as increasing the degree of agency that the software exhibits.

To be able to model multi-agent systems there is a need to define the qualities that an individual agent will be considered to possess. For this purpose a definition of these properties has been taken from Wooldridge and Jennings (1995).

- autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
- social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
- reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
- pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.

A multi-agent system is a grouping of agents that either cooperate or compete in order to fulfill individual or collective goals. The agents communicate with one another by passing messages that allow them to coordinate their actions (Ferber (1999)). Because agents are autonomous and reactive their behaviour within the system cannot be predicted and therefore the overall behaviour of the system cannot be predicted. This emergent behaviour is what makes multi-agent systems ideal for solving dynamic, unpredictable problems. But, some of this behaviour can be undesirable and lead to system failures (Greaves et al. (2004)).

Jennings (2000) argues that all complex systems are distributed and that using the agent paradigm is ideal for engineering complex systems. This is because multi-agent systems can be modelled as a series of hierarchical organisational relationships that can be decomposed at different levels of abstraction into distributed, interacting, autonomous agents. This makes it possible for the organisation to be modelled and reasoned about at both the individual agent level and at the organisational level. Agents also retain their own thread of control. Removing the need for central control and replacing it with the distributed control of agents can reduce the complexity of the system. Having agents in control of their own interactions means that the low-level interactions that take place do not have to be predicted at design time. The agents can manage the coordination of their knowledge and actions.

An example of a multi-agent system being used to solve a complex problem is described in Neagu et al. (2006). The system must find the most efficient way to transport goods from storage to different delivery addresses using several vehicles. Each of the vehicles is modelled as an agent and allowed to negotiate with one another based on the cost of each agent to make each delivery. This allows the agents to plan their routes and the system to dynamically adapt to changes in the environment or customer needs.

Multi-agent systems are ideal for building complex systems because of the flexibility of their interactions (Jennings (2000)). The properties of multi-agent systems that make them ideal for complex problem solving also make the actions of the system unpredictable. This unpredictability is undesirable when the system has critical objectives (Perraju (1999)). Zambonelli and Omicini (2004) argue that because of the possible unpredictability multi-agent systems have to be engineered using rigorous methodologies in order to direct their behaviour. The complexity of verifying agent behaviour, that an agent will satisfy its task, is also recognised by Wooldridge and Dunne (2001), amongst others, who consider formal verification and model checking as techniques that need to be applied to multi-agent systems to overcome this complexity.

To be able to model and verify multi-agent systems using Event-B the concepts that need to be modelled must be identified. For this purpose the following section surveys a selection of AOSE methodologies.

3.2 Survey of Agent-Oriented Software Engineering Methodologies

Software engineering methodologies cover several phases of software development. Three of these phases are requirements analysis, design and implementation. Requirements analysis creates models of the problem domain and results in the specification of the tasks that the system must perform in order to solve the problem. The design phase takes the results of the analysis and models a system that will fulfill the requirements. The implementation phase is when the software is written that complies with the design of the system. Other software engineering phases include testing and maintenance (Pressman (2000)).

AOSE is the practise of constructing software systems using the agent metaphor. For agent technology to be used and used successfully the agent-oriented abstractions need to be engineered in the same way that traditional software is engineered (Zambonelli and Omicini (2004)). AOSE design methodologies range from those that offer full coverage of the software development process to those that only provide a high-level design. Most extend existing design methodologies (Sudeikat et al. (2004)). The methodologies reviewed below include some informal as well as formal methodologies and are intended to

provide a sample of those available. Some knowledge of the Unified Modelling Language (UML) is assumed.

The purpose of this literature review is to learn from the experience of methodologies that have been developed for AOSE. Several methodologies have been reviewed. The main concentration is on methodologies that use formal methods. Informal methodologies have been included to create a balanced view. An evaluation of the literature reviewed will examine the concepts that are needed to model multi-agent systems, how the complexity of modelling multi-agent systems can be overcome and how formal methods can be used in AOSE.

This review is structured as follows: First the methodologies are described. Then the evaluation will be introduced and the methodologies compared for each part of the evaluation. Finally the results will be discussed in terms of the current research.

Gaia

Gaia (Wooldridge et al. (2000)) is a methodology specifically designed for AOSE. It starts with an analysis of the system to be developed. The system is modelled as an organisation and broken down into agent roles and interactions. The roles are then assigned their responsibilities and the permissions needed to perform each particular role. The interactions are assigned protocols that allow the agents to perform the interactions. The design phase follows on from the analysis phase and produces three models. The agent model defines the agent types used in the system. These types are built from one or more of the roles identified during the analysis phase. The services model is derived from the responsibilities and protocols identified for each role during the analysis phase. The services are described abstractly as their inputs, output and pre- and post-conditions. The acquaintance model models the communication links between the agent types. The final models produced by the design phase are intended to be further developed into an implementable design using traditional design methods,including object-oriented techniques. Gaia has been extended to make it suitable for the design of open multi-agent systems using the coordination of communications to enforce the social laws of the system on heterogeneous agents (Zambonelli et al. (2000)). Other additions to the methodology include the introduction of organisational rules to the analysis stage and the organisational structure to the design phase (Zambonelli et al. (2003)).

AUML

AUML (Bauer (2001); Bauer et al. (2001)) extends the UML notation used for the design of object-oriented systems. It takes some parts of the UML and adapts them to the design of agent systems as well as adding new concepts. The specification of the

UML2.0 standard includes changes that are useful in the design of agent systems (Bauer and Odell (2005)). Use case diagrams provide a tool for analysis of the system and agents can be modelled as actors internal or external to the system. The design phase in AUML is centred on the agent class. Agent class diagrams show the dependencies between agents as associations between the classes. Abstractly the agent class models the roles required in the system and in the more concrete models the agent class models the agents that perform the roles. An agent class is composed of the agents head, body and communicator. The head models the state of the agent and its goals, the body models the actions of the agent and the communicator models the messages that the agent can send and receive. The agent is controlled by the agent head, which creates and responds to communications by triggering actions. The behaviour of an agent is modelled using either sequence or collaboration diagrams showing actions triggered pro-actively, in response to state changes, or reactively, in response to communications from other agents. Plans are modelled using activity diagrams. The organisation structure of multi-agent systems can be modelled using UML2.0 Composite Structure diagrams.

MAS-CommonKADS

MAS-CommonKADS (Iglesias et al. (1997)) is a methodology for the development of multi-agent systems derived from the CommonKADS methodology for the development of knowledge-based systems. This makes it particularly appropriate for modelling agent reasoning. The main adaptation is the introduction of inter-agent coordination into the design. The first phase of the methodology is called the conceptualisation phase. The result of this phase is a set of use case diagrams outlining the functions of the system and message sequence charts showing high level coordination of the identified system components. The analysis phase of the methodology is used to determine the more detailed requirements of the system. The agents identified in the use cases are assembled as an agent model. The tasks that the agents will carry out are modelled in the task model. The static relationships between the agents and between the agents and other objects in the system are modelled in the organisation model. The interactions between the agents, along with their required capabilities and the protocols used, are modelled in the coordination model. The message sequence charts from the conceptualisation phase are expanded into agent sequence diagrams. The expertise model models both the knowledge of the agents and the methods used to apply the knowledge to the agent's tasks. The design phase refines the models to represent the parts of the system dependent on the development platform. The architecture of the identified agents is modelled and decomposed into modules that support the required functions. The infrastructure of the system is modelled to include the agent relationships and the required supporting infrastructure for the system, such as directory and naming services. The design is refined so it can be implemented on the required hardware and software systems.

SMART

The Z formal language has been used to create the SMART framework (Luck and d'Inverno (1995a,b)). The framework is intended to formally define agent concepts and can be used as a starting point for the design of a multi-agent system using Z. It is built from specifications from a taxonomy of agent definitions. The specifications begin with the definition of an object and, through schema inclusion, builds definitions for agents and autonomous agents. Objects are entities in an environment that are capable of actions on the environment. Agents are objects that perceive the environment and have a purpose. This purpose is modelled as goals that the agent can have or be attributed by another agent. Autonomous agents are agents that have motivations that drive them to achieve, create or destroy goals. The SMART framework has been further extended to define interaction between agents in a multi-agent system. It has been used to create a formal specification of the AgentSpeak(L) language to provide a formal model of an implementable language (d'Inverno et al. (1998)). The SMART framework has been extended to provide a more detailed definition of autonomy (Luck and d'Inverno (1997)), goal generation and inter-agent relationships (Luck and d'Inverno (2006)).

Concurrent METATEM

Concurrent METATEM (Fisher (1993)) is an executable formal language based on temporal logic. It extends the programming language METATEM (Fisher (2006)) to allow the specification of concurrent multi-agent systems. Agents are modelled as a specification of their inputs and outputs and a set of rules. The rules consist of a pre-condition that models a state of the agent, this can be a belief, and a goal or action. The pre-condition implies the goal e.g. $complete\ project \Rightarrow receive\ bonus$. An operational model specifies how the concurrently executing agents communicate. The specification is directly executable. This allows the models to be reasoned about and then executed. The specifications can be verified through model checking or formal verification. The agent specifications can be refined and single agents can be refined into groups of agents. Extensions to Concurrent METATEM allow more complex multi-agent systems to be specified (Fisher (2005)). The temporal logic specifications can be extended to include the logics of belief or knowledge to specify the information that the agents will reason about. The specifications of multi-agent systems can be structured by grouping the agents in the system.

DESIRE

DESIRE is a method originally developed for formally specifying knowledge-based complex reasoning systems that has been extended for the development of multi-agent systems (Brazier et al. (1995, 1997, 1998)). The method focuses on the reasoning capabilities of the agents. The analysis phase results in a requirements specification for the system. There are two design strategies available. The tasks that the system is required to perform can be modelled and then delegated to agents, or the agents can be identified followed by their processes. The tasks, processes and agent knowledge are modelled as components connected by their inputs and outputs. The design is modelled, using the same modelling elements, at differing levels of abstraction. From the perspective of the overall system down to the components that construct the system's agents. The modelling is done both informally, using diagrams, and formally, using temporal logic-based specifications. The formal models can be verified for correctness and the components can be re-used.

Tropos

Tropos (Bresciani et al. (2004)) is a methodology that covers the development life cycle from requirements engineering to system implementation. The requirements of the system are informally modelled using a notation taken from agent-based requirements analysis. The current system, or the real world, is modelled and then the interaction between the 'system-to-be' and the real world is modelled. Formal models are used alongside the informal during the requirements phase as a way to verify the models (Perini et al. (2004)). Formal Tropos, a temporal logic-based formal language (Fuxman et al. (2003)), is used to model check and animate the requirements models. In the design phase the architectural design takes the idea of the 'system-to-be' and models that in terms of actors, goals, resources, tasks and plans. The detailed design models the system considering the details of the implementation platform. An example of development with Tropos uses AUML to model agent interactions, capabilities and plans in the design phase (Bresciani et al. (2004)).

3.3 Evaluation of AOSE Methodologies

Exploring and analysing current AOSE methodologies will show not only how agent theories have been applied to the area of software engineering, but also the techniques used to overcome the difficulties of creating a practical design method based on agent theories.

The evaluation below examines the significance the methodologies place on the different agent concepts, the methods used for dealing with the complexity of modelling multi-agent systems, and the formality of the method used. The agent concepts will show what needs to be modelled. The complexity management and the formality will show how they should be modelled.

3.3.1 Multi-Agent System Concepts

Identifying the concepts that are central to current AOSE methodologies will show what elements need to be used to develop a multi-agent system in Event-B. It will also show how they can be modelled by examining how each concept has been modelled in the different methodologies. The selection of the concepts is based on the modelling elements of the evaluated methodologies that we consider to be fundamental to the construction of the models. Other concepts are present in the methodologies, but these either hold less importance in the models or can be categorised under the selected concepts.

The reviewed AOSE methodologies can highlight the concepts that are required to develop multi-agent systems (Zambonelli and Omicini (2004)). The concepts that have been identified as fundamental to the methodologies are: organisations, goals, roles, interaction protocols, tasks and plans. The organisation of a multi-agent system is its structure (Jennings (1993)). Goals describe the motivations of an agent (Ferber (1999)). A role is a set of activities that an agent can carry out as part of the multi-agent organisation (Ferber (1999)). Interaction protocols are rules that define the messages that are used for agent coordination and the order in which they are communicated (Huget and Koning (2003)). Tasks are small functional units of work allocated to a role that are undertaken in order to complete an activity that can achieve a goal (Iglesias et al. (1997)). A plan is an order of tasks or activities that can be used to achieve a goal (Ferber (1999)).

The most common of these concepts and the techniques used to model them will be examined in more detail later in this chapter. These concepts will be considered for use in a methodology for the development of multi-agent systems in Event-B. To be able to model multi-agent systems in Event-B methods must be used to manage the complexity of the models.

3.3.2 Complexity Management

Two well known methods for dealing with complexity in software development are decomposition and abstraction (Booch (1991)). Decomposition is when a system is divided into smaller parts that can then be refined independently of the system. This allows the developer to understand how the system works as a whole and then concentrate

on the detailed workings of individual components. Abstraction is the act of viewing only the characteristics of a system that are considered essential for the context appropriate to the observer. This allows the developer to reason about the behaviour of a system without having to worry about the implementation details. Abstraction and decomposition work well together. Systems can be decomposed at different levels of abstraction. The evaluation below looks at the specific use of these techniques in the AOSE methodologies.

Gaia, MAS-CommonKADS and Tropos all provide abstract models for the first stages of development and have clear relationships between the abstract models and their more concrete counterparts. MAS-CommonKADS and Tropos models develop to quite concrete final designs. AUML offers a reasonable level of abstraction with UML use case diagrams and moves to quite a concrete design. Combining Gaia with AUML can create a methodology with a good level of abstraction for multi-agent systems and a concrete, object-oriented, final design. DESIRE and Concurrent METATEM both allow the developer to choose the level of abstraction at which to design their models. The compositional approach of DESIRE means that a clear relationship can be built between abstract models and the more concrete models of components. The formal syntax of Concurrent METATEM allows a formal refinement relationship to be built between abstract and concrete models. The SMART framework chooses a low level of abstraction for the starting point of the framework and moves toward a higher level of abstraction. The existence of the framework allows the developer to start a slightly higher level. The use of schema inclusion does reduce the complexity of the specification by hiding the details of the lower levels of the system in the included schemas. The schema inclusion relationship in Z is well defined (Diller (1994)).

In multi-agent system development decomposition can be used to decompose the system into a group of interacting agents, as well as individual agents. Gaia and AUML decompose the system into roles that can then be assigned to agents. Tropos uses decomposition during the requirements phase to decompose the system into actors, goals and plans that are later assigned to agents. DESIRE composes a multi-agent system from the tasks or processes and knowledge of the system. Concurrent METATEM uses decomposition to decompose a multi-agent system into agents. The SMART Framework uses schema inclusion to compose the system. Each component must be modelled with the intention to compose it into the system. Decomposition is used on the individual agents as well as the system. In AUML each agent is composed of a communicator, head and body.

3.3.3 Formality

All of the reviewed methodologies that use formal methods do so in different ways. DESIRE uses formal methods for their traditional purpose of verifying the correctness

Table 3.1: Central Concept Comparison

methodology	goal	role	task	organisation	interaction protocol	plan
gaia		✓		✓	✓	
AUML	✓	✓			✓	✓
MAS CommonKADS			✓	✓	✓	
SMART	✓				✓	
Concurrent METATEM	✓					
DESIRE			✓			
Tropos	✓	✓				✓

of the formal models and then feeds the results of this analysis back into the informal models. Tropos limits their use to the requirements phase of the development to ensure that the requirements are consistent and unambiguous. Concurrent METATEM exploits the executable nature of the temporal logics used to create directly executable specifications. SMART uses the Z formal language to ensure that the agents developed are consistent with its underlying agent theory. Tropos and DESIRE both use formal methods alongside informal diagrams in order to make the models more accessible to those who do not have formal method experience.

3.4 Modelling Techniques for Multi-Agent System Concepts

Table 3.1 shows that of the concepts used in the reviewed methodologies goals, roles and interaction protocols are the most common, alongside that of agent. The methodologies that have their roots in knowledge-based systems use the concept of tasks. Whereas, those based directly on agent theories focus more on goals. The techniques used to represent the three concepts in the AOSE methodologies are examined in detail below. The knowledge gained from this examination will help to find a technique for modelling the different concepts.

3.4.1 Agent Interaction and Coordination

Multi-agent systems are a way of breaking down a complex and distributed problem into smaller problems that can be solved by distributed software agents. For agents to be able to cooperate to solve a problem they must be able to coordinate their knowledge and actions (Jennings (1993)). Coordination is also needed when agents compete for resources.

Agents can coordinate their knowledge by informing one another of their beliefs and knowledge. Actions can be coordinated either through instruction or negotiation. This coordination can be achieved by communicating with messages (Ferber (1999)).

Agent interaction protocols are a method of describing the messages that can be sent between agents and the ordering of those messages. They can be considered to be a convention of a system that governs the social interactions of the agents (Jennings (1993)).

The Foundation for Intelligent Physical Agents (FIPA) have published specifications for several interaction protocols. These include *request* (FIPA (2002e)), *query* (FIPA (2002d)), *brokering* (FIPA (2002b)), and *contract net* (FIPA (2002c)). The protocols are informally specified using UML interaction sequence diagrams as the notation. This informal specification is ambiguous (Paurobally et al. (2005)) and this could lead to agents complying with the protocol, but unable to coordinate.

Agent Communication Languages (ACL) allow agents to coordinate their actions and knowledge through a shared language that can be passed between agents in messages. Programming languages, such as Prolog, can be used for this purpose (Papadopoulos (2000)). However, the languages specifically designed for inter-agent communication provide a more tailored approach. A standardised ACL, like FIPA ACL (FIPA (2002a)), offer an expressive syntax for interoperability between heterogeneous agents.

Interaction protocols are a concept used in four of the reviewed methodologies. In the analysis phase of Gaia interactions are modelled between roles (Wooldridge et al. (2000)). Protocols for interaction are defined abstractly based on the purpose of the interaction, the participants in the interaction and the information passed between the participants. In the design phase the interactions between agents are expressed by their relationships in an acquaintance model. Gaia does not produce designs at the level of detail that will prescribe the messages that are sent between the agents.

Interaction protocols are modelled in AUML using an interaction protocol diagram that is an extension of a UML sequence diagram (Bauer et al. (2001)). The order and content of the messages passed between the participating agent roles is defined as part of the protocol. The lifelines of the agent roles can be split using AND, OR and XOR relationships to show the different possible reactions to the receipt of messages.

The interaction between agents in the conceptualisation phase of MAS-CommonKADS is modelled using message sequence charts (Iglesias et al. (1997)). These models are then refined in the coordination model to include the information passed between the agents and the performatives of the messages by introducing models of the states and events of the interaction. Interaction protocols can then be identified for the interaction. Interaction protocols are modelled using message sequence charts and can be re-used.

The SMART framework specifies a set of Z schemas that model the interactions between agents in a multi-agent system (d'Inverno and Luck (1997)). For agents to interact there must be a goal that needs to be fulfilled. Agents are engaged by, or cooperate with, the agent that generated the goal. The engaged, or cooperating, agent can then adopt the goal. The framework includes specifications of the different possible relationships between agents. The structure of the framework can then be used to construct specifications of interaction protocols.

Gaia and AUML base the possible interactions between the agents on the roles that the agents play in the system. SMART bases the interactions on the shared goals of the agents. The information exchanged by the messages is modelled by the methodologies and MAS-CommonKADS designs include the changes in state caused by the interaction. None of the methodologies prescribe an ACL for the message content. All of the methodologies intend that the interaction protocols can be re-used.

3.4.2 Agent Goals

A goal for an agent can be described as a desire that an agent believes to be achievable (Rao and Georgeff (1991)). The evaluation of the AOSE methods identified that agent goals are key concepts in AUML, the SMART framework, Tropos and Concurrent METATEM. Other methodologies implicitly use goals as a concept. For example, Gaia models how societies of agents cooperate to fulfill system-level goals (Wooldridge et al. (2000)). As a key concept the use of goals in the identified AOSE methodologies needs to be further examined.

The goals of an agent in AUML are defined for individual agent roles (Bauer (2001)). The state description field in the agent head can be used to define a logical description of states. These states can be beliefs or goals or other instance variables. How the agent behaves is defined by these states. The agent's reactive and pro-active behaviours are specified using UML diagrams, e.g. sequence and state diagrams. Using these diagrams the receipt of a message can change the beliefs of an agent, that can lead to a new goal and that, in turn, can lead to a message being sent in reply. Goals at a system level are implicit in AUML and can be modelled as the final activity of an activity diagram that models the agent roles or the services provided by the agents (Bauer and Odell (2005)).

In Concurrent METATEM the goals of the agents are expressed as predicates in temporal logic e.g. $\diamond rich$. Agents are specified with a set of rules where the state of an agent, its belief or knowledge, will imply the satisfaction of the goal e.g. $\diamond win\ lottery \Rightarrow \diamond rich$.

In the SMART framework goals are described as a desirable state of affairs (Luck and d'Inverno (2006)). A distinction is made between goals and motivations. Motivations are desires that lead to the generation or adoption of goals. Tasks are also defined as a state of affairs to be achieved and are therefore the same as goals (Luck and d'Inverno

(1997)). Agents are specified as objects with a non-empty set of goal attributes and autonomous agents are agents with motivations. The agent behaviour is defined by a function that that specifies the actions taken according to the agents motivations, goals and the state of the environment. Shared goals are used to create inter-agent cooperative relationships.

Goals are concepts used in all phases of software development using the Tropos methodology (Bresciani et al. (2004)). They are defined in the Tropos meta model and are described as an actor's, or agent's, strategic interests. Goals are first introduced in early-phase requirements engineering using the i* method to produce relationships between actors and goals. This can create dependencies with other actors. The goals and dependencies shared by actors are decomposed to construct the architecture of the system. The agents identified in this architecture have their goals specified in detail during the detailed design phase.

The Tropos methodology uses the i* method for Goal Oriented Requirements Engineering. There are several other methods that use goals as a key concept for analysing system requirements. Goal Oriented Requirements Engineering and Agent Oriented Requirements Engineering use goals to describe a system in a way that can be understood by stakeholders (Lamsweerde (2001)). The remainder of this section will review methods for Goal Oriented Requirements Engineering. This review will provide further information about how goals can be used in software engineering.

The methods described below distinguish between soft and hard goals (Bresciani and Donzelli (2003)). A soft goal is a goal whose fulfillment is difficult to define. It may be that the agent who originates the goal decides that it has been fulfilled. Hard goals are goals whose fulfillment is easily defined.

In the KAOS method of Goal Oriented Requirements Engineering goals are statements of intent (Letier and van Lamsweerde (2002)). The method specifies agents and goals. The agents of the system need to be able to cooperate to achieve the goals. The goals and agents are decomposed until the goals can be assigned to individual agents. The goals that are assigned to the agents in the system become the requirements for the system. Agents are considered to be state transition systems and the operations of the agents operationalise the agent's goals and move the agent between states.

i* (Yu (2002)) is a requirements engineering method used in the Tropos methodology. Goals are used to model dependencies between agents. This models the social aspects of multi-agent systems as well as the intentional aspects. The goals and agents are decomposed until the goals can be assigned to individual agents. This creates an architecture of the system, in terms of agents, that can be described by the reasons for the architecture, in terms of goals.

REF is intended to be a simpler modelling framework than i* (Bresciani and Donzelli (2003)). Agents are used to model the organisation of the system and the goals define the relationships between the agents. Tasks, resources and constraints are included in the goal diagrams to provide more detail about the system architecture.

The methods examined above treat goals as states that the system moves between that define an agent's behaviour and trigger actions. The goals create dependency relationships between agents and this can be used to create a system architecture.

3.4.3 Agent Roles

The role of an agent defines their capabilities and behaviour within a system and is the basis on which an agent interacts with a group of agents. An agent in an auction will interact with the other agents by taking the role of buyer or seller. The role of an agent is a label for its recurring dependencies and interactions (Parunak and Odell (2002)).

An agent's role is also a way of grouping agents. An agent's role can describe a set of distinguished properties or behaviour that are common to the agents performing that role (Bauer (2001)).

Agents can perform several different roles and a system can contain several agents performing the same role. In an agent marketplace an agent may be buying goods from one agent and then selling them on to another, and so must be capable of performing both the role of buyer and of seller.

In the Gaia methodology roles are abstract concepts that are used to analyse the system (Wooldridge et al. (2000)). The analysis phase of Gaia begins with the identification of the roles of the system and the organisation of the system is viewed as the roles and their relationships. A role has the four attributes of responsibilities, permissions, activities and protocols. The responsibilities describe the functionality required by the role in terms of liveness and safety properties. The permissions describe the role's access to resources. The activities are the private internal actions that need to be carried out to perform the role. The protocols attribute identifies the interactions protocols that the role can use to communicate. During the design phase the agents of the system are identified as a set of one or more roles.

In AUML roles are described by single agent classes, with attributes and operations, at the conceptual level of modelling (Bauer (2001)). At the specification level a collection of one or more roles is assigned to an agent class. The roles can then be carried out by agents that are instances of the agent class.

In the Tropos methodology the actors identified in the early requirements analysis can be roles, positions or agents (Bresciani et al. (2004)). The abstract actor is a role where the agent playing the role does not need to be identified in the model. Positions are sets

of roles that can be assigned to agents at a more concrete level. An actor is an agent when the model is concrete enough to identify an individual that will fulfill the abstract position (Yu and Mylopoulos (1994)).

In all three of these methodologies roles are used as abstract concepts for agents. Roles are modelled as a collection of states and actions. The more concrete models of agents can perform one or more of the roles identified at the abstract level.

3.5 The Use of the B-Method and Event-B in AOSE

AOSE methodologies have been examined to show the different possible approaches to modelling multi-agent systems. The evaluation highlighted the techniques used. A further review of literature on modelling multi-agent systems in the B-Method and Event-B should show how some of these techniques have been used and other techniques that may be useful.

Fadil and Koning (2005) use the B-Method to model a multi-agent system. There is a specification of the system as a whole, at a single level of abstraction, with operations to add and remove agents. This is the specification of a platform not of an actual problem and can, therefore, be generic. The agents are specified individually and interact through their roles. The agent machine includes the role machine. The role machine, in turn, includes a machine that models the interaction protocol being used. The agent machine starts the interaction by calling the operations in the role machine and then the role machine will call the operations in the protocol machine. The protocol machine is responsible for making sure that the protocol is complied with. This architecture is presented in a practical case study of a travel system. There does not appear to be a relationship between the multi-agent system and the agents that construct it. This is perhaps necessary as the use of B and the operation style would imply a centralised thread of control where the operations will be called by the system. The authors intend to refine the specifications further.

Event-B has been used to model a contract net case study in Gao et al. (2007). The system is an abstract machine based on agents and their roles. The protocol machine models the exchange of messages by events that are triggered by states controlled by the agent roles. The contract net machine is adapted for a system for designing automobiles. The ‘includes’ clause from the B-Method is used to compose the agent machine into the role machine, the role machine into the protocol machine and the protocol machine into system machine. The authors have refined the model, but offer no examples or method for refinement in the paper.

Event-B has been used to model heterogeneous mobile agents at different locations on a network (Iliasov et al. (2005, 2006)). The purpose of this is to provide abstract models

where the interoperability of the mobile agents is ensured. The abstract specification models the agents in a system called a location. The location provides the services that allow mobile agents to leave and join the location. During several refinements the model is enhanced to provide fault-tolerance mechanisms and to allow agents to coordinate with the other agents that are present at the location. One of the main concepts used in the models is that of scope. The scope is an abstraction that formalises how interactions can take place. An agent must join or create a scope to coordinate with other agents. The agents in a scope communicate through a private tuple space. In order to join a scope an agent must be capable of performing a required role. Any agents that are incompatible with the requirements of that role need not be considered. The overall system can, therefore, operate as an open system, whilst ignoring inter-agent compatibility issues. This method puts the responsibility for the services needed for coordination onto the platform rather than the individual agents. The specification concentrates on modelling the agent platform. A focus of the research is to provide fault-tolerance as part of the agent platform.

3.6 Summary

The literature review in this chapter describes and evaluates a selection of AOSE methodologies that use both formal and informal methods. The techniques used to create models of multi-agent systems have been examined in more detail. The evaluation shows the concepts that are used in the models of multi-agent systems, how formal methods are used and how the complexity of the models are managed.

When it comes to modelling multi-agent systems agent interactions are an important concept to model. Without interaction between the agents they will be unable to coordinate their information and actions, and function as a system. Interaction protocols model the messages and the order that messages are exchanged to fulfill an activity within a multi-agent system. Interaction protocols are modelled in the reviewed AOSE methodologies as sequence diagrams that show the order for the exchange of messages. They are modelled as part of the roles that are assigned to agents.

Goals represent the motivations of the agents in the system. The agents will collaborate to perform shared goals. Without goals the agents would not act autonomously or proactively. Goals are modelled in the AOSE methodologies as the link between the state of the agents and the actions that they perform.

Roles are an abstract representation of agents. Agents can perform one or more roles. Roles are modelled in the reviewed AOSE methodologies as a set of states and actions.

Abstraction and decomposition are two appropriate techniques for coping with the complexity of modelling multi-agent systems. Decomposition allows models of systems to be

broken down into their components whilst maintaining a relationship with the original model.

Several methodologies for the development of multi-agent systems use formal methods in their approach. The reasons for their use range from creating consistent and unambiguous requirements to being able to execute the models. Developing informal diagrams alongside formal ones can make formal development more accessible.

The reviewed uses of the B-Method and Event-B in modelling multi-agent systems show how the concepts that are needed in a multi-agent system can be modelled at an abstract level. Roles are used to specify the activities of the agents and the interactions of the agents are controlled by the role. The interactions of agents are modelled abstractly and refinement can be used to create detailed models of agents that will perform the interactions as specified in the abstract models.

Chapter 4

Incremental Development Process: Stage One - Goal Elaboration

This chapter describes Stage One of the Incremental Development Process. The chapter begins with a description of how the concepts identified in the evaluation of AOSE methodologies have been used in the Incremental Development Process. The goal diagrams and goal relationships used in the Incremental Development Process are introduced. Guidance is given for translating the diagrams for Stage One into Event-B models.

The development of a model in Event-B can be described as a process that begins with the abstraction of the system being modelled and continues through several refinements that add detail to the abstract model. The developer will iterate the process as more is learnt about the model through its development. This thesis presents a process for the development of multi-agent systems in Event-B that has been captured as two stages. The first stage models the system requirements from the system perspective as goals and refines these goals to model the interactions of the system. The goal model guides the development of associated Event-B models that formally specify the required system functionality. The second stage takes the goal model developed in the first stage and adds communications that are required for the system to be modelled as the interactions of agents and allocates the communications to the agent roles that can be identified for the system. This model is then used to guide the further refinement of the Event-B system model into component models of the system's agent roles.

For conciseness the Incremental Development Process will be referred to as the Process.

The Process is general and has been applied to two case studies of increasing complexity. The Process begins with the high-level requirements of a multi-agent system and

finishes by producing abstract Event-B models of the system components. Each of these components can be refined and are open to possible re-use.

This chapter begins by explaining the rationale behind the choices made when creating the Process. Stage One of the Process is then described.

4.1 Using the Multi-Agent System Concepts

The Process has been developed to use the three most prevalent agent concepts identified in Chapter 3 as the basis for the conceptualisation of multi-agent systems. This creates a method for developing multi-agent systems that is centred on fundamental agent properties.

Several of the AOSE methodologies described in Chapter 3 construct different models of the system based on the different agent concepts. Integrating all of the aspects into one model makes it possible to reason about all of the aspects of the system and how they affect one another rather than considering each separately and assuming that they will not conflict. Producing three different formal models and verifying them separately would also create a greater burden for the developer. Therefore, the three main concepts of agent goals, agent interaction and agent roles need to be combined in the one formal model. At the system level the goals of an agent are only visible by the actions that they take. In an Event-B model the actions of the system are modelled as events. The fulfillment of each goal can be modelled as an event in the Event-B model. An Event-B model is a state transition system and can model the interaction of a distributed system as the transition between states with the triggering of each event moving the interaction to the next state. The role of each agent defines their goals and the actions available to them. A role in an Event-B model can be defined by the decomposition of the model into groups of states and events.

The Process needs a starting point and the system can be modelled from the perspective of any of the agent concepts. However, to start the process from the system requirements requires the first conceptualisation of the system to use an appropriate model. Goals are modelled in software development as part of the early phases of requirements engineering. The goals can be modelled alongside agents that at this abstract level of system development could be implemented by hardware, software or human intervention (Yu (1997)). The goal diagram from Goal-Oriented Requirements Engineering can be used as a starting point for the model of the system. This allows the system to be modelled informally and the other concepts can then be constructed from the identified goals. Goal diagrams are considered to be accessible to the different stakeholders because they capture the rationale behind the model (Lamsweerde (2001)). Using a model that can be integrated with models from early-phase requirements engineering will help to address the transition between later phases of requirements engineering and

the translation of the requirements into formal models of multi-agent systems. Having models of the requirements based on an intentional perspective allows this perspective to be captured in the translation to the formal models. The formal models can then be analysed and the results can be fed back into the requirements analysis as well as refined to a verified concrete design of the system.

The goals of the agents are introduced by the Process by firstly modelling the system as a set of goals and then refining the goals so they can be allocated to the different agent roles required for the system. These goals are then related to events and variables in an Event-B specification of the system. This approach ensures that the individual agent goals conform with the overall system goals and that the combined functioning of each agent will embody the overall functionality of the system.

The relationships used in the goal model can be used to describe the interaction required to fulfill the goals of the system and fit a state transition system that can be used to form an Event-B model. Each transition will occur as agents take actions to fulfill their goals. The system level behaviour is modelled before the individual agent behaviour to ensure that the individual agents will work together as a multi-agent system.

The Process models the shared behaviour of the system and then decomposes the system into the agent roles and the system environment. The Process is also designed to model each agent with an individual thread of control by encapsulating the state and actions of each agent. The guarded events of the Event-B method model each action as being executed atomically. The actions of an agent will not interfere with the actions of the other agents in the system. The final decomposition of the system makes concrete this separation of processing in the model.

Using Event-B as the modelling language creates models of reactive systems that include their environment that influences, and is influenced by, the agent behaviour. The guarded events in Event-B can be triggered when the state of the system changes, due to the action of another event.

4.2 Stage One

The Incremental Development Process is separated into two stages. In this section Stage One will be described. The purpose of Stage One is to model the system as a set of goals that describe the interactions of the system. Event-B models are created by analysing the goals and relationships used to construct models of the system goals. The perspective of the Event-B models in Stage One are of a single system of interacting agents that changes state as the agents interact.

Stage Two continues the elaboration of the goal diagram and the refinement of the Event-B model. The perspective of the model changes as communications are introduced to

model the interacting agents as separate entities. Stage Two results in specifications of the roles of the agents in the system.

Modelling of systems in Event-B begins with the specification of the abstract requirements of the system that can then be refined by adding more detail to reflect the functioning system. Specifying the abstract requirements of a system in an Event-B model is not an easy task. Having additional models to help the developer discover a useful level of abstraction at which to express the system requirements will make the task more accessible. Refining the system requirements to a model of a functioning system requires many modelling decisions and several iterations of those decisions. Capturing the decisions and then elaborating those decisions independently of the formal models in a uniform manner should ease the burden on the developer, especially a developer with limited experience of formal methods.

The rest of this section describes how the goal diagrams are constructed and how these informal models of multi-agent systems can be captured in formal system models in Event-B.

4.2.1 The Goal Diagram

A single goal is chosen as the starting point for the goal model. This is because it is the elaboration relationships that describe the interactions of the system. A single Event-B model will be constructed from these elaboration relationships to model the system interactions.

The root goal can be ascertained from the name of the system or the highest level requirement of the system. The fulfillment of this goal will solve the problem that the system is being designed to solve. This goal can be elaborated by deciding what lower level goals need to be fulfilled in order to fulfill the higher level goal. The goals are then elaborated until the discerned goals can be allocated to the individual agent roles that can be identified for the system.

For simplicity, only one model has been chosen to be translated into Event-B, the goal model. A developer may wish to use other informal models, including other agent and goal models that model the early-phase requirements of the system, that correspond to the goal model.

The elaboration of the goal diagram will create a tree structure with the goals related by four possible relationships. The relationships that are used will create either an order to, or a choice between, the goals. The ordering of the goals means that the goal diagram can be seen as the interactions of the system goals when read from left to right. This gives an abstract view of the state transitions of the system that can then be translated into an Event-B model.

The goal diagram gives a visual abstraction of the Event-B models that can then be referenced alongside the Event-B models to aid understanding as an informal representation of the formal models.

4.2.2 Goal Elaboration

The purpose of the goal elaboration is to refine the system goals into goals that can be assigned to individual agents. The refinement of goals will be referred to as goal elaboration to distinguish it from the refinement of the Event-B models. The goals are elaborated into one or more sub-goals. The fulfillment of a set of refining sub-goals will result in the fulfillment of the parent goal. An elaboration with a single sub-goal renames the parent goal. There are four different elaboration relationships available for Stage One of the Process. Each set of sub-goals will be related to one another with either a THEN, AND, or one of two different OR relationships.

The relationships used to refine the goals in the goal diagram help to construct a model of a system of interacting agents. Having the order also focuses the model on the concept of interaction between the agents in the system as each stage can be separated into the goals and the order in which they can occur. Having the OR relationships models the choices that the agents will make. Having the AND and THEN relationships models the dependencies between the goals and the different agents that are eventually allocated the goals.

The use of four relationships differs from other notations that use mainly AND and OR decomposition relationships (Bresciani and Donzelli (2003); Lamsweerde (2001)) or decomposition and dependency relationships (Yu (1997)). This is partly due to the translation between the goal diagram and the Event-B models. The Event-B model is a state transition model and having an order in the goal model, as provided by a THEN relationship, makes the translation more straightforward. The specification of the relationships in the formal model offers more precision and there is a significant difference in the behaviour of the model between the specification of an inclusive and exclusive OR relationship.

The THEN relationship between goals establishes an order in which the goals must be fulfilled. The left-hand goal must be fulfilled prior to the right-hand goal. The right-hand goal can only be fulfilled if the left-hand goal has previously been fulfilled. Both sub-goals must be fulfilled in order for the parent goal to be fulfilled. An example of a THEN goal elaboration is shown in Figure 4.1. In the goal diagram the abstract $goal1$ will be fulfilled if $goal1a$ is fulfilled followed by $goal1b$.

The AND relationship specifies that all of the related goals are required to be fulfilled for the parent goal to be fulfilled. The AND relationship does not specify an order for the fulfillment of the goals. An example of an AND goal elaboration is shown in Figure 4.2.

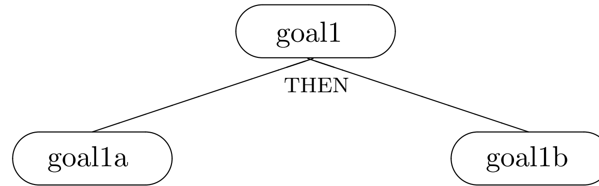

Figure 4.1: A THEN Elaboration

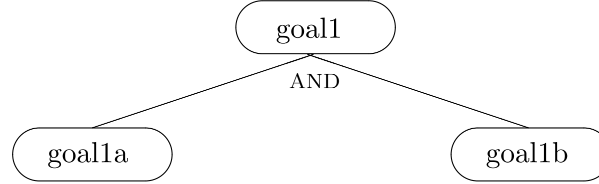

Figure 4.2: An AND Elaboration

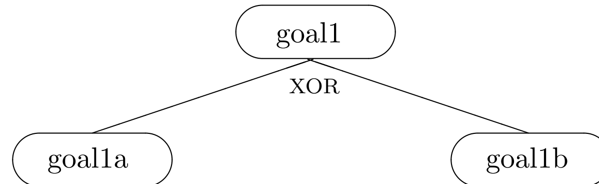

Figure 4.3: An XOR Elaboration

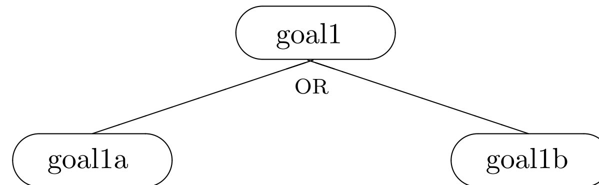

Figure 4.4: An OR Elaboration

In the goal diagram the abstract $goal1$ will be fulfilled if both $goal1a$ and $goal1b$ are fulfilled.

The XOR relationship creates an exclusive choice between the related goals. Either of the sub-goals can be fulfilled for the parent goal to be fulfilled and only one of them must be fulfilled in each interaction. Figure 4.3 shows a goal diagram with an XOR elaboration. In this goal diagram $goal1$ will be fulfilled if $goal1a$ is fulfilled or if $goal1b$ is fulfilled, but not both.

The OR relationship differs from the XOR relationship because it is not an exclusive choice. The OR relationship means that the fulfillment of either of the related goals is independent of the other. Figure 4.4 shows a goal diagram with an OR relationship: $goal1a\ OR\ goal1b$. Either $goal1a$ or $goal1b$ can be fulfilled and the fulfillment of one or both goals will lead to the fulfillment of the parent goal. The OR relationship is useful for keeping the relationships abstract at higher levels of goal elaboration. Using an XOR or THEN relationship may be too restrictive to describe how the sub-goals that are elaborated later in the goal model will interact.

An important concept for the goal model is that of a *completion condition*. The completion condition for a goal is the set of conditions that need to be satisfied for the goal to be fulfilled. The starting condition for a goal is of less importance in the goal

models. It is often the completion condition of a THEN-related goal. The completion and starting conditions for goals are similar to the concepts of preconditions and postconditions on operations. The postcondition of an operation will be satisfied as long as the precondition is upheld on the execution of the operation (Hoare (1969)).

The completion conditions of a set of goals can be used to distinguish the function of the different relationships. The THEN relationship, e.g. Figure 4.1, requires that the completion condition for any goals to the left of the THEN relationship, $goal1a$, to be the same as the starting condition for the goals to the right of the THEN relationship, $goal1b$. The set of goals will be fulfilled when the completion condition for the goals to the right of the THEN relationship, $goal1b$, has been reached. The AND relationship requires that the starting condition for each goal to be the same and the completion condition for the relationship is a conjunction of the completion conditions for the AND-related goals. An XOR relationship requires that the completion condition for each goal is disjoint from the starting condition for each goal. An OR relationship requires that the starting condition for each goal is the same and the completion condition for the relationship to be a disjunction of the completion conditions for the OR-related goals.

All of the goals in the goal diagram are linked through their relationships. Using goal elaboration does allow you to elaborate each goal in isolation and then examine how adding this detail can then affect the system. The system as whole cannot be understood by analysing a single goal and its elaboration tree.

Using Event-B as the specification language to model the goal elaboration has the advantage of being able to refine one event into multiple events and add new events to a refinement model. This allows the goal elaboration relationships from the goal diagrams to be directly translated to events in the formal models.

4.2.3 Constructing and Refining the Event-B Models

The goal diagram that has been constructed using the elaboration relationships described above can be used to help construct an Event-B refinement chain that creates an Event-B representation of the system. To keep the goal model simple the graphical notation used has been limited. This means that the guidance given for translation between the informal and formal models cannot be particularly systematic. Adding further notation may enable a more systematic translation, but this will limit the flexibility of the conceptual informal model.

Preserving the intentional perspective in Event-B can be achieved by having the events and state variables of the system based on the goals in the goal diagram. The translation between the goal model and Event-B model specifies that a change in the state variable changes which events can occur. This models a change in the motivational state of the system triggering an action.

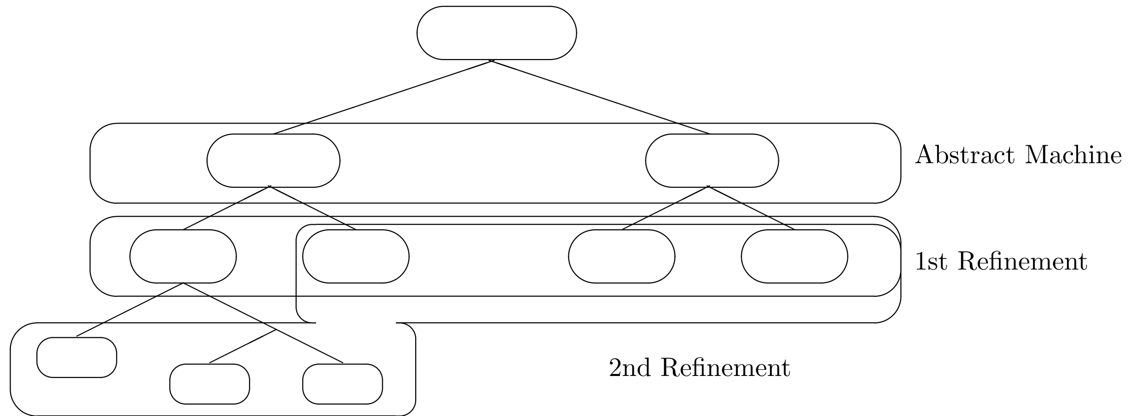

FIGURE 4.5: How Goal Elaboration Can Relate to Event-B Refinement

An Event-B abstract machine can be constructed using the first level of goal elaboration in the goal diagram. Each of the subsequent levels of goal elaboration can be used to help construct an Event-B refinement of the abstract machine. For each refinement the lowest nodes for each branch of the goal tree are included in the specification. Figure 4.5 shows how a goal tree can be related to an Event-B refinement chain.

The goal elaboration relationships in the goal diagram affect how the Event-B models are constructed. These elaboration relationships in the goal diagram cannot be analysed in isolation when constructing the Event-B models. The relationships at the higher levels of the goal tree will have an affect on how the relationships between the goals in the Event-B models from the lower levels of the goal tree are modelled. These relationships are preserved through the Event-B refinement relationship.

When using the goal diagram as a guide to construct the Event-B model each of the goals are modelled in two ways. They are modelled as events of the system that represent the goal being fulfilled. They are also modelled as states that the system moves into when the event occurs. Modelling the goals as states, as well as events, allows the order that is defined by the goal elaboration relationships to be enforced in the Event-B model.

The Event-B models represent a state transition system model of the interaction with the fulfillment of the goals, or triggering of the events, moving the system through the states of the interaction. Each interaction between a group of agents can be seen as a conversation. The conversation is the fundamental concept of each of the Event-B models constructed in the Process. The concept of a conversation comes from the FIPA ACL Message Structure Specification (FIPA (2002a)). It specifies that a conversation-id field should be used in each message to allow agents to distinguish between each set of interactions. It is a useful concept when modelling a state transition system of agent interaction as it allows each agent to be involved in more than one set of interactions concurrently, whilst controlling the progress of the individual conversations by their state. Each development should include a context that specifies deferred sets CONVERSATION and AGENT.

An example of how an Event-B model can be constructed from a single goal is shown in Figure 4.6. Variables and events are specified to represent the goals. The state variables for the goals are specified as relationships between conversations and agents. The relationship is modelled as a partial function, $\nrightarrow$, so that each conversation will be with one agent, $goalstate \in CONVERSATION \nrightarrow AGENT$. How this relationship changes for systems with interactions between several agents will be described below. The goal states are initialised as empty. The events representing the goals are parameterised by a conversation, c, and an agent, a, where the conversation is not in the goal state for that event, $c \notin dom(goalstate)$. The action of the event will add a relationship between the conversation and agent to the goal state, $goalstate := goalstate \cup \{c \mapsto a\}$.

```
VARIABLES goalstate
INVARIANTS
   goalstate ∈ CONVERSATION ⇸ AGENT

EVENTS
    INITIALISATION
          BEGIN
            goalstate := ∅
          END
    goalevent
          ANY  c, a  WHERE
            c ∈ CONVERSATION
            a ∈ AGENT
            c ∉ dom(goalstate)
          THEN
            goalstate := goalstate ∪ {c ↦ a}
          END
 END
```

FIGURE 4.6: Event-B Model for a Goal

The Event-B abstract machine is constructed from the first level of goal elaboration. The refinement of the abstract machine will continue through each further level of goal elaboration in the goal diagram. The goal relationships used to refine the goal model will direct how the refinement relationships in the Event-B models are specified. Event-B refinement will require that the relationships specified in the abstract machine are upheld in the refinement models. Using the goal models to guide the refinement will guide the construction of the refinement models that contain the gluing invariants and event guards to discharge the proof obligations generated by the Event-B refinement.

Fisher et al. (2007) identify properties that leading agent theories and formal methods used for the development of agent systems share. These are:

- an *informational* component, so as to be able to represent the agent's beliefs or knowledge;

- a *motivational* component, to represent the agent's desires or goals; and
- a *dynamic* component, allowing the representation of the agents activity.

These properties are modelled by the Process. The state of the Event-B models captures the informational component with each state representing the agent's knowledge about its place in a conversation. The events of the model capture the dynamic component through the actions that the agent is capable of performing. The motivational component is captured by the goal models. This abstraction is then translated into the Event-B models as part of its state.

Representing Goal Relationships in Event-B

The rest of this section will show how the goal elaboration relationships can be used to guide the development of both abstract machines and refinement models. How the use of Event-B restricts the use of some of the relationships will also be described.

A THEN elaboration relationship between two sub-goals creates an order in which the goals must be fulfilled. The Event-B model can reflect this by enforcing an order in which the events for the goals can be triggered. This can be achieved using the guards for the event and enforced by adding an invariant condition to the model.

When a goal in an abstract model is elaborated by two goals that form a THEN relationship the events and variables that are introduced for the goals must refine the abstract events and variables for the abstract goal. The refinement relationship will be between the state and event for the abstract goal and the state and event for the right-hand sub-goal. The event for the left-hand sub-goal will be introduced in the refinement as a new event.

Figure 4.7 shows an Event-B refinement model that models a THEN goal elaboration relationship as shown in Figure 4.1. This goal elaboration relationship can be syntactically described as $goal1 \sqsubseteq goal1a\ THEN\ goal1b$, or $goal1$ 'is refined by' $goal1a$ THEN $goal1b$. The two state variables that represent each of the goals, $goalstate1a$, $goalstate1b$, model the THEN relationship with one goal specified as a subset of the other, $goalstate1b \subseteq goalstate1a$. The events of the model, $goalevent1a$ and $goalevent1b$, uphold the THEN relationship. The first event, $goalevent1a$, is only enabled when the conversation is not in the state for goal1a. The action of the event adds a relationship between the conversation and agent to the state $goalstate1a$. The subset relationships between the states in the model mean that the relationship must be the same as the relationship added to the first state in the first event. Therefore, the agent in the relationship represents the agent with the goal that is fulfilled by the first event. The second event can only occur when the relationship is in the $goalstate1a$ state and not in the $goalstate1b$ state. The action of the event then adds the relationship to the $goalstate1b$ state.

The Event-B specification of the THEN relationship models the causal relationship between the goals in the THEN relationship. The guard, $c \mapsto a \in goalstate1a$, for `goalevent1b` means that the event can only occur if `goalevent1a` has already occurred and the model has moved the conversation into `goalstate1a`. This also ensures that if the state of the model were to be affected in a way that removed the conversation from `goalstate1a` before `goalevent1b` were to occur then `goalevent1b` would be prevented from occurring.

To refine a goal in the Event-B model with a THEN elaboration a gluing invariant is added to the model. The gluing invariant specifies that the variable $goalstate1b$ is equal to the state for the parent goal, $goalstate1 = goalstate1b$. The variable $goalstate1a$ is specified as a subset of `CONVERSATION` as it is not related to another goal in Figure 4.1. The $goalevent1b$ is specified as a refining event and $goalevent1a$ is specified as a new event.

```
VARIABLES goalstate1a, goalstate1b
INVARIANTS
   goalstate1a ∈ CONVERSATION ⇸ AGENT
   goalstate1b ⊆ goalstate1a
   goalstate1 = goalstate1b
EVENTS
    INITIALISATION
         BEGIN
           goalstate1a, goalstate1b := ∅
         END
      goalevent1a
         ANY  c, a  WHERE
           c ∈ CONVERSATION
           a ∈ AGENT
           c ∉ dom(goalstate1a)
         THEN
           goalstate1a := goalstate1a ∪ {c ↦ a}
         END
      goalevent1b  REFINES  goalevent1
         ANY  c, a  WHERE
           c ↦ a ∈ goalstate1a
           c ↦ a ∉ goalstate1b
         THEN
           goalstate1b := goalstate1b ∪ {c ↦ a}
         END
END
```

FIGURE 4.7: Event-B Model for a THEN Elaboration

The use of the subset relationship for controlling the states of the model is convenient for the THEN relationship. An alternative is to use disjoint sets and to remove the conversation from one state before it can move into its next state. The use of disjoint sets is

more reflective of an OR relationship. Figure 4.8 shows an Event-B model that uses disjoint sets to model a THEN relationship. The variables $goalState1a$ and $goalState1b$ are modelled as disjoint, $goalState1a \cap goalState1b = \varnothing$. The THEN relationship is upheld by the guards and actions of the events. The event $goalEvent1a$ takes a conversation that is in neither of the states and adds it to $goalState1a$. The event $goalEvent1b$ takes a conversation that is in the variable $goalState1a$, removes it and adds it to $goalState1b$.

The advantage of using the subset relationship in the Event-B models is that the relationships between the states that represent the different goals of the model are specified in the invariants of the model. This ensures that the abstract goal elaboration relationships are upheld in the refinement of the Event-B models more strongly than if specified only in the event guards. This also makes the discharge of some of the refinement proof obligations easier as the invariant conditions can be used in the proof. Because the THEN elaboration creates the order of the events it is the relationship that will be used most often, especially at the higher levels of abstraction. Therefore, another advantage of using the subset relationship between the states is that it reflects the main relationships in the model.

```
VARIABLES
   goalState1a, goalState1b
INVARIANTS
   goalState1a, goalState1b ∈ CONVERSATION ⇸ AGENT
   goalState1a ∩ goalState1b = ∅
EVENTS
    INITIALISATION
          BEGIN
            goalState1a, goalState1b := ∅
          END
    goalEvent1a
          ANY  c, a  WHERE
            c ∈ CONVERSATION
            a ∈ AGENT
            c ∉ dom(goalState1a)
            c ∉ dom(goalState1b)
          THEN
            goalState1a := goalState1a ∪ {c ↦ a}
          END
    goalEvent1b
          ANY  c, a  WHERE
            c ↦ a ∈ goalState1a
          THEN
            goalState1a := goalState1a \ {c ↦ a}
            goalState1b := goalState1b ∪ {c ↦ a}
          END
END
```

FIGURE 4.8: Alternative Representation of Goals in Event-B

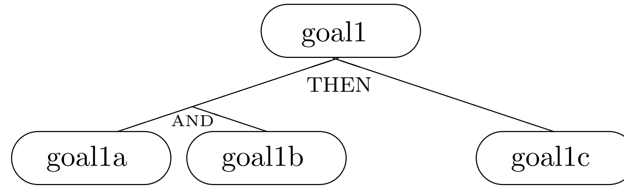

Figure 4.9: AND-THEN Goal Elaboration

An AND relationship cannot be modelled in isolation in Event-B using the specification style employed by the Process. This is because the completion condition for the whole AND relationship cannot be described by the completion conditions for the individual events. Instead, a separate event must be included in the Event-B model that will occur when all of the AND-related events have occurred. This can be modelled in the goal diagram by combining the AND relationship with a THEN relationship, as shown in Figure 4.9.

The AND relationship between the goals $goal1a$ and $goal1b$ must be fulfilled before the THEN-related goal, $goal1c$. There is no order for the fulfillment of the AND-related goals. $goal1c$ has a starting condition that all of the AND-related events have occurred. An invariant condition can be added to the Event-B model to specify that the state variable for $goal1c$ is a subset of an intersection of the state variables for the AND-related goals: $goal1c \subseteq goal1a \cap goal1b$. The guard for each of the events for $goal1a$ and $goal1b$ specifies that the selected conversation is not in the state variable for the goal and the action adds the conversation to that state variable. The guard of the event for $goal1c$ should specify that the selected conversation is in each of the state variables for the AND-related goals, $c \in dom(goalstate1a)$ and $c \in dom(goalstate1b)$.

Figure 4.10 shows how an AND-THEN goal elaboration relationship can be used to guide the development of an Event-B refinement. The refinement relationship is specified similarly to the THEN elaboration model. The goal diagram in Figure 4.9 can be described as $goal1 \sqsubseteq (goal1a\ AND\ goal1b)\ THEN\ goal1c$. Because of the THEN relationship the events for the AND-related goals, $goalevent1a$ and $goalevent1b$ are new events in the model and the event $goalevent1c$ refines the abstract event. The gluing invariant relates the abstract state variable $goalstate1$ to the concrete variable $goalstate1c$.

The XOR relationship between two sub-goals creates an exclusive choice of which of the goals must be fulfilled. Exclusive events in an Event-B model would have disjoint guards. However, to model the XOR relationship between two goals the Event-B model requires two events that can be triggered from the same state. To model this in the specification style used in the Process the guards for the events must be the same and the action of each event must disable both guards. An invariant condition is added to the model to specify the state variables as disjoint: $goalstate1a \cap goalstate1b = \varnothing$. This means that each conversation can be in only one of the states. A guard is added to each of the events that specifies that the event can only be triggered if the selected

```
VARIABLES goalstate1a, goalstate1b, goalstate1c
INVARIANTS
    goalstate1a, goalstate1b ∈ CONVERSATION ⇸ AGENT
    goalstate1c ⊆ goalstate1a ∩ goalstate1b
    goalstate1 = goalstate1c
EVENTS
    INITIALISATION
        BEGIN
          goalstate1a, goalstate1b, goalstate1c := ∅
        END
      goalevent1a
        ANY  c, a  WHERE
          c ∈ CONVERSATION
          a ∈ AGENT
          c ∉ dom(goalstate1a)
        THEN
          goalstate1a := goalstate1a ∪ {c ↦ a}
        END
      goalevent1b
        ANY  c, a  WHERE
          c ∈ CONVERSATION
          a ∈ AGENT
          c ∉ dom(goalstate1b)
        THEN
          goalstate1b := goalstate1b ∪ {c ↦ a}
        END
      goalevent1c  REFINES  goalevent1
        ANY  c, a  WHERE
          c ↦ a ∈ goalstate1a
          c ↦ a ∈ goalstate1b
          c ↦ a ∉ goalstate1c
        THEN
          goalstate1c := goalstate1c ∪ {c ↦ a}
        END
END
```

Figure 4.10: Event-B Model for an AND-THEN Elaboration

conversation is not in the state of the opposing goal in the relationship as well as the state for the event goal, $c \notin goalstate1a$ and $c \notin goalstate1b$. The goal relationships can be specified in the models by only using the event guards to enforce the conditions in which the events can be triggered. Including the invariant condition makes the relationship a necessary property of the model. This property can then be used in later refinements to discharge proof obligations. The goal diagram in Figure 4.3 can be described as $goal1 \sqsubseteq goal1a\ XOR\ goal1b$. Figure 4.11 shows how an XOR elaboration can be used to create choice in an Event-B refinement.

The refinement in Event-B using an XOR elaboration allows the abstract event to be

refined by either of the refining events. The completion condition for the abstract event is equivalent to either of the completion conditions for the sub-goals. A gluing invariant is added that specifies that the state for the abstract goal is equal to a conjunction of the states for the sub-goals, $goalstate1 = goalstate1a \cup goalstate1b$. Both of the events for the sub-goals are specified as refining the abstract event.

```
VARIABLES goalstate1a, goalstate1b
INVARIANTS
   goalstate1a, goalstate1b ∈ CONVERSATION ⇸ AGENT
   goalstate1a ∩ goalstate1b = ∅
   goalstate1 = goalstate1a ∪ goalstate1b
EVENTS
    INITIALISATION
         BEGIN
           goalstate1a, goalstate1b := ∅
         END
      goalevent1a  REFINES  goalevent1
         ANY  c, a  WHERE
           c ∈ CONVERSATION
           a ∈ AGENT
           c ∉ dom(goalstate1a)
           c ∉ dom(goalstate1b)
         THEN
           goalstate1a := goalstate1a ∪ {c ↦ a}
         END
      goalevent1b  REFINES  goalevent1
         ANY  c, a  WHERE
           c ∈ CONVERSATION
           a ∈ AGENT
           c ∉ dom(goalstate1b)
           c ∉ dom(goalstate1a)
         THEN
           goalstate1b := goalstate1b ∪ {c ↦ a}
         END
 END
```

FIGURE 4.11: Event-B Model for an XOR Elaboration

The OR relationship creates an inclusive choice between the goals. Each of the events in the Event-B model can be triggered from the same state whether or not the conversation is in the state for an OR-related goal.

It is not possible to directly refine an Event-B event using an OR relationship. The OR relationship is specified in Event-B in a way that will allow the events that are linked by an OR relationship to both occur. There is also no restriction on how many times they can occur, though the effect of the action will not affect the state of the model on a second occurrence. When refining an event into multiple events in Event-B the behaviour of the model should not diverge from the behaviour of the abstract model. A variable that is affected once by an abstract event should only be affected

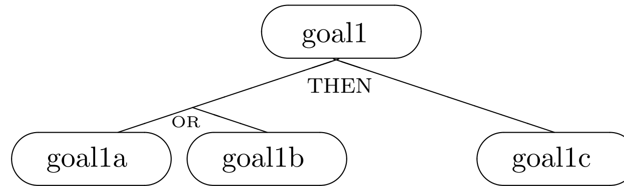

FIGURE 4.12: OR-THEN Goal Elaboration

once in a combination of the refining events. The specification of events for goals in an OR relationship does not guarantee that the behaviour does not diverge and cannot be proven to refine a single abstract event. This is possible with an XOR relationship as only one of the related events can occur once. An OR relationship can be specified in an abstract machine because it is not restricted by the need for a refinement relationship.

An OR elaboration can be included in a refinement model with an extra event that specifies the when the interaction will move on. This means that the OR relationship in the goal model, that is below the first level of goal elaboration, must always be on the left of a THEN relationship. A goal diagram of an OR-THEN elaboration is shown in Figure 4.12. The THEN relationship is seen as a causal relationship in this case as the occurrence any of the OR-related events will cause the event for $goal1c$ to occur. The events for $goal1a$ and $goal1b$ can still continue to occur after this, but this will not affect the rest of the model.

A refinement model that uses an OR-THEN elaboration relationship is shown in Figure 4.13. As with the AND-THEN relationship the refinement is similar to the THEN relationship refinement. The gluing invariant specifies that the abstract $goalState1$ variable is refined by the concrete $goalsState1c$ variable. $goalEvent1c$ refines the abstract event $goalEvent1$ and $goalEvent1a$ and $goalEvent1b$ are new events.

The translation between the goal model and the Event-B model can be assisted by writing the goal model as a statement, e.g. $goal1$ *THEN* ($goal2$ *XOR* $goal3$), using brackets to confirm the order of the evaluation for the relationships.. This tactic will place the focus on the relationships that need to be described in the Event-B specification and may help to highlight any mistakes in the formation of the relationships.

```
VARIABLES
   goalState1a, goalState1b, goalState1c
INVARIANTS
   goalState1a, goalState1b ∈ CONVERSATION ⇸ AGENT
   goalState1c ⊆ goalState1a ∪ goalState1b
   goalState1 = goalState1c
EVENTS
    INITIALISATION
         BEGIN
           goalState1a, goalState1b, goalState1c := ∅
         END
    goalEvent1a
         ANY  c, a  WHERE
           c ∈ CONVERSATION
           a ∈ AGENT
           c ∉ dom(goalState1a)
         THEN
           goalState1a := goalState1a ∪ {c ↦ a}
         END
    goalEvent1b
         ANY  c, a  WHERE
           c ∈ CONVERSATION
           a ∈ AGENT
           c ∉ dom(goalState1b)
         THEN
           goalState1b := goalState1b ∪ {c ↦ a}
         END
      goalEvent1c  REFINES   goalEvent1
         ANY  c, a  WHERE
           c ↦ a ∉ goalState1c
           c ↦ a ∈ goalState1a ∪ goalState1b
         THEN
           goalState1c := goalState1c ∪ {c ↦ a}
         END
END
```

FIGURE 4.13: Event-B Model for an OR-THEN Elaboration

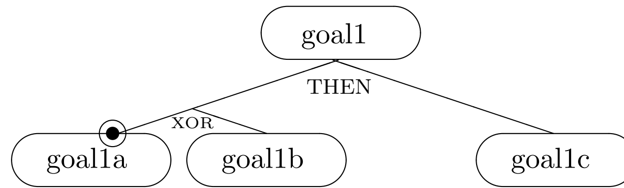

Figure 4.14: Goal Elaboration With an Endpoint Goal

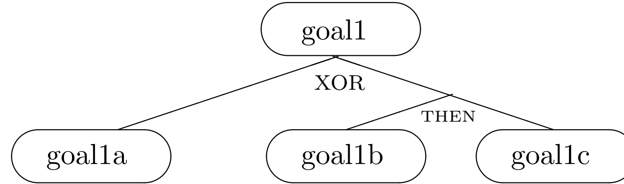

Figure 4.15: Goal Elaboration Without an Endpoint Goal

4.2.4 Endpoint Goals

In a multi-agent system it is possible for the interaction to end before it has been completed. An example of this behaviour in a multi-agent interaction is when a refuse goal is fulfilled because an agent has no intention of completing the requested action. The goals that represent these endpoints can be should be highlighted in the goal model with a circle that represents a stop, as shown in Figure 4.14. The fulfillment of the endpoint goal, $goal1a$, in Figure 4.14 will end the interaction. The fulfillment of $goal1b$ will lead to the fulfillment of $goal1c$ and the interaction will be completed.

Without endpoint goals any goal diagrams that include several stages of agent interaction could become quite complex. To avoid using endpoint goals the first goal elaboration relationship would split the first goal into an XOR relationship between Success and Failure goals. All of the goals that end the interaction prematurely could then be elaborated from the Failure goal. The XOR relationship would require the common goals in the interaction to be repeated on both sides of the relationship. This may then lead to multiple events in the Event-B model for the same goal for the interaction to be modelled.

In some cases the same affect as an endpoint goal could be achieved by nesting the goals differently. For example Figure 4.15 describes the same relationships as Figure 4.14. The goal diagram in Figure 4.14 makes the ordering of the interaction of the relationships clearer. The fact that there is meant to be a choice between $goal1a$ and $goal1b$ is shown in Figure 4.14, whereas Figure 4.15 describes a choice between $goal1a$ and $goal1b$ *THEN* $goal1c$. Both diagrams have the same affect, but the emphasis is on the choice between $goal1a$ and $goal1b$ in Figure 4.14 and this may describe the desired interaction better depending on the system being modelled.

The use of an endpoint goal that elaborates an XOR-related goal will end the conversation as the other related goals will not be able to be fulfilled. The main use of an

endpoint goal is in an XOR-THEN elaboration relationship, as shown in Figure 4.14. If there is no THEN relationship after the XOR then the conversation will end following the fulfillment of either $goal1a$ or $goal1b$ regardless of one of them being an endpoint goal. The use of an endpoint goal to elaborate an OR-related goal will not end the conversation as the other related goals may still be fulfilled and the conversation may then continue. This may be useful in a system that uses one-to-many interactions.

The use of an endpoint goal to elaborate an AND-related goal is not recommended as it may lead to an undesirable affect. For example, if an agent needs to respond with three messages(m1, m2, m3) in any order before the conversation can continue this can be modelled using the relationships $m1$ *AND* $m2$ *AND* $m3$ *THEN Continue*. If one of the AND-related goals were to be modelled as an endpoint goal and it were to be fulfilled before one of the other AND-related goals then the *Continue* goal would be prevented from being fulfilled. This would mean that the order of execution of the AND-related goals would affect the subsequent behaviour, which is not the intended characteristic of the AND relationship.

The addition of an endpoint goal will often be used to introduce a safety property to the Event-B model. This means that the safety of the model is strengthened by the use of endpoint goals. The use of endpoint goals in the Event-B models can lead to an event that was previously enabled at one point in the abstract model being disabled in the refinement model. This is a disadvantage of using endpoint goals, but using them allows the models to specify the system with one version of the interaction. An endpoint event that is in an abstract machine will not affect the enableness as it is part of the behaviour of the abstract machine.

A similar mechanism can be found in the StAC business processing language based on process algebra (Butler and Ferreira (2004)). An *early termination* ends a business process before the main tasks have been concluded. The presence of an early termination, $\odot$, in a block of tasks can cause the following tasks to be skipped. The scope of an early termination is determined by braces e.g. $\{A;\ \odot;\ B\} \,\|\, S$ allows S to continue processing when the early termination is executed within the block.

The intended scope of the endpoint goal in the goal diagram is within each conversation. In a system with one-to-one interactions the fulfillment of the endpoint goal will end the entire conversation. In a system with one-to-many interactions the fulfillment of the endpoint goal will end the particular agents participation in the conversation, but the conversation may continue with other participants. If the agent is the initiator of the conversation this may end the entire conversation. If all of the participants fulfill the endpoint goal this would have the same affect as in the one-to-one interaction.

The specification of an endpoint goal in an Event-B model will alter the invariants and guards of the model. The Event-B model shown in Figure 4.16 is a specification of

the goal model shown in Figure 4.14. Because `goal1a` is an endpoint goal the invariant specifies *goalstate1c* as a subset of *goalstate1b*, but not *goalstate1a*. The guard of *goalevent1c* allows it to occur only if the conversation is in the domain of *goalstate1b*: $c \in dom(goalstate1b)$. If the XOR relationship was replaced with an OR relationship then the *goalevent1b* could still occur after *goalevent1a* and *goalevent1c* would then be enabled.

VARIABLES goalstate1a, goalstate1b, goalstate1c

INVARIANTS

$goalstate1a,\ goalstate1b \in CONVERSATION \nrightarrow AGENT$

$goalstate1a \cap goalstate1b = \varnothing$

$goalstate1c \subseteq goalstate1b$

$goalstate1 = goalstate1c$

EVENTS

INITIALISATION

BEGIN

$goalstate1a,\ goalstate1b,\ goalstate1c := \varnothing$

END

goalevent1a

ANY c, a WHERE

$c \in CONVERSATION$

$a \in AGENT$

$c \notin dom(goalstate1a)$

$c \notin dom(goalstate1b)$

THEN

$goalstate1a := goalstate1a \cup \{c \mapsto a\}$

END

goalevent1b

ANY c, a WHERE

$c \in CONVERSATION$

$a \in AGENT$

$c \notin dom(goalstate1b)$

$c \notin dom(goalstate1a)$

THEN

$goalstate1b := goalstate1b \cup \{c \mapsto a\}$

END

goalevent1c REFINES goalevent1

ANY c, a WHERE

$c \mapsto a \in goalstate1b$

THEN

$goalstate1c := goalstate1c \cup \{c \mapsto a\}$

END

END

FIGURE 4.16: Event-B Model for an XOR-THEN Elaboration with an Endpoint Goal

The use of endpoint goals and the non-guarantee provided by their use resembles the autonomy of a multi-agent system. An agent in a system will not guarantee to complete a task and so there is no guarantee that an interaction will be concluded as expected.

4.2.5 One-to-Many Interactions

Multi-agent systems often involve interactions between several agents. An agent may begin a conversation with a group of agents within the system. The Event-B models must reflect this by modelling sets of agents fulfilling their goals. Stage Two of the Process will refine the models to show how individual agent's actions lead to the collective actions.

In the Event-B models the state variables that represent sets of agents will be specified as many-to-many relationship, $\leftrightarrow$, between conversations and agents. With one-to-one communication the state variables record the conversation and the agent that has the goal related to the first event of the interaction. With one-to-many interaction it is the many agents whose state needs to be recorded.

Figure 4.17 shows an abstract Event-B model of a goal that involves multiple agents. The variable for the goal is specified as a many-to-many relationship between the conversations and the agents, $goalstate \in CONVERSATION \leftrightarrow AGENT$. The event *goalevent* is parameterised by a conversation and a relationship that has the conversation as its domain and a subset of the `AGENT` set as its range. This relationship is added to the state variable by the action of the event.

```
VARIABLES goalstate
INVARIANTS
  goalstate ∈ CONVERSATION ↔ AGENT
EVENTS
    INITIALISATION
        BEGIN
          goalstate := ∅
        END
    goalevent
        ANY c, as WHERE
          c ∈ CONVERSATION
          as ∈ CONVERSATION ↔ AGENT
          c ∉ dom(goalstate)
          dom(as) = {c}
          ran(as) ⊆ AGENT
        THEN
          goalstate := goalstate ∪ as
        END
 END
```

FIGURE 4.17: Event-B Model for a Goal That Uses Broadcast Communication

The modelling of one-to-many interactions does not affect how the goal elaboration relationships are used to guide the refinement of the Event-B models. The multiplicity of the parameters of the events are preserved through the refinement, but the refinement is carried out in the same manner as with the single agent communication. For example, Figure 4.18 shows a THEN refinement of a goal that uses multiple agents.

```
VARIABLES goalstate1a, goalstate1b
INVARIANTS
   goalstate1a ∈ CONVERSATION ↔ AGENT
   goalstate1b ⊆ goalstate1a
   goalstate = goalstate1b
EVENTS
    INITIALISATION
         BEGIN
           goalstate1a, goalstate1b := ∅
         END
    goalevent1a
         ANY c, as WHERE
           c ∈ CONVERSATION
           as ∈ CONVERSATION ↔ AGENT
           c ∉ dom(goalstate1a)
           dom(as) = {c}
           ran(as) ⊆ AGENT
         THEN
           goalstate1a := goalstate1a ∪ as
         END
    goalevent1b REFINES goalevent
         ANY c, as WHERE
           c ∈ dom(goalstate1a)
           c ∉ dom(goalstate1b)
           as ⊆ {c} ◁ goalstate1a
         THEN
           goalstate1b := goalstate1b ∪ as
         END
 END
```

FIGURE 4.18: Event-B Model for a Refinement That Uses Broadcast Communication

The variables for the refinement are specified in the same manner as the variables in the THEN refinement shown in Figure 4.7. The events follow the same pattern as the THEN refinement above with the new event adding a many-to-many relationship to the new variable for the goal $goal1a$. The event for $goal1b$ refines the abstract event. The guard $c \in dom(goalstate1a)$ ensures that $goalevent1a$ has occurred. The action of the event adds a relationship that is a subset of the one added to $goalstate1a$, $as \subseteq \{c\} \lhd goalstate1a$, to the variable $goalstate1b$. The $\lhd$ operator restricts the domain of $goalstate1a$ to the contents of the set $\{c\}$.

The specification in Figure 4.18 models a set of agents first reaching $goalstate1a$ and then reaching $goalstate1b$. This is different from a set of agents where the individual agents first reach $goalstate1a$ and then reach $goalstate1b$, which would be a more realistic model of the behaviour of a distributed system. The refinement of the models through Stage Two of the Process will introduce more detail and model how the actions of the individual agents contribute to combined behaviour such as that modelled in Figure 4.18.

4.3 Summary

Stage One of the Incremental Development Process refines the goal-based requirements of a multi-agent system through the construction of a goal diagram that models the goals of the system. The goal diagram can be used to guide the development of Event-B formal models of the interacting system. This separates the modelling decisions from the Event-B modelling.

The goal diagram captures the motivational aspects of a multi-agent system and the relationships used help to create a state transition model of interaction based on the goals and modelled in Event-B. The goal diagram was chosen as the informal model, initially, because it is based on agent motivations, but also because it is a model that is accessible to different stakeholder and it can be integrated with early-phase requirements analysis. The goal elaboration relationships used in the goal diagram were chosen because they are an intuitive way to decompose the goals and they can be represented in Event-B.

The following chapter will describe how Stage Two of the Process will use the goal diagram constructed for Stage One to create Event-B models by adding distributed coordination to the model.

Chapter 5

Incremental Development Process: Stage Two - Distributed Coordination

This chapter describes Stage Two of the Incremental Development Process. The goal diagrams from Stage One are developed to include the distributed coordination mechanisms required for a multi-agent system. The different steps required to complete Stage Two are described.

Stage Two of the Incremental Development Process builds on the models developed during Stage One. The agent perspective is introduced by refining the goal diagram developed in Stage One to introduce the goals needed for the agents to coordinate through communication, adding the required agent roles and then allocating the goals to the different agent roles. The Event-B models created in Stage One can then be refined to include the agent roles as specified in the goal model and then decomposed into the components that represent the agent roles and their required resources. The state of the system model becomes separated into, and encapsulated by, the individual agent roles that construct the system. The components are specified as Event-B abstract machines that can be further refined.

The change from the system perspective to the agent perspective should change the way that the developer views the goal diagram to an interaction between separated entities. When the interactions are between one-to-many or many-to-many agents the change of view is more marked, as the communicating goals added to the diagram may be fulfilled multiple times for each agent that is involved in the interaction.

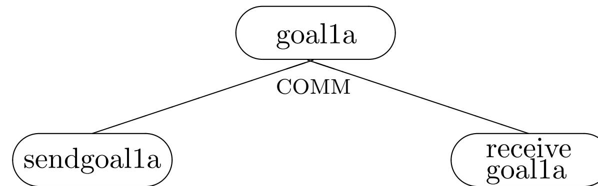

Figure 5.1: Goal Elaboration With Communicating Goals

5.1 Communicating Goals

The first step in the Process for Stage Two is to refine the goal diagram to add communicating goals. The agents in the system need to be able to coordinate their individual goals to be able to fulfill the goals of the system. Refining the goals into sub-goals whose fulfillment requires communication will allow the abstract goals to be coordinated between the separate agents in the system. This strengthens the representation of interaction in the models.

Pairs of communicating goals are used to elaborate the goals in the goal diagram that was developed in Stage One of the Process. These goals represent the sending and the receiving of a communication. The goals are linked by a COMM relationship. The COMM relationship is similar to the THEN relationship as the communication has to be sent before it can be received. The COMM relationship differs from the THEN relationship when it is being specified in the Event-B models. Figure 5.1 shows a goal that has been elaborated by communicating goals.

5.2 Broadcast Communication

Agents that are involved in a one-to-many interaction may need to communicate by broadcast, or multi-cast, communication. The models need to be able to represent communications that are sent by and received by more than one agent. To indicate broadcast communicating goals in a goal diagram the COMM relationship is changed to a BCOMM relationship as shown in Figure 5.2. When a send goal is changed to represent a broadcast communication it occurs once, but its corresponding receive goal will need to be able to occur multiple times for each message that is sent.

Any communicating goals that are replies to the original broadcast may also need to be changed to show their multiplicity if they are to be fulfilled by multiple agents. Communicating goals that are on the right-hand side of a THEN relationship with a pair of broadcast goals may be replies that need to be highlighted in the goal diagram as occurring multiple times. For each reply to a broadcast message an agent will send a message that will then be received. This multiplicity for a set of sub-goals is indicated using a *COMM relationship as shown in Figure 5.3.

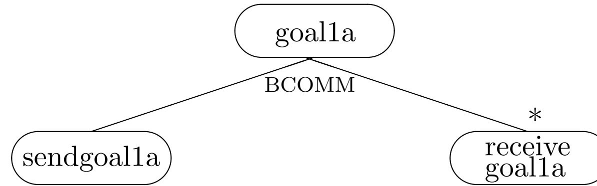

FIGURE 5.2: Goal Elaboration With Broadcasting Goals

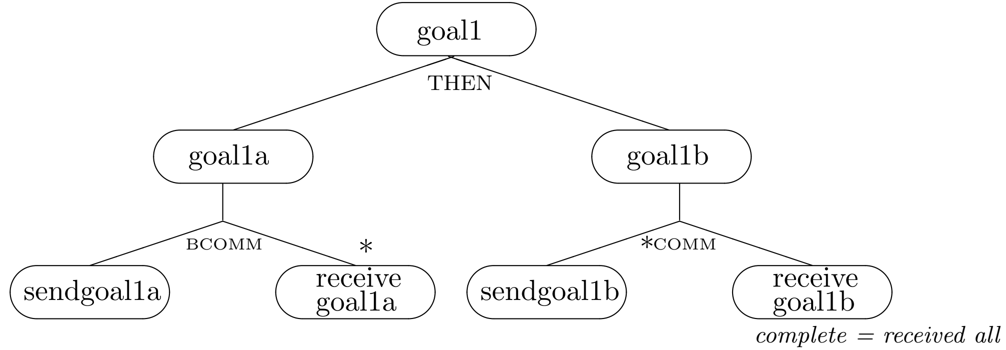

FIGURE 5.3: A Broadcast Goals Elaboration With a Reply

TABLE 5.1: Available Completion Conditions

Annotation	**Description**	**Example Guard Condition**
receive all	Responses received from all agents	$\{c\} \lhd informR = \{c\} \lhd proposeS$
receive 1+	At least one response received	$c \in dom(proposeR)$
receive X	Received specified number(X) of responses	$card(\{c\} \lhd informR) = X$
after deadline	The deadline for responses has passed	$c \in afterTimeout$
selection	A selected group of agents have responded	$\{c\} \lhd informR = \{c\} \lhd selected$

The multiplicity introduced by the broadcast communications changes the completion condition for the goals. The completion condition for a goal to the left of a BCOMM relationship remains the same as it only happens once. The completion conditions for a goal to the right of a BCOMM relationship and for goals in a *COMM relationship will change. Because each goal can occur multiple times the completion condition for a goal will be relevant only for a single agent. Therefore, the completion condition for all agents, or for the whole system, will have to be specified separately. An annotation can be made to the goal diagram to indicate a system-wide completion condition for a goal that is to the right of a BCOMM or in a *COMM relationship. The available system completion conditions can be found in Table 5.1 along with examples of guard conditions that can be used to specify the completion conditions in the Event-B models.

5.3 Role Allocation

The next part of Stage Two of the Process is to identify the roles of the agents that will be involved in the interaction and allocate the communicating goals to the individual agent roles. At the end of the Process the system model will be decomposed into components

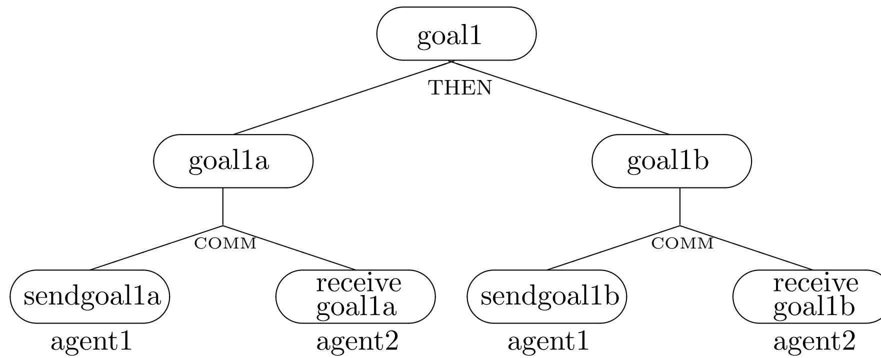

FIGURE 5.4: A THEN Elaboration With Role Allocation

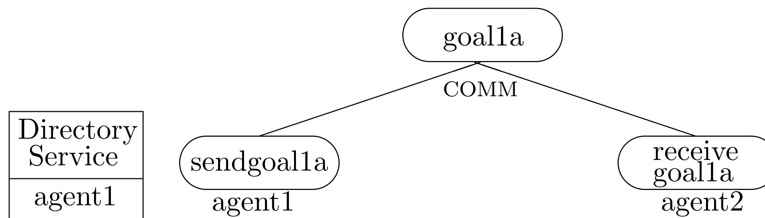

FIGURE 5.5: Goal Diagram With Resource Allocation

that include the agent roles identified in this step. The goal diagrams can be updated to show the allocation of the agent roles to the goals by writing an identifier for the agent role beneath the goal. When the role that the goal is being allocated will be performed by multiple agents in an interaction a subscript will be added to the role identifier to show this multiplicity e.g. $agent1_n$. Figure 5.4 shows a goal diagram for a THEN elaboration that has been given communicating goals and the roles for the agents have been identified and allocated.

Allocating the roles changes the view of the model from the system to the individual agents. Each agent will encapsulate their own state and only be aware of the state of the system when they receive a communication that informs them of a change in state. When allocating agent roles to the communication goals the developer must consider the information about the interaction that is available to the agents that are performing the roles.

5.4 Allocating Resources

The resources that the agents need to perform their actions need to be added to the goal diagram. Resources are modelled using a box that names the resource. The resource should be placed to one side of the goal diagram. The names of the agent roles that use the resource can be listed as attributes of the resource. Figure 5.5 shows a goal diagram with a resource added. The resource is a directory service and it is to be used by the `agent1` role to find names of agents in the system that can perform specific roles. The resource may become one of the component models when the system is decomposed.

All changes have been made to the goal diagram and the Event-B models can now be refined to reflect the changes that have been made.

5.5 Refining the Event-B Models

The changes made to the goal diagram in Stage Two can be added as a single refinement step to the Event-B model that resulted from Stage One. The COMM relationship used to create the communicating goals is similar to the THEN relationship with the send event being specified to occur before the receive event. One difference is that new variables are introduced to represent the communicating goals and variables for the goals from the previous refinements remain, rather than being refined by the new variables. Another difference is how the new events and state variables for the communicating goals refine the events in the abstract model. This is dependent on the how the goals have been allocated to the agent roles. Using broadcast communicating goals in the goal diagram changes how the state variables are specified. The role allocation introduces new variables to the Event-B model. These new variables are used to identify the agents that have particular roles in a conversation and restrict which events can be triggered by these agents.

Figure 5.6 shows an Event-B refinement with just the communicating goals added. The goals are not broadcast goals. Other changes will need to be made to the model in this refinement step to include all of the information added to the goal diagram. These changes, including the role allocation, are introduced later in this section.

```
VARIABLES goalstate1a, goalstate1aS,  goalstate1aR
INVARIANTS
   goalstate1aS ⊆ goalstate1a
   goalstate1aR ⊆ goalstate1aS
EVENTS
  INITIALISATION
     BEGIN
     goalstate1a, goalstate1aS, goalstate1aR := ∅
     END
 sendgoalevent1a  REFINES  goalevent1a
     ANY  c, a  WHERE
        c ∈ CONVERSATION
        a ∈ AGENT
        c ↦ a ∉ goalstate1aS
        c ∉ dom(goalstate1a)
     THEN
        goalstate1aS := goalstate1aS ∪ {c ↦ a}
        goalstate1a := goalstate1a ∪ {c ↦ a}
     END
  END

receivegoalgoal1a
  ANY  c, a  WHERE
     c ∈ CONVERSATION
     a ∈ AGENT
     c ↦ a ∈ goalstate1aS
     c ↦ a ∉ goalstate1aR
  THEN
     goalstate1aR := goalstate1aR ∪ {c ↦ a}
  END
```

FIGURE 5.6: Event-B Model for the Communicating Goals Elaboration

Variables are introduced to the model for each of the communicating goals. The variables for the send goal represent a communication being generated and have been suffixed with a 'S'. The variables for the receive goals represent a communication being received and have been suffixed with an 'R'. The variables are all defined as a partial function from the set of conversations to the set of agents. This allows one of each communication to be made to one agent in a conversation. The agent in the relationship represents the agent that will receive the message. The COMM relationship between the sending and receiving goals requires that the send goal is fulfilled before the receive goal. This is modelled in the Event-B refinement similarly to the THEN relationship with the receive goal state specified as a subset of the send goal state, $goalstate1aR \subseteq goalstate1aS$. Because there is no agent role allocation included in the model shown in Figure 5.6 the Event-B event refinement is specified between the abstract goal and the send goal. This is the default approach because the agent that makes the decision to fulfill the abstract goal is usually the agent that sends the communication. The allocation of the agent roles can require the refinement relationship to change and this will be described later in this section. The state for the send communicating goal is specified as a subset of the variable that represents the abstract goal as the action of fulfilling the abstract goal will lead to the fulfillment of the send goal, $goalstate1aS \subseteq goalstate1a$. The state for the abstract goal, e.g. $goalstate1a$, remains in the model to represent the state of the agent's reasoning. This variable has the potential to be refined further to represent other aspects of the agent's reasoning. The send event, $sendgoalevent1a$ in the model in Figure 5.6, refines the abstract event for the elaborated goal. The action of the event adds the relationship between the conversation and agent to the states for the communicating send goal and the abstract goal. The event for the receive goal is specified as a new event. The action of the event adds the relationship to the state for the receive goal, $goalstate1aR$.

Goals that have not been elaborated with communicating goals may not need to be refined any further. Data refinement may still need to be applied. For example, if an abstract event, $event1$, whose goal has not been elaborated has the guard $c \in state1$ and in the refinement model the variable $state1$ is refined by a conjunction of the variables $state1a$ and $state1b$ then the guard for $event1$ in the refinement model will refer to either a conjunction of the variables, $state1a \cup state1b$, or, in some cases, to just one of them e.g. $c \in state1a$.

5.5.1 Broadcast Communication

Using broadcast communications will restrict which of the events for the communicating goals can refine the abstract event. The sending of a broadcast message from a single agent to multiple agents in the system will occur only once and the send communicating goal can refine the abstract goal. The receive goal will need to be able to occur multiple

times and this event must be introduced as a new event. In the Event-B model the refinement of an abstract event to a pair of broadcast communicating goals that send a broadcast message to be received by multiple agents should always be to the send event.

```
VARIABLES  goalstate1a, goalstate1aS, goalstate1aR
INVARIANTS
   goalstate1aS ⊆ goalstate1a
   goalstate1aR ⊆ goalstate1aS
EVENTS
  INITIALISATION
     BEGIN
      goalstate1a, goalstate1aS, goalstate1aR := ∅
     END
  sendgoalevent1a  REFINES  goalevent1a
     ANY  c, as  WHERE
        c ∈ CONVERSATION
        as ∈ CONVERSATION ↔ AGENT
        c ∉ dom(goalstate1a)
        dom(as) = {c}
        ran(as) ⊆ AGENT
     THEN
        goalstate1aS := goalstate1aS ∪ as
        goalstate1a := goalstate1a ∪ as
     END
  END
  receivegoalevent1a
     ANY  c, a  WHERE
        c ∈ CONVERSATION
        a ∈ AGENT
        c ↦ a ∈ goalstate1aS
        c ↦ a ∉ goalstate1aR
     THEN
        goalstate1aR := goalstate1aR ∪ {c ↦ a}
     END
```

FIGURE 5.7: Event-B Model for Broadcast Goals Elaboration

Figure 5.7 shows how an Event-B model can be refined when modelling broadcast communication. This refinement differs from the refinement shown in Figure 5.6 that has no broadcast goals. The state variables for the broadcast goals have been specified as a subset of the parent goal, *goalstate1a*, which was specified as a many-to-many relationship between the set of conversations and the set of agents in the abstract model. This is because there can be several agents involved in each communication in a conversation. The rest of the invariants in the model remain as specified in Figure 5.6 that has no broadcast goals. The receive event is a new event that models each individual communication being received.

5.5.2 Role Allocation

A description of agent roles is given in Chapter 3 Section 3.4.3. When allocating agent roles in the Process a role can be seen as the type of agent that is responsible for fulfilling one or more of the goals that have been modelled in the goal diagram. Using roles rather than agents at this level of abstraction allows a concrete specification of an agent to be specified as capable of more than one of the roles developed in the Process.

For each pair of communicating goals there should be two different roles. One of the roles will be allocated the send goal and the other will be allocated the receive goal. Which of the roles is responsible for the fulfillment of the abstract goal will determine

which of the events in the Event-B model will refine the abstract event. Often the agent role that is allocated the send goal will be responsible for the abstract goal. The agent will act to send a message because it has a goal that can be fulfilled by sending the message.

The allocation of roles requires new variables to be added to the refinement that will specify the roles of the agents involved in the interaction. Guards are added to the events to specify the role that is required for the agent to be involved in a particular event. At this stage in the Event-B refinement chain the role variables are specified as global variables. This is because the information being communicated between the agents by the state variables is restricted to the conversation and one agent, or in the case of one-to-many interactions the set of agents. When the information is extended to include all the agents involved in the interaction it will be possible for the agents to encapsulate their own record of the different roles of the agents involved in the interaction. This will be possible later in the Process when resources for communication between the agents are modelled.

The variable that specifies an agent's role will be a relationship between the set of conversations and the set of agents. The domain of the relationship is the domain of the state for the send goal that is the left-most goal in the diagram that the role has been allocated. This is because the roles are decided when the agents are selected for the first communication that is sent. The relationship is specified as a total function, $\rightarrow$, which specifies a many-to-one relationship between the conversation and the agent. This variable should be specified as a many-to-many relationship, $\leftrightarrow$, if the role is at the receiving end of a broadcast communication.

The events of the model will be updated to specify the identity of each agent in each role being recorded on the left-most send events. This means that when a message is sent by an agent to another agent both of their roles in the interaction are specified. The guards of the remaining events in the model are updated to include a specification of the role for the agents that are involved in the communication.

Figure 5.8 shows how the Event-B refinement from Figure 5.6 can been updated to include the agent role specification. The variables required to specify the agent role allocation have been added to a model with communicating goals. The role variables, *agent1* and *agent2*, have been specified as a total function between the state variable for the first send goal and the set of agents, $agent1, agent2 \in dom(goalstate1aS) \rightarrow AGENT$. $sendgoalevent1a$ now includes a variable to represent the sending agent that is distinct from the variable that represents the receiving agent, $a \neq a2$. The action of the event adds a relationship between the conversation and the sending agent, a, to the variable for the agent role that has been allocated the event, $agent1$. A relationship between the conversation and the receiving agent, $a2$, is added to the variable for the role that has been allocated the receive goal, $agent2$. For each of the remaining events

a guard has been added that ensures that the communication is for, or to, the correct agent e.g $c \mapsto a \in agent2$.

```
VARIABLES
   goalstate1a, goalstate1aS, goalstate1aR, agent1, agent2
INVARIANTS
   goalstate1aS ⊆ goalstate1a
   goalstate1aR ⊆ goalstate1aS
   agent1, agent2 ∈ dom(goalstate1aS) → AGENT
EVENTS
  INITIALISATION
     BEGIN
      goalstate1aS, goalstate1aR, agent1, agent2 := ∅
     END
    sendgoalevent1a  REFINES  goalevent1a
     ANY  c, a, a2  WHERE
        c ∈ CONVERSATION
        a ∈ AGENT
        a1 ∈ AGENT
        a ≠ a2
        c ↦ a ∉ goalstate1aS
        c ∉ dom(goalstate1a)
     THEN
        goalstate1aS := goalstate1aS ∪ {c ↦ a}
        goalstate1a := goalstate1a ∪ {c ↦ a}
        agent1 := agent1 ∪ {c ↦ a}
        agent2 := agent2 ∪ {c ↦ a2}
     END
  END

    receivegoalevent1a
     ANY  c, a  WHERE
        c ∈ CONVERSATION
        a ∈ AGENT
        c ↦ a ∈ agent1
        c ↦ a ∈ goalstate1aS
        c ↦ a ∉ goalstate1aR
     THEN
        goalstate1aR := goalstate1aR ∪ {c ↦ a}
     END
```

Figure 5.8: Event-B Model With Role Allocation

5.5.3 Resource Allocation

All of the information added to the goal diagram for Stage Two has been added as a single refinement step. The resources that have been added to the goal diagram can either be added in the same refinement step as the communicating goals and roles, or to reduce the complexity of the refinement it may be preferable to add them as a single new refinement step. If there are several resources it may be preferable to have one refinement for each resource.

For each resource that is included in the model a variable that abstractly models its behaviour needs to be added to the Event-B model. For a directory service, like the one added to the goal diagram in Figure 5.5, a variable can be added that relates roles to agents. Every multi-agent system in which the agents communicate by message passing will require a messaging medium resource. This can be modelled as a set of messages.

Some shared resources that are added to the model will refine variables that are already in the Event-B model. The messaging medium resource will be used to replace the state variables for the send communicating goals in the events for the receive communicating goals. This will allow the state variables to be encapsulated and the messaging medium variable will be used to communicate changes in state. Other resources will be added as a superposition refinement to the model. A directory service resource can be used as a global variable. A guard can be added to events that will use the directory service to select an agent that can perform a particular role.

The decision for exactly how to model a resource is left to the developer as there are many different types of resources that could be added. The rest of this section provides a couple of examples of how resources can be modelled. Some resources may best be implemented as an agent role in the system rather than as a resource. A developer should consider this option when designing the system.

Adding a Directory Service Resource

A directory service resource will be used by agents to find the names of other agents in the system that can perform a particular role. An agent can use this knowledge to make specific requests from agents that it knows are capable of fulfilling the request.

Adding a directory service resource to the model can be done in a single refinement step. The context is extended to include a set that contains the different roles available for the system. A variable is added to the model that relates the roles to agents: $directory \in ROLE \leftrightarrow AGENT$. A guard is then added to the appropriate events to use the *directory* variable to restrict the event variable that represents the agent or agents that are involved in the interaction. Figure 5.9 shows an extract of an Event-B refinement that adds a directory service. The event shown is the same as one of the

events shown in Figure 5.8, but with an additional guard. This new guard has been underlined to highlight that it has been added to previous versions of this event. The *ROLE* set defined in the context includes two roles, $\{agent1role, agent2role\}$.

```
INVARIANTS
   directory ∈ ROLE ↔ AGENT
EVENTS
    sendgoalevent1a  REFINES   sendgoalevent1a
      ANY  c, a, a1  WHERE
        c ∈ CONVERSATION
        a ∈ AGENT
        a1 ∈ AGENT
        a ≠ a1
        c ↦ a ∉ goalstate1aS
        c ∉ dom(goalstate1a)
        agent2role ↦ a ∈ directory
      THEN
        goalstate1aS := goalstate1aS ∪ {c ↦ a}
        agent1 := agent1 ∪ {c ↦ a1}
        agent2 := agent2 ∪ {c ↦ a}
      END
```

FIGURE 5.9: Event-B Model for Directory Service Resource Refinement

The directory service can be added to the Event-B model as a superposition refinement as it has not been added as a shared resource and does not affect the interactions of the agents in the system. A shared resource can be a more challenging refinement, as will be shown in the next section.

Adding and Refining a Messaging Medium Resource

A messaging medium resource will be used by the agents to send messages to one another to enable them to coordinate their actions. To be able to send the messages they must be added to the model along with the variables that model the messaging medium to carry the messages. Adding the messages allows more information to be passed between the agents and the state of each agent role can be hidden from the other agent roles. The variables that communicate the messages can then be further refined to model a generic messaging medium.

The Event-B context is extended to include a message record type. The specification of the message creates a record with fields for the *sender*, *receiver* and *conversation*. The specification of the context in Figure 5.10 uses the syntax proposed in Evans and Butler (2006). The syntax implicitly specifies a set of messages and types the *sender* and *receiver* fields as: $sender, receiver \in MESSAGE \rightarrow AGENT$ and the *messageConversation* field as: $messageConversation \in MESSAGE \rightarrow CONVERSATION$. The record can be extended to include further fields in refinement.

For the agents to be able to pass messages coordination variables are added to the system model. The variables are specified as a subset of the message record set. There

$$\begin{array}{lll} MESSAGE & :: & sender \in AGENT, \\ & & receiver \in AGENT, \\ & & messageConversation \in CONVERSATION \end{array}$$

FIGURE 5.10: Extended Context for Step One

is one coordination variable for each of the sending and receiving goal pairs to make the refinement step easier. A further refinement will specify a single messaging medium for the system. The coordination variables are used to replace the state variables for the send goals in the guards of the receive events. The relationships between the agents and the conversations can be ascertained from the fields of the message so the send variables can be encapsulated. Figures 5.11 and 5.12 show the first refinement model for the messaging medium.

$$\begin{array}{l} goalstate1aM \subseteq MESSAGE \\ goalstate1aS = ((goalstate1aM \lhd messageConversation)^{-1};\ sender) \\ agent2agent1,\ agent2agent2 \in dom(goalstate1aR) \rightarrow AGENT \\ agent2agent1 \subseteq agent1 \\ agent2agent2 \subseteq agent2 \\ agent1 = (goalstate1aM \lhd messageConversation)^{-1};\ sender \\ agent2 = (goalstate1aM \lhd messageConversation)^{-1};\ receiver \end{array}$$

FIGURE 5.11: State of the First Event-B Refinement for the Messaging Medium

The invariants shown in Figure 5.11 include a condition that relates the variables that record a message being sent to the related fields of the message record, $goalstate1aS = ((goalstate1aM \lhd messageConversation)^{-1};\ sender)$. This allows the send variables to be encapsulated for each agent role and the coordination variables to replace any access to them in the guards of events that belong to other agent roles. The send variable, $goalstate1aS$, is a set of relationships between $CONVERSATION$ and $AGENT$. The gluing invariant specifies this relationship as equal to a composition of the *messageConversation*, $MESSAGE \rightarrow CONVERSATION$, and *sender*, $MESSAGE \rightarrow AGENT$, fields of the message record.

Suppose

$$\begin{array}{l} m \in goalstate1aM, \\ sender(m) = s, \\ receiver(m) = r \text{ and} \\ messageConversation(m) = c \end{array}$$

then

$$\begin{array}{l} c \mapsto r \in (goalstate1aM \lhd messageConversation)^{-1};\ receiver \\ \text{and } c \mapsto s \in (goalstate1aM \lhd messageConversation)^{-1};\ sender. \end{array}$$

The $\lhd$ operator restricts the domain of $messageConversation$ to the messages that are in $goalstate1aM$. The result of this restriction is then inverted using the inverse operator, $^{-1}$. The composition relation, ; , composes the fields on the shared $MESSAGE$ element of the record field relationship.

Given

$$m \mapsto c \in messageConversation\ and$$
$$m \mapsto a \in sender$$

then the composition

$$\{m \mapsto c\}^{-1};\ \{m \mapsto a\}$$

produces

$$\{c \mapsto a\}$$

This style of invariant will be referred to as a 'join' operator. It uses the record variable, e.g. m, to join the fields of the record, e.g. a and c, in a relationship. The join operator could be used to relate the fields of any record variable.

For the agents to be autonomous they need to be able to encapsulate their state and behaviour. The previous refinement model includes variables that record the agents performing the different roles in the conversation. This state information needs to be encapsulated within each agent role. With the addition of messages to the model more information is being communicated between the agents. Previously, only the identity of one agent and the conversation was passed in place of messages. Now the message record can identify both the sender and receiver of the message. An agent that receives a message can record information about the agent that sent the message as well as their own role in the interaction. The last five invariants in Figure 5.11 are used to specify new variables that record role information. The role variables from the abstract model, $agent1$ and $agent2$, can be considered to be encapsulated by the agent1 role. Two new variables, $agent2agent1$ and $agent2agent2$, are specified. $agent2agent1$ is agent2's local encapsulation of the $agent1$ role information. The invariants specify the new variables as relationships between the domain of the variable for the receive goal and the set of agents, $agent2agent1 \in dom(goalstate1aR) \rightarrow AGENT$. This is because the roles will first be recorded by the agent2 role when the first message is received. The variables are also specified as subsets of the existing role variables so the information they record can be proven consistent. The final invariants use the join operator to describe the relationship between the existing role variables and the message fields. This is specified in the same way as the relationship between the send goal variables and the messages. The difference is that the agent1 role is related to the sender of the message and the

agent2 role is related to the receiver of the message. The new role variables can then be proven to replace the existing ones in the guards of the events that have been allocated to the agent2 role.

```
sendgoalevent1a REFINES sendgoalevent1a
  ANY  c, a, a2, m  WHERE
    c ∈ CONVERSATION
    a ∈ AGENT
    a1 ∈ AGENT
    c ↦ a ∉ goalstate1aS
    c ∉ dom(goalstate1a)
    a ≠ a2
    m ∈ MESSAGE
    sender(m) = a
    receiver(m) = a2
    messageConversation(m) = c
  THEN
    goalstate1aS := goalstate1aS ∪ {c ↦ a}
    agent1 := agent1 ∪ {c ↦ a}
    agent2 := agent2 ∪ {c ↦ a2}
    goalstate1aM := goalstate1aM ∪ {m}
  END

receivegoalevent1a
 REFINES receivegoalevent1a
  ANY  c, a, m  WHERE
    c ∈ CONVERSATION
    a ∈ AGENT
    c ∉ dom(goalstate1aR)
    m ∈ goalstate1aM
    sender(m) = a
    messageConversation(m) = c
  THEN
    goalstate1aR := goalstate1aR ∪ {c ↦ a}
    agent2agent1 := agent2agent1 ∪ {c ↦ a}
    agent2agent2 := agent2agent2 ∪
                      {c ↦ receiver(m)}
  END
```

FIGURE 5.12: Events of the First Event-B Refinement for the Messaging Medium

Figure 5.12 show the events of this refinement where the separate messaging medium variables are used to communicate between the events and the role variables are only accessed in the events for the roles that will encapsulate them. Each of the events in the refinement model is parameterised by an element of the message set. The event guards restrict the fields of the message so they can be related to the other variables used in the event, e.g. $sender(m) = a$. The action of the send event adds the message event variable to the coordination variable, $goalstate1aM := goalstate1aM \cup \{m\}$. The receive event restricts the message event variable to be an element of the coordination variable. The role variables that are encapsulated by the receiving agent role, $agent2agent1$ and $agent2agent2$, are updated by the action of the event to the sender and receiver of the message. The receive event only needs to access the coordination variable and the fields of the message record.

The specification of the coordination variables creates a different set of messages for each communicating goals pair. To create a more realistic model of a generic communication medium the separate sets must be combined in a further refinement. To be able to model a single medium the type of message being sent must be distinguishable. The context could been extended to add a type field to the message record. Because of the possibly large number of different message types for an interaction it is recommended that the definition of the field uses a technique suggested in Butler and Yadav (2007). The message types are specified as disjoint sets. The disjoint sets are simpler for the provers to distinguish and lead to a larger amount of automatic proof than when using the record fields to distinguish the types. Figure 5.13 shows the extended context with the definition of the message types.

```
PROPERTIES
        GOAL1A, GOAL1B ⊆ MESSAGE
        disjoint(GOAL1A, GOAL1B)
```

Figure 5.13: Further Extended Context for the Messaging Medium

The second refinement of the messaging medium refines the multiple coordination variables into a single set of messages to which each message is added. The type of the message being sent is added to the guard of each event, $m \in GOAL1A$. The gluing invariants describe each of the abstract sets as an intersection between the single set and the set for the corresponding message type e.g. $goalstate1aM = msgset \cap GOAL1A$. Figure 5.14 shows a sending and receiving event that have been refined to use the single set of messages as the coordination medium.

```
INVARIANTS
   msgset ⊆ MESSAGE
   goalstate1aM = msgset ∩ GOAL1A
   goalstate1bM = msgset ∩ GOAL1B
EVENTS
 sendgoalevent1a REFINES sendgoalevent1a
   ANY c, a, a2, m WHERE
      c ∈ CONVERSATION
      a ∈ AGENT
      a2 ∈ AGENT
      c ↦ a ∉ goalstate1aS
      c ∉ dom(goalstate1a)
      a ≠ a2
      m ∈ MESSAGE
      sender(m) = a
      receiver(m) = a2
      messageConversation(m) = c
      m ∈ GOAL1A
   THEN
      goalstate1aS := goalstate1aS ∪ {c ↦ a}
      agent1 := agent1 ∪ {c ↦ a}
      agent2 := agent2 ∪ {c ↦ a2}
      msgset := msgset ∪ {m}
   END

 receivegoalevent1a REFINES receivegoalevent1a
   ANY c, a, m WHERE
      c ∈ CONVERSATION
      a ∈ AGENT
      c ∉ dom(goalstate1aR)
      m ∈ msgset
      sender(m) = a
      messageConversation(m) = c
      m ∈ GOAL1A
   THEN
      goalstate1aR := goalstate1aR ∪ {c ↦ a}
      agent2agent1 := agent2agent1 ∪ {c ↦ a}
      agent2agent2 := agent2agent2 ∪
                        {c ↦ receiver(m)}
   END
```

Figure 5.14: Events From the Second Event-B Refinement for the Messaging Medium

The goals of the system have been allocated to the individual agent roles and the behaviour of the system has been fully encapsulated within the agent roles and resources. No more changes need to be made to the goal diagram. The decisions required to separate the system into its individual components have been made and the Incremental Development Process continues by decomposing the Event-B model.

5.6 System Decomposition

Decomposing the Event-B model into the different component models will allow them to be further developed individually into more concrete models. The models developed to this stage can be quite large with a potentially large number of variables and events. Developing the component models individually will allow the developer to concentrate on each component. The decomposition will also reflect the distributed nature of the components.

The system model is to be decomposed into abstract machines that specify the components of the system. With the introduction of any required resources, such as the messaging medium, the shared behaviour of the system has been modelled. The only behaviour of the individual agents that is relevant to the system model is their ability to perform the interaction events. These are the events that have been introduced by the elaboration of the goal diagram with communicating goals. The reasoning behind the actions of the agents is not circumscribed by the system models. This preserves the autonomy of the agents and can be specified in refinements of the component models after decomposition.

To decompose the system the states and events for the goals are specified in an abstract machine for the agent role to which they have been allocated. The resources are specified as a variable in an abstract machine that has events for interacting with the resource. This creates agent role component models and component models of the environment.

The diagram in Figure 5.15 provides an overview how the decomposed model is synchronised. The components **agent1** and **agent2** both have access to the **middleware** component. The **agent1** component has the event `sendgoalevent1a`. The output of the event is a message and this can be synchronised with the input of the `send` event in the **middleware**. The receive event in the **middleware** has an output that is a message and this can be synchronised with the input of the `receivegoalevent1a` event of the **agent2** component.

The method of decomposition used in the Incremental Development Process is the event-based decomposition described in Chapter 2 Section 2.2.5. This allows the variables of the model to be separated into the different components and the components can then be refined individually. Figure 5.16 shows how the refinement model, *Synch*, synchronises the events of the component models, $agent1component$, $agent2component$ and *middleware*, to complete the behaviour specified in the abstract model, *System*. Each event of the refinement model accesses the associated event in the agent role component models. The events of the component for a messaging medium resource are used to transfer the messages between the components. In the model extract shown in Figure 5.16 the `sendgoalevent1a` event in the refinement model declares a local variable, m, and then instantiates it with the message that is output from the `sendgoalevent1a` event in

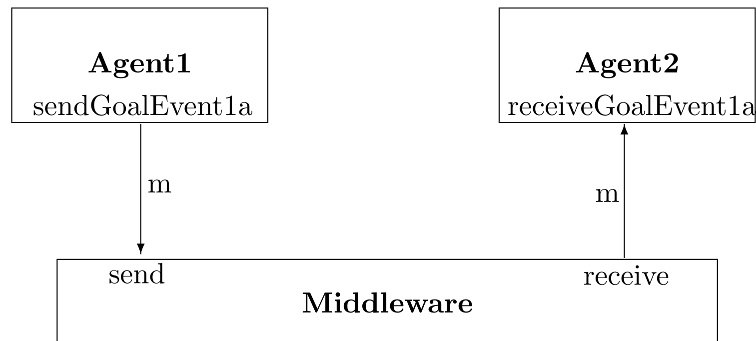

FIGURE 5.15: Synchronising the Refinement Model and the Abstract Component Models

the `agent1` component, $m \leftarrow ag1.sendgoalevent1a$. The message is then used as a parameter for the `send` event for the `middleware` component. The `receivegoalevent1a` in the refinement model is parameterised by an agent and a local variable, m, is declared. The receive event from the `middleware` component outputs a message for that agent, $m \leftarrow mw.receive(a)$. The `receivegoalevent1a` event in the agent2 component is parameterised with the message m, $ag2.receivegoalevent1a(m)$. The refinement relationship for the synchronising model, *Synch*, ensures that the combined behaviour of the component models can be proven to be consistent with the behaviour of the system model, *System*.

```
MACHINE  Synch  REFINES  System
SEES context
INCLUDES ag1.agent1component,
   ag2.agent2component, mw.middleware
EVENTS
   sendgoalevent1a  =
     VAR  m  IN
       m  ←  ag1.sendgoalevent1a;
       mw.send(m)
     END
   receivegoalevent1a  =
     ANY  a  WHERE
       a  ∈  AGENT
     THEN
         VAR  m  IN
           m  ←  mw.receive(a);
           ag2.receivegoalevent1a(m)
         END
     END
```

```
MACHINE agent1component SEES context
m0 <--  sendgoalevent1a  =
     ANY  c, a, a2, m  WHERE
       c  ∈  CONVERSATION
       a  ∈  AGENT
       a2  ∈  AGENT
       c  ↦  a  ∉  goalstate1aS
       c  ∉  dom(goalstate1a)
       a  ≠  a2
       m  ∈  MESSAGE
       sender(m)  =  a
       receiver(m)  =  a2
       messageConversation(m)  =  c
       m  ∈  goal1a
     THEN
       goalstate1aS  :=  goalstate1aS  ∪  {c  ↦  a}
       agent1  :=  agent1  ∪  {c  ↦  a}
       agent2  :=  agent2  ∪  {c  ↦  a2}
       m0  :=  m
     END
MACHINE agent2component SEES context
  receivegoalevent1a (m) =
     PRE  m  ∈  MESSAGE  THEN
        ANY  c, a  WHERE
          c  ∈  CONVERSATION
          a  ∈  AGENT
          m  ∈  GOAL1A
          c  ∉  dom(goalstate1aR)
          sender(m)  =  a
          messageConversation(m)  = c
         THEN
          goalstate1aR  :=  goalstate1aR  ∪  {c  ↦  a}
          agent2agent1  :=  agent2agent1  ∪  {c  ↦  a}
          agent2agent2  :=  agent2agent2  ∪  {c  ↦  receiver(m)}
        END
     END
```

```
MACHINE middleware
 SEES context
 send (m) =
    WHEN
       m  ∈  MESSAGE
     THEN
       msgset  :=  msgset  ∪  {m}
     END
 m <--receive (a) =
    PRE  a  ∈  AGENT THEN
       ANY  m0  WHERE
          m0  ∈  msgset
          receiver(m0)  =  a
       THEN
          m  :=  m0
       END
    END
```

FIGURE 5.16: Synchronising the Refinement Model and the Abstract Component Models

5.7 Discussion on Patterns

There are similarities between the Incremental Development Process and the use of design patterns in software engineering. Design patterns are intended to make software engineering easier by capturing the expertise of experienced software developers and making it available in a manner that can be re-applied in other developments (Gamma et al. (1995)). The purpose of a design pattern is to capture structures and decisions within a design that are common to similar modelling and analysis tasks. They can be re-applied when undertaking similar tasks in order to reduce the duplication of effort. Design patterns originate in architecture as designs that could be combined to create buildings, suburbs and towns (Alexander et al. (1977)). Design patterns were made popular in software engineering as a development aid for object-oriented software engineering by Gamma et al. (1995).

There are several definitions of software design patterns and their required elements. Gamma et al. (1995) define a pattern as 'the solution to a recurring problem in a particular context'. Riehle and Zuellighoven (1996) describe a pattern as 'the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts' and a design pattern as 'a pattern whose form is described by means of software design constructs'. Coad (1992) defines a pattern in a more specific way as 'a fully realized form, original, or model accepted or proposed for imitation'.

Collections of design patterns in software engineering can be presented in different forms. The patterns in Gamma et al. (1995) are presented as a catalogue with each pattern solving a different design problem and being applicable at a single level of abstraction. A pattern language is an interwoven set of patterns that can be combined to solve different modelling problems with alternative patterns that can be applied for different types of problems (Price (1999)).

Refinement patterns are a type of design pattern that capture the design decisions required to refine a model for a particular purpose (Moriconi et al. (1995)). This can be to add new elements to the model or to perform a common data refinement step. Refinement patterns for formal models should help to discharge the proof obligations generated by the refinement step (Leavens et al. (2006)).

Alexander suggests that each pattern should describe a recurring problem and then describe a re-usable solution (Alexander et al. (1977)). Design patterns in software are constructed from different elements. The patterns catalogued by Gamma et al. (1995) consist of elements such as name, problem, motivation and structure. The format for patterns in Beck and Johnson (1994) involves pre-conditions, problem, constraints and solution.

The Incremental Development Process is intended to make the development of multi-agent systems in Event-B easier by allowing agent concepts to be captured in Event-B

models. In the Process the same modelling steps are re-applied for each set of goals and relationships. There are three parts of the process that could be identified as patterns.

A THEN relationship between two goals will be translated into Event-B by following the guidelines. The translation of a THEN relationship into Event-B follows a pattern. This translation pattern is not a design pattern.

The goal relationships capture patterns used in Event-B, such as state transition and choice. These patterns have been abstracted as relationships for the goal diagrams and re-applied when following the Process to model the interactions of a multi-agent system. These are more like design patterns, but they are not as well defined. The XOR relationship can solve the problem ‘how do I model an agent having a choice between two goals?’, but it is also abstract enough to solve the problem ‘how do I model a system having a choice in Event-B?’. Examples of design patterns in software engineering are generally similar to those in Gamma et al. (1995) that solve more specific problems e.g. composing objects and selecting algorithms at runtime. The use of these goal relationship patterns is specific to the specification style used in the Process.

The final refinement steps of the Process could be described as refinement patterns. They describe a way of refining the model to incorporate features and change the structuring. In doing so encapsulate the entire refinement step, but currently, the proof obligations must be discharged each time the patterns are applied.

The problem that the Process is intended to solve is to guide the translation between the goal diagrams and the formal model in a way that captures the motivations of agents as the basis for their interactions. The different parts of the Process discussed above do not offer a core solution to this problem. They must be combined in the Process to solve this problem.

Chapter 8 presents a set of Event-B modelling patterns for modelling fault-tolerance in multi-agent systems. These patterns are closer to the definitions of design patterns. They address a specific problem and offer an abstract solution along with an Event-B example that can be re-used. The context for the problem is multi-agent systems and the problems are modelling the different behaviours required to provide a model of a fault-tolerant multi-agent system. The problem of modelling for fault-tolerance is one that will be found in many multi-agent systems.

5.8 Summary

Stage Two of the Incremental Development Process takes the goal diagram and Event-B models of the system created as part of Stage One and refines them to model the interacting agents and environment that embody the system. As with Stage One the goal diagram is used to separate the modelling decisions from the formal models.

The goal diagram is first elaborated to model the communications of the agents by the agent roles that are identified for the system. The Event-B models are then refined to incorporate the agent roles and communications. The Event-B models are then further refined to introduce the resources for the system and then the system model is decomposed into synchronised agent role and resource component models. These refinement steps are achieved by using a standard approach to refining the variables and events following the Process.

Chapter 6

Case Study : Query-If

This chapter presents the development of a system that uses a FIPA interaction protocol to show that the Incremental Development Process can be used to model a multi-agent system. The case study is based on the FIPA Query Interaction Protocol. The informal specification of the protocol can be found in FIPA (2002d). The informal specification contains two possible interactions; Query-Ref and Query-If. The Query-Ref interaction allows the interaction initiator to refer to an object as part of their query. The Query-If interaction allows the initiator to query whether a proposition is true or false. To provide a simple case study only one of the interactions has been developed. The Query-If interaction protocol was chosen as the first case study as it is a simple, three stage, interaction between no more than two agents. A further case study of a more complex interaction is presented later in the next chapter.

6.1 Case Study

The Query-If interaction begins with an agent that initiates the interaction by sending a *query* message to another agent in the system. The agent that receives the message will respond with either an *accept* or a *refuse* message. The accept message indicates that the participating agent intends to provide a response to the query. The refuse message informs the initiator that the participating agent does not intend to provide a response to the query. If the participant has accepted the query, the accept message will be followed by either an *inform* or a *failure* message. The inform message will provide a response to the query of either true or false. A failure message will let the initiator know that the participant was prevented from responding to the query.

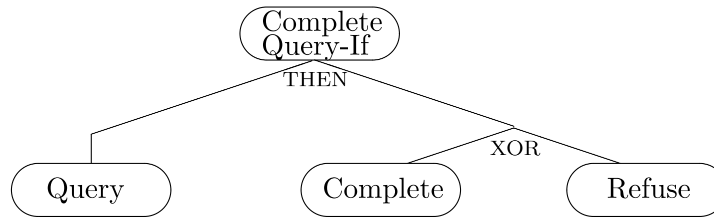

Figure 6.1: First Level of Goal Elaboration for Query-If

6.2 Stage One

To show the translation between the goal model and Event-B the first level of goal elaboration has been extracted from the goal diagram in Figure 6.1. The diagram shows that the overall goal for the system is first elaborated by three goals. The Query goal represents a query being made, the Complete goal represents the response to the query being completed and the Refuse goal represents the failure of the system to complete the query. The relationships in the goal diagram form an interaction that begins with the Query goal and is followed by either the Complete goal or the Refuse goal. Modelling the Refuse goal creates a system model that can cope with a negative result from a query request.

The abstract machine, shown in Figure 6.2, has three states and three events that correspond to the goals in the first level of goal elaboration in the goal diagram. The context for the model specifies a set of conversations. The left-most goal is the Query goal and so the `query` event is parameterised by a relationship between a conversation and agent that it adds to the `queried` state. The agent in the relationship represents the agent that has the goal that is fulfilled by the `query` event. The THEN relationship between the Query goal and the Complete and Refuse goals means that the states for the Complete and Refuse goals are specified as subsets of the `queried` state. The events for the Complete and Refuse goals also have a guard to ensure that they can only be triggered when the selected conversation is already in the `queried` state. The XOR relationship between the Complete and Refuse goals adds a further invariant to the model that specifies the `completed` and `refused` states as disjoint. The events for the Complete and Refuse goals have a further guard that prevents them from being triggered by a conversation that is in the opposing state.

Figure 6.3 shows the second level of goal elaboration. The first refinement of the Event-B abstract machine can be constructed by analysing this extract from the goal diagram.

The Complete goal has been further elaborated by three sub-goals. The first goal is the Accept goal and this represents an agent of the system having the goal to agree to the query request. The Inform goal represents the agent answering the query after they have accepted the query request. The Failure goal represents the agent being unable to answer the query after they have accepted the query request. The Inform and Failure goals are associated with an XOR relationship indicating that only one of them may

```
MACHINE  m0
SEES  context
VARIABLES queried, completed, refused
INVARIANTS
   queried ∈ CONVERSATION ⇸ AGENT
   completed, refused ⊆ queried
   refused ∩ completed = ∅
EVENTS
 INITIALISATION
     BEGIN
        queried, completed, refused := ∅
     END
  query
     ANY  c, a  WHERE
        c ∈ CONVERSATION
        a ∈ AGENT
        c ∉ dom(queried)
     THEN
        queried := queried ∪ {c ↦ a}
     END
  complete
     ANY  c, a  WHERE
        c ↦ a ∈ queried
        c ↦ a ∉ completed
        c ↦ a ∉ refused
     THEN
        completed := completed ∪ {c ↦ a}
     END
  refuse
     ANY  c  WHERE
        c ↦ a ∈ queried
        c ↦ a ∉ refused
        c ↦ a ∉ completed
     THEN
        refused := refused ∪ {c ↦ a}
     END
  END
```

FIGURE 6.2: Query-If: Abstract Machine

occur in an interaction. They are linked to the Accept goal by a THEN relationship. The Accept goal is on the left of the THEN relationship and must occur before either the Inform or Failure goals can occur. The XOR relationship between the abstract Complete and Refuse goals means that the system will either fulfill the Refuse goal or fulfill the Accept goal followed by either the Inform goal or the Failure goal.

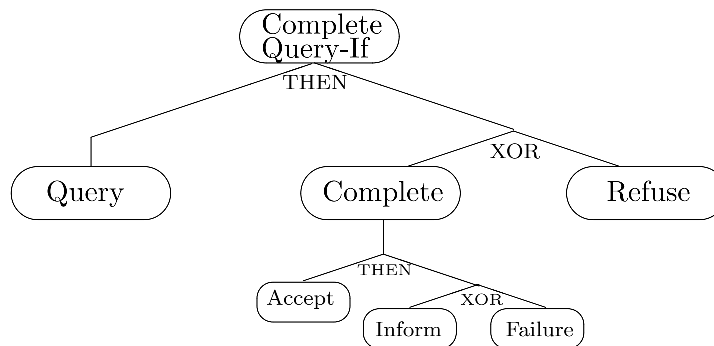

FIGURE 6.3: Second Level of Goal Elaboration for Query-If

The first refinement of the Event-B abstract machine can be constructed by analysing the second level of goal elaboration in the goal diagram. The Query and Refuse goals are not elaborated, so in the Event-B refinement model the events `query` and `refuse` are the same as the abstract events `query` and `refuse` and the specification of the state variables for the goals remains the same. The Event-B refinement model is shown in Figure 6.4.

The Complete goal from the first level of goal elaboration in the goal diagram is elaborated by three goals that are joined by two relationships. The goals and their relationships can be described as Accept THEN (Inform XOR Failure). The Event-B refinement is constructed by replacing the state and event for the Complete goal with this new set of goals and relationships. The gluing invariant is a conjunction of the states for the goals in the XOR relationship: $completed = informed \cup failed$. The *informed* and *failed* events are specified as refining the abstract `complete` event and the event for the Accept goal, *accept*, is introduced as a new event. The THEN relationship also creates an invariant that specifies the state for the Inform and Failure goals as a subset of the state for the Accept goal: $(informed \cup failed) \subseteq accepted$. The pattern for the XOR relationship adds an invariant that specifies the states for the goals as disjoint: $failed \cap informed = \varnothing$. The XOR relationship from the abstract model is now upheld using the *accepted* state: $accepted \cap refused = \varnothing$.

```
MACHINE  m1 REFINES  m0 SEES  context
VARIABLES
   queried, accepted, refused, informed, failed
INVARIANTS
   accepted ⊆ queried
   informed, failed ⊆ accepted
   accepted ∩ refused = ∅
   failed ∩ informed = ∅
   completed = informed ∪ failed
EVENTS
 INITIALISATION
     BEGIN
      queried, accepted, refused, informed, failed := ∅
     END
  query  REFINES  query
     ANY  c, a  WHERE
       c ∈ CONVERSATION
       a ∈ AGENT
       c ∉ dom(queried)
     THEN
       queried := queried ∪ {c ↦ a}
     END
  refuse  REFINES  refuse
     ANY  c, a  WHERE
       c ↦ a ∈ queried
       c ↦ a ∉ refused
       c ↦ a ∉ accepted
     THEN
       refused := refused ∪ {c ↦ a}
     END
  failure  REFINES  complete
     ANY  c, a  WHERE
       c ↦ a ∈ accepted
       c ↦ a ∉ failed
       c ↦ a ∉ informed
     THEN
       failed := failed ∪ {c ↦ a}
     END
  accept
     ANY  c, a  WHERE
       c ↦ a ∈ queried
       c ↦ a ∉ accepted
       c ↦ a ∉ refused
     THEN
       accepted := accepted ∪ {c ↦ a}
     END
  inform  REFINES  complete
     ANY  c, a  WHERE
       c ↦ a ∈ accepted
       c ↦ a ∉ informed
       c ↦ a ∉ failed
     THEN
       informed := informed ∪ {c ↦ a}
     END
END
```

FIGURE 6.4: Query-If: First Refinement

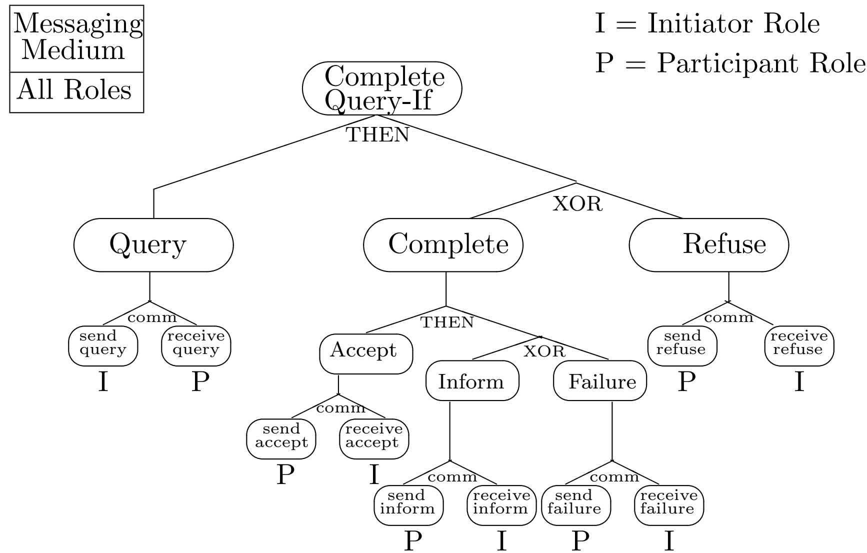

Figure 6.5: Goal Model for Query-If

6.3 Stage Two: Introducing Send and Receive

All of the goals in the goal diagrams involve interaction and have been be elaborated to communicating goals. The roles that can be identified for the Query-If case study are the *initiator* of the query and the *participant* that responds to the query. Each of the interactions in the system will be between two agents and there will be only one agent for each role in each conversation. A messaging medium resource is required for the agents to communicate. There are no broadcast goals required for this case study.

Figure 6.5 shows how the goals have been elaborated and the roles that have been assigned to each of the communicating goals. The allocation of the goals to the agent roles does not alternate for each of the communicating goals. Only the first send goal is allocated to the initiator agent who will then wait for a series of responses from the participant agent before the conversation is completed.

Because of the introduction of the events to model the communicating goals the Event-B models will be split into more than one figure or have only an extract shown in the rest of this section. The full specification of the models can be found in Appendix A.

Figure 6.6 shows the specification of the invariants of the communicating goals refinement. It specifies variables for each of the communicating goals in the goal diagram. The state variables for the send goals are suffixed with an ‘S’ and the state variables for the receive goals are suffixed with an ‘R’. The COMM relationship between each of the send and receive goals means that the state variables for the receive goals are specified as subsets of the state variables for the send goals. The send communicating goals have been allocated to the agent roles that are responsible for the abstract goals. This means that the refinement relationship for the abstract events and variables can be with the events and variables for the send communicating goals e.g. $queryS \subseteq queried$. The role

$$
\begin{array}{l}
queryS \subseteq queried \\
queryR \subseteq queryS \\
refuseS \subseteq queryR \\
refuseR \subseteq refuseS \\
acceptS \subseteq queryR \\
acceptR \subseteq acceptS \\
informS \subseteq acceptS \\
informR \subseteq informS \\
failS \subseteq acceptS \\
failR \subseteq failS \\
initiator,\ participant \in dom(queryS) \rightarrow AGENT \\
refuseS \subseteq refused \\
acceptS \subseteq accepted \\
informS \subseteq informed \\
failS \subseteq failed
\end{array}
$$

FIGURE 6.6: Query-If: Invariants of the Second Refinement

variables are modelled as total functions between the domain of the state variable for the first send goal and the set of agents: $dom(queryS) \rightarrow AGENT$.

Figures 6.7 and 6.8 show the events for the refinement. The initialisation event is not shown. All of the variables of the model are initialised as empty. The first event is the `sendQuery` event that is parameterised by conversation that is not in the domain of the `queried` state and two different agents. This event refines the `queried` event from the abstract model. The action of the event adds a relationship between the conversation and one of the agents to the `queryS`, `initiator` and `queried` variables. The other agent represents the agent that is receiving the message and is related to the conversation in the `participant` variable. The `receiveQuery` event can be triggered when there is a relationship between a conversation and the initiator of the conversation in the `queryS` state and the action of the event adds the relationship to the `queryR` state. When there is a relationship in the `queryR` state either the `sendAccept` or `sendRefuse` event can be triggered. These events refine the `accept` and `refuse` events from the abstract model, respectively. Each of the events has an action that adds a relationship to their associated state. The XOR relationship between the abstract goals requires that the states are specified as disjoint and the guards of the events prevent them from occurring if the relationship is already in the opposing state e.g. $c \mapsto a \notin refused$. The guards of the events also specify that the agent that is selected for the event to receive the message must be the agent that has the initiator role for the conversation. The `receiveAccept` and `receiveRefuse` events are both new events that can occur when there is a relationship in the appropriate send state and the action of the events will add the relationship to the receive state for the event. The agent that is selected for these events to receive the message must be the initiator of the conversation.

The events shown in Figure 6.8 continue the interaction after an accept message has been sent. Either the `sendInform` or the `sendFailure` event can be triggered when there is a relationship in the `accepted` state, $c \mapsto a \in accepted$. Similarly to the `sendAccept` and

```
sendQuery  REFINES  query
  ANY  c, a, ar  WHERE
    c ∈ CONVERSATION
    a ∈ AGENT
    ar ∈ AGENT
    c ∉ queried
    a ≠ ar
  THEN
    queryS := queryS ∪ {c ↦ a}
    initiator := initiator ∪ {c ↦ a}
    participant := participant ∪ {c ↦ ar}
    queried := queried ∪ {c ↦ a}
  END

receiveQuery
  ANY  c, a  WHERE
    c ↦ a ∈ queryS
    c ↦ a ∉ queryR
    c ↦ a ∈ initiator
  THEN
    queryR := queryR ∪ {c ↦ a}
  END

sendAccept  REFINES  accept
  ANY  c, a  WHERE
    c ↦ a ∈ queryR
    c ↦ a ∉ accepted
    c ↦ a ∉ refused
    c ↦ a ∈ initiator
  THEN
    acceptS := acceptS ∪ {c ↦ a}
    accepted := accepted ∪ {c ↦ a}
  END

receiveAccept
  ANY  c, a  WHERE
    c ↦ a ∈ acceptS
    c ↦ a ∉ acceptR
    c ↦ a ∈ initiator
  THEN
    acceptR := acceptR ∪ {c ↦ a}
  END

sendRefuse  REFINES  refuse
  ANY  c, a  WHERE
    c ↦ a ∈ queryR
    c ↦ a ∉ refused
    c ↦ a ∉ accepted
    c ↦ a ∈ initiator
  THEN
    refuseS := refuseS ∪ {c ↦ a}
    refused := refused ∪ {c ↦ a}
  END

receiveRefuse
  ANY  c, a  WHERE
    c ↦ a ∈ refuseS
    c ↦ a ∉ refuseR
    c ↦ a ∈ initiator
  THEN
    refuseR := refuseR ∪ {c ↦ a}
  END
```

FIGURE 6.7: Query-If: Events of the Second Refinement (Part 1)

`sendRefuse` events the abstract events for the `sendInform` and `sendFailure` events are in an XOR relationship and so must occur exclusively from one another for each conversation. The `receiveInform` and `receiveFailure` events are new events that can be triggered when there is a relationship between the conversation and the initiator of the conversation in the `informS` and `failureS` states, respectively. The actions of the events add the relationship to the state for their respective receive goal.

6.4 Stage Two: Introducing the Messaging Medium

The next Event-B refinement model introduces the messaging medium resource. The refinement model adds the messages that are sent between the agents and the medium through which they are sent. Adding the messages and the messaging medium allows the agents to fully encapsulate their own state information as they will be able to exchange more information through the message fields. They will communicate through

```
sendInform   REFINES   inform
    ANY   c, a   WHERE
        c ↦ a ∈ accepted
        c ↦ a ∉ informed
        c ↦ a ∉ failed
        c ↦ a ∈ initiator
    THEN
        informS := informS ∪ {c ↦ a}
        informed := informed ∪ {c ↦ a}
    END
sendFailure   REFINES   failure
    ANY   c, a   WHERE
        c ↦ a ∈ accepted
        c ↦ a ∉ failed
        c ↦ a ∉ informed
        c ↦ a ∈ initiator
    THEN
        failS := failS ∪ {c ↦ a}
        failed := failed ∪ {c ↦ a}
    END
```

```
receiveInform
  ANY   c, a   WHERE
      c ↦ a ∈ informS
      c ↦ a ∉ informR
      c ↦ a ∈ initiator
  THEN
      informR := informR ∪ {c ↦ a}
  END

receiveFailure
  ANY   c, a   WHERE
      c ↦ a ∈ failS
      c ↦ a ∉ failR
      c ↦ a ∈ initiator
  THEN
      failR := failR ∪ {c ↦ a}
  END
```

Figure 6.8: Query-If: Events of the Second Refinement (Part 2)

the variables that represent the messaging medium and these become the only variables that are global to the agents in the system.

To model the messages the Event-B context is extended to include a message record with sender, receiver and conversation fields. The specification of the extended context for this development is the same as the specification shown in Figure 5.10.

```
INVARIANTS
        queryM, acceptM, refuseM, informM, failureM ⊆ MESSAGE
        queryS = (queryM ◁ messageConversation)⁻¹; sender
        acceptS = (acceptM ◁ messageConversation)⁻¹; receiver
        refuseS = (refuseM ◁ messageConversation)⁻¹; receiver
        informS = (informM ◁ messageConversation)⁻¹; receiver
        failureS = (failureM ◁ messageConversation)⁻¹; receiver
        pInitiator, pParticipant ∈ dom(queryR) → AGENT
        pInitiator ⊆ initiator
        pParticipant ⊆ participant
        initiator = (queryM ◁ messageConversation)⁻¹; sender
        participant = (queryM ◁ messageConversation)⁻¹; receiver
```

Figure 6.9: Query-If: State of the Third Refinement

The state of the third refinement is shown in Figure 6.9. To model the messaging medium a subset of the message record set is specified for each of the types of message that are to be sent in the model e.g *queryM*. The specification still includes the send and receive variables as these are to be used by each agent to keep track of their current state within the interaction. There is a gluing invariant that relates each of the new message sets to the send variables. The content of the message sets can be proven to replace the send variables in the guards of the receive events and the send variables can remain private within the state of the appropriate agent. The gluing invariant,

e.g. $queryS = (queryM \lhd messageConversation)^{-1}; sender$, uses the join operator, as described in Chapter 5 Section 5.5.3, to relate the information held in the receiver and conversation fields of the messages to the information held in the send variables.

Because of the information that can be exchanged by the agents using the messages the role variables can be encapsulated in this refinement. Two new role variables, `pInitiator` and `pParticipant`, are specified as subsets of the original `initiator` and `participant` role variables. A gluing invariant relates the sender, receiver and message conversation fields of the message to the original role variables. This can be used to prove that relationships added to the role variables in the `receiveQuery` event, by the participant, are a subset of the role variables encapsulated by the initiator. The invariant uses the join operator with: $initiator = (queryM \lhd messageConversation)^{-1}; sender$ relating the initiator variable to the conversation and sender of the message and: $participant = (queryM \lhd messageConversation)^{-1}; receiver$ relating the conversation and receiver of the message to the participant. The invariant condition requires that it is proven that these are the values recorded in the action of the `sendQuery` event and enables the fields of the message to replace the reference to the original role variables in the guards of events that are allocated to the participant role.

The invariant conditions used to relate the content of the messages to the send and role variables follow a pattern. They all use the join operator. The information that is now contained in the model with the introduction of messages could allow the variables for the send goals e.g. *queryS* to be removed. However the model is based on the goals of the agents and when the roles are decomposed later in the Process the retention of these variables will allow the individual models to keep a more detailed model of the agents actions in relation to its goals. It may also be the case that a send goal is not directly related to one of the earlier abstract goals and the variable would be the only record of the send goal being fulfilled.

```
sendQuery  REFINES sendQuery
    ANY  c, a, ar, m  WHERE
       c ∈ CONVERSATION
       a ∈ AGENT
       ar ∈ AGENT
       c ∉ dom(queried)
       a ≠ ar
       m ∈ MESSAGE
       sender(m) = a
       receiver(m) = ar
       messageConversation(m) = c
    THEN
       queryS := queryS ∪ {c ↦ a}
       initiator := initiator ∪ {c ↦ a}
       participant := participant ∪ {c ↦ ar}
       queried := queried ∪ {c ↦ a}
       queryM := queryM ∪ {m}
    END
```

```
receiveQuery  REFINES receiveQuery
   ANY  c, a, m  WHERE
      m ∈ queryM
      c ↦ a ∉ queryR
      sender(m) = a
      messageConversation(m) = c
   THEN
      queryR := queryR ∪ {c ↦ a}
      pParticipant := pParticipant ∪ {c ↦ receiver(m)}
      pInitiator := pInitiator ∪ {c ↦ a}
   END
```

FIGURE 6.10: Query-If: Example Events from the Third Refinement

Figure 6.10 shows the `sendQuery` and `receiveQuery` events from the third refinement of the Event-B model. The refinement of the other events of the model are similar to the refinement of these two events. The full refinement model can be found in Appendix A. Both of the events are refined to include an extra event variable that represents the message that is being exchanged. The additional guards of the events specify the content of the message fields that are required for the message. An additional action of the `sendQuery` event adds the message to the `queryM` message set. The guards of the `receiveQuery` event no longer access the `queryS` variable and instead check that the message is in the `queryM` message set. The guards no longer access the original role variables. This refinement can be proven through the invariants that relate the original role variables to the fields of the message. The action of the `receiveQuery` event adds the information from the message fields to the new role variables.

The fourth refinement refines the multiple message sets into a single message set that models a generic messaging medium. This has been achieved with the Query-If case study by extending the context to include disjoint constant sets for each of the message types and using these types to replace the message sets from the third refinement with a single message set in the fourth refinement. Figure 6.11 shows the specification of the extended context and Figure 6.12 shows the specification of the state of the fourth refinement. The single message set is specified as a subset of the set of messages in the same way as the multiple sets in the abstract model. The gluing invariants relate each of the abstract message sets to an intersection of the single message set and the type that has been specified in the context. A further set `PROPOSITION` and two constants, `question` and `answer`, have been added to the context so the query and the response to the query can be modelled. When a query message is sent in the case study it will include the proposition that the initiator agent requires answering. When an inform message is sent in the case study it will include a response of either true or false to the original query. Because the fields of the messages are specific to the type of message they have been introduced at the same stage as the message types.

`SETS`

$\quad PROPOSITION$

`PROPERTIES`

$\quad QUERY,\ ACCEPT,\ REFUSE,\ INFORM,\ FAIL \subseteq MESSAGE$

$\quad question \in QUERY \rightarrow PROPOSITION$

$\quad answer \in INFORM \rightarrow BOOL$

$\quad disjoint(QUERY,\ ACCEPT,\ REFUSE,\ INFORM,\ FAIL)$

Figure 6.11: Query-If: Extended Context for the Fourth Refinement

The events of the fourth refinement have an additional guard specifying to which message type the message event variable belongs. The action of the send events is refined to add the message to the single message set rather than one of the multiple sets. Figure 6.13 shows how the `sendQuery` and `receiveQuery` events have been specified for the fourth refinement.

```
INVARIANTS
   msgset ⊆ MESSAGE
   queryM = msgset ∩ QUERY
   acceptM = msgset ∩ ACCEPT
   refuseM = msgset ∩ REFUSE
   informM = msgset ∩ INFORM
   failureM = msgset ∩ FAIL
```

FIGURE 6.12: Query-If: State of the Fourth Refinement

```
sendQuery REFINES sendQuery
   ANY c, a, ar, m WHERE
      c ∈ CONVERSATION
      a ∈ AGENT
      ar ∈ AGENT
      c ∉ dom(queryS)
      a ≠ ar
      m ∈ MESSAGE
      m ∈ QUERY
      sender(m) = a
      receiver(m) = ar
      messageConversation(m) = c
      question(m) ∈ PROPOSITION
   THEN
      queryS := queryS ∪ {c ↦ a}
      initiator := initiator ∪ {c ↦ a}
      participant := participant ∪ {c ↦ ar}
      queried := queried ∪ {c ↦ a}
      msgset := msgset ∪ {m}
   END
```

```
receiveQuery REFINES receiveQuery
   ANY c, a, m WHERE
      m ∈ msgset
      c ∉ dom(queryR)
      sender(m) = a
      messageConversation(m) = c
      m ∈ QUERY
      question(m) ∈ PROPOSITION
   THEN
      queryR := queryR ∪ {c ↦ a}
      pParticipant := pParticipant ∪
                         {c ↦ receiver(m)}
      pInitiator := pInitiator ∪ {c ↦ a}
   END
```

FIGURE 6.13: Query-If: Example Events of the Fourth Refinement

The next step of the Process is to decompose the Event-B model into abstract component models. The components for the Query-If case study are the initiator agent role, the participant agent role and the messaging medium. The decomposition is possible because the shared behaviour of the system has been modelled and each component can be refined separately from the other components.

This refinement step creates the fifth refinement model that is used to synchronise the components and the three abstract component models that have been decomposed from the case study. This allows the synchronisation of the components to be proven a refinement of the abstract model ensuring that the full functionality modelled by the abstract model is fulfilled by the components.

Figure 6.14 shows how the fifth refinement synchronises the initiator and middleware components to refine the abstract `sendQuery` event. The `sendQuery` event in the refinement model creates a new variable `m`. The output from the `sendQuery` event in the initiator component is assigned to the variable `m`. The variable is then used as an input parameter to the `send` event of the middleware component that encapsulates the coordination medium. The abstract components are proven to fulfill the functionality

```
REFINEMENT m5  REFINES   m4
  sendQuery  =
    VAR  m  IN
      m ← initiator.sendQuery;
      mw.send(m)
    END
```

```
MACHINE initiator0
mr <--  sendQuery  =
   ANY  a, c, ar, m   WHERE
      a ∈ AGENT ∧
      c ∈ CONVERSATION ∧
      c ∉ dom(queryS) ∧
      ar ∈ AGENT ∧
      ar ≠ a ∧
      m ∈ MESSAGE ∧
      sender(m) = a ∧
      receiver(m) = ar ∧
      messageConversation(m) = c ∧
      m ∈ QUERY
      question(m) ∈ PROPOSITION
   THEN
      initiator := initiator ∪ {c ↦ a} ||
      queryS := queryS ∪ {c ↦ a} ||
      participant := participant ∪ {c ↦ ar} ||
      queried := queried ∪ {c ↦ a}
      mr := m
   END
```

```
MACHINE middleware0
   send (m) =
      WHEN
         m ∈ MESSAGE
      THEN
         msgset := msgset ∪ {m}
      END
```

Figure 6.14: Query-If: Synchronising of the Fifth Refinement and the Abstract Component Models

of the system model through this synchronisation. The agent role components can now be refined to model the internal reasoning of the agent.

Chapter 7

Case Study : Contract Net

The Query-If case study is an interaction between two agents. The Contract Net case study, presented in this chapter, illustrates the application of the Incremental Development Process to the complex communications that can be involved in interactions between multiple agents. In the contract net interaction protocol the communications are broadcast to multiple agents in the system that may all respond differently.

7.1 Case Study

The contract net interaction protocol has been chosen as a case study as it is commonly used in work on multi-agent systems. This should make it easier for the reader to understand and to compare with other work.

The contract net interaction protocol is a distributed negotiation process (Smith (1980)). The initiator of the protocol advertises the existence of a task that it needs completing by broadcasting a *call for proposals* to find an agent, or group of agents, that offer the most advantageous proposal to carry out a required task. The agents that receive the *call for proposals* can place a bid to complete the task by sending a *proposal.* Participants in the protocol are committed to the bids that they propose. When the initiator selects a bid or a group of bids the participants are informed of the decision and those selected will complete the task. The contract is completed when the participants *inform* the initiator of the protocol that the task is completed or that a *failure* has occurred. The case study has been developed using the FIPA specification of the contract net (FIPA (2002c)).

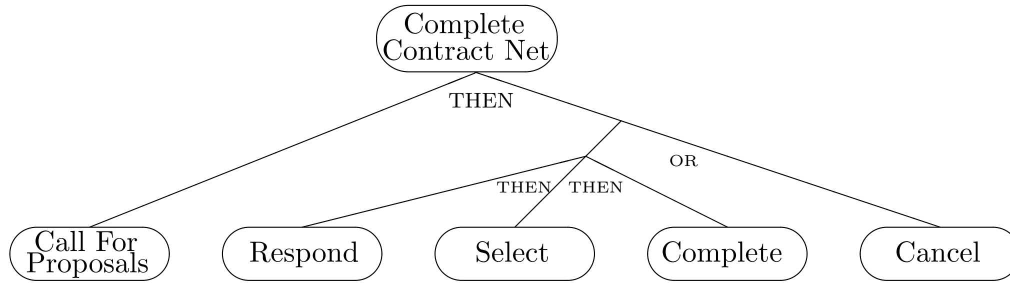

FIGURE 7.1: The First Level of Goal Elaboration for the Contract Net

7.2 Stage One

The first level of goal elaboration for the Contract Net case study is shown in Figure 7.1. To construct the Event-B abstract machine the first level of goal elaboration must be analysed.

The main goal of the system is to complete the contract net interaction. This goal has been elaborated into five goals. The Call For Proposals goal represents an agent initiating the contract net protocol. The Respond goal represents the agents that receive the call for proposals message making a response. The Select goal represents one or more proposals being chosen and the Complete goal represents the agreement being fulfilled. The Cancel goal is required for when the initiating agent changes its intentions and cancels the contract at any point in the interaction. The interaction of the system, as described by the goal elaboration relationships, begins with the Call For Proposals goal, which is followed by the Respond goal, which is followed by the Select goal and finally the Complete goal. The Cancel goal is related to the Respond, Select and Complete goal by an OR relationship and can be fulfilled at any point after the Call For Proposals goal has been fulfilled.

The abstract machine has five events and five state variables that correspond to the five goals in the first level of elaboration of the goal diagram. The guards and invariant conditions are constructed according to the goal elaboration relationships in the goal diagram. The context for the model specifies a deferred set of conversations and a set of agents.

The Contract Net case study involves one agent that interacts with multiple agents to negotiate the contract. Because of this the events are parameterised by a conversation and a relationship between that conversation and a subset of agents, as described in Chapter 4 Section 4.2.5. When the interaction is between two agents the agent that is being communicated to is recorded by the goal variables as there is only one agent that can send or receive each message. Because their are multiple agents involved in the contract net interaction the variables that record the fulfillment of each goal must record the participants. Each participant that has sent or received a message is recorded

allowing more than one of each message to be sent or received by other participants. The initiator will remain the same throughout the interaction and does not need to be recorded at this point.

The `cfp` variable is a relationship between conversations and the agents in the system and represents the goal of the system to make a call for proposals has been fulfilled. The `responded` variable is a subset of the `cfp` variable and represents the state of each agent fulfilling the goal to respond to the call for proposals. The `select` variable is a subset of the `responded` variable and represents the group of responses that have been selected by the initiator of the conversation. The `completed` variable represents the state of the system when the agents that have been selected have fulfilled their commitment. The `cancelled` variable models the initiator changing its goal and cancelling the conversation.

The first event in the interaction of the system is the `callForProposals` event that takes a conversation and relationship between the conversation and agents and moves it into the `cfp` state. The guards $dom(as) = \{c\}$ and $ran(as) \subseteq AGENT$ specify that the domain of the relationship as is the conversation c and that the range is a subset of the $AGENT$ set. The action of the event adds the relationship to the `cfp` state variable, $cfp := cfp \cup as$. The Call For Proposals goal is related to the Respond goal by a THEN relationship. The `respond` event takes a relationship that is in the `cfp` state and not in the `responded` state. The action of the event adds the relationship to the `responded` state. The Respond goal in the goal diagram is related to the Select goal by a THEN relationship. The `select` event takes a subset of the relationships that are in the `responded` state and adds it to the `selected` state. This models the initiator of the conversation making a selection from the responses that it has received. The guard $as \subseteq \{c\} \lhd responded$ specifies the relationship as as a subset of the `responded` variable that has its domain restricted to the contents of the set $\{c\}$. The next goal in the goal diagram is the Complete goal that is to the right of a THEN relationship with the Select goal. The `complete` event takes a subset of relationships that are in the `selected` state, $as \subseteq \{c\} \lhd selected$, and not in the `completed` state, $c \notin dom(completed)$. The action of the event adds the relationships to the `completed` state. The final goal in the diagram is the Cancel goal. The Cancel goal is to the right of the THEN relationship that originates with the Call For Proposals goal and is related to the other goals in the diagram by an OR relationship. The `cancel` event takes a set of relationships that are in the `cfp` state and not in the `cancelled` state and the action of the event adds the relationship to the `cancelled` state.

The guards of the `select` event have been weakened following an iteration of the design. Because the interaction is between multiple agents and how the select goal has been elaborated in the next level of goal elaboration the guard that was originally $c \notin dom(selected)$ was changed to $as \cap selected = \varnothing$. This means that the select

event in the abstract machine would be able to occur more than once. The next level of goal elaboration introduces a new event that prevents this in the refinement model.

```
MACHINE m0 SEES context
VARIABLES
   cfp, responded, selected, completed, cancelled
INVARIANTS
   cfp ∈ CONVERSATION ↔ AGENT              responded, cancelled ⊆ cfp
   selected ⊆ responded                    completed ⊆ selected
EVENTS
  INITIALISATION
     BEGIN
      cfp, responded, selected, completed, cancelled := ∅
     END
    callForProposals                       respond
     ANY c, as WHERE                         ANY c, a WHERE
        c ∈ CONVERSATION                        c ↦ a ∈ cfp
        c ∉ dom(cfp)                            c ↦ a ∉ responded
        as ∈ CONVERSATION ↔ AGENT              THEN
        dom(as) = {c}                           responded := responded ∪ {c ↦ a}
        ran(as) ⊆ AGENT                        END
     THEN
        cfp := cfp ∪ {c}
     END
    select                                 complete
     ANY c, as WHERE                         ANY c, as WHERE
        c ∈ dom(responded)                      c ∈ dom(selected)
        as ⊆ responded                          c ∉ dom(completed)
        as ∩ selected = ∅                       as ⊆ {c} ◁ selected
     THEN                                    THEN
        selected := selected ∪ as                completed := completed ∪ as
     END                                     END
    cancel
     ANY c, as WHERE
        c ∈ dom(cfp)
        c ∉ dom(cancelled)
        as ⊆ {c} ◁ cfp
     THEN
        cancelled := cancelled ∪ as
     END
  END
```

FIGURE 7.2: Contract Net: Abstract Machine

A more detailed model of the system can be obtained by refining the goal diagram. Figure 7.3 shows how the goals have been further elaborated for this case study.

The first refinement of the abstract machine can be constructed by analysing the second level of goal elaboration in the goal diagram. The Respond goal has been elaborated by two sub-goals, Propose and Refuse, that are related with an XOR relationship that creates an exclusive choice between the fulfillment of the goals. The Refuse goal is an endpoint goal. The Select goal has been elaborated by three sub-goals; Select THEN Accept XOR Reject. The Select goal represents the system choosing one or more of the proposals. The Accept goal represents the system accepting proposals that have been selected. The Reject goal represents the system rejecting the proposals that have not

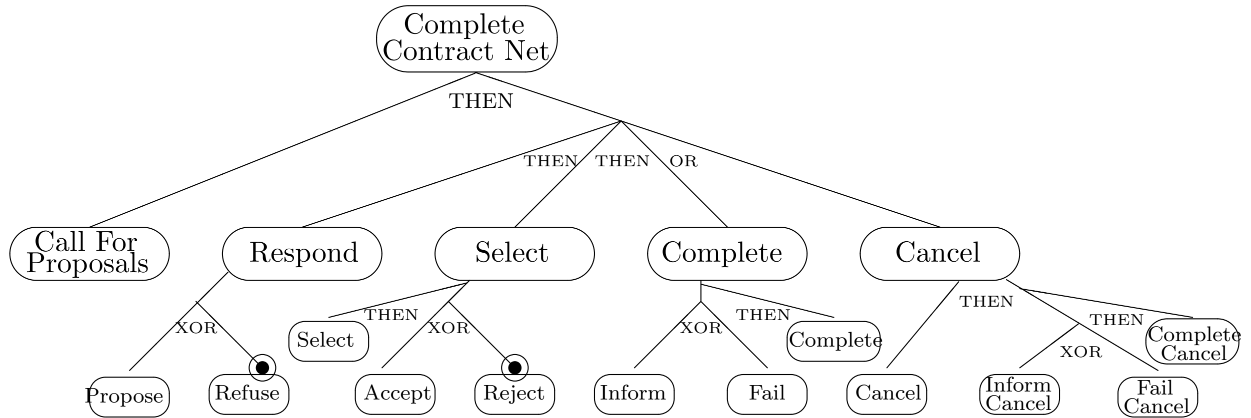

FIGURE 7.3: The Second Level of Goal Elaboration for the Contract Net

been selected. The Reject goal is also an endpoint goal. The Complete goal has been elaborated by the three sub-goals; (Inform XOR Fail) THEN Complete. The Inform and Fail goals represent the different agents reporting the success or failure of their accepted tasks. The Complete goal represents the system deciding that the interaction has been completed either successfully or unsuccessfully. The Cancel goal has been elaborated by a set of four sub-goals. The Cancel sub-goal represents the decision to cancel the interaction. This is followed in a THEN relationship by the two sub-goals Inform Cancel and Fail Cancel that are related by an XOR relationship. The Inform Cancel and Fail Cancel goals represent the agents responding to the request to cancel the conversation. The Cancel Complete goal models the conversation being successfully or unsuccessfully cancelled. The relationships between the elaborated goals create a more detailed model of the system interaction. The interaction is started, as before, with the Call For Proposals goal being fulfilled. This is followed, through the THEN relationship from the first level of goal elaboration, by either the Propose or Refuse goal for each of the agents involved in the conversation. The Propose goal will be followed by the Select goal. The Select goal will be followed by the Accept and Reject goal with each agent that proposed being either accepted or rejected. The fulfillment of the Accept goal will be followed by either the Inform or Fail goal. The interaction is completed at this point with the Complete goal, unless the Cancel goal has been fulfilled at any point after the Call For Proposal goal. In this case either the Inform Cancel or the Fail Cancel goal need to be fulfilled followed by the Cancel Complete goal before the interaction is finished.

The introduction of the endpoint goals at this level of refinement restricts the behaviour of the model from the abstract machine. The refinement model will be able to finish an interaction before the abstract machine.

The second level of goal elaboration in the goal diagram can be used to construct

$$
\begin{array}{ll}
refused,\ proposed \subseteq cfp & refused \cap proposed = \varnothing \\
selected1 \subseteq proposed & \\
accepted,\ rejected \subseteq selected1 & accepted \cap rejected = \varnothing \\
informed,\ failed \subseteq accepted & informed \cap failed = \varnothing \\
cancelledStarted \subseteq cfp & \\
informCancelled,\ failCancelled \subseteq cancelledStarted & \\
informCancelled \cap failCancelled = \varnothing & \\
responded = refused \cup proposed & \\
selected = accepted \cup rejected &
\end{array}
$$

FIGURE 7.4: Contract Net: Invariants of the First Refinement

an Event-B refinement of the abstract machine. This refinement is shown in Figure 7.4 and Figure 7.5. The Call For Proposals goal has not been elaborated so the `callForProposals` event and `cfp` variable remain the same as in the abstract machine. The abstract `responded` variable and `respond` event have been refined by the variables and events for the Propose and Refuse goals. The variables are specified as subsets of the `cfp` variable. The events are parameterised by a relationship between a conversation and agent that are in the `cfp` variable and not in either the `proposed` or `refused` variables. The action of the events adds the relationship to the appropriate variable e.g. $proposed := proposed \cup \{c \mapsto a\}$. The Accept and Reject goals refine the abstract Select goal. Because of this the event and variable for the Select sub-goal are new variables and have been named `select1` and `selected1`, respectively, to prevent any naming clashes in the tools. Because Refuse and Reject are both endpoint goals the `selected1` variable is specified as a subset of the `proposed` variable and the `informed` and `failed` variables are specified as subsets of the `accepted` variable. The `select1` event is parameterised by a conversation and a subset of the proposed variable for that conversation, $as \subseteq \{c\} \lhd proposed$. The action of the event adds as to the `selected1` variable. The `select1` event is a new event and the selection by the initiator is now modelled as occurring once because of the guard $c \notin dom(selected1)$. The `accept` and `reject` events refine the abstract `select` event because they are to the right of the THEN relationship. The `accept` event takes the relationships that were added to the `selected1` variable in the `select1` event, $as = \{c\} \lhd selected1$, and adds them to the `accepted` variable. The `reject` event takes the relationships that are in the `proposed` variable, but not in the `selected1` variable, $as = (\{c\} \lhd proposed) \setminus (\{c\} \lhd selected1)$. Both the `accept` and `reject` events can occur because the guard of the abstract `select` event was weakened in the abstract machine. The `inform` and `fail` events both take a relationship that is in the `accepted` variable and not in either the `informed` or `failed` variables. The action of the events add the relationship to the appropriate variable. The `complete` event refines the abstract `complete` event by adding a subset of the `inform` and `failed` variables to the `completed` variable, $as \subseteq \{c\} \lhd (informed \cup failed)$. The Cancel goal has been elaborated by four sub-goals that can be described by their relationships as *Cancel THEN ((Inform Cancel XOR Fail Cancel) THEN Cancel Complete)*. The first THEN relationship means that a new event and variable are introduced for the

Cancel sub-goal. The second THEN relationship means that new events and variables are introduced for the Inform Cancel and Fail Cancel sub-goals that are linked by the XOR relationship. The event for the Cancel Complete goal refines the abstract `cancel` event. The variable for the Cancel sub-goal is specified as a subset of the `cfp` variable because of the THEN relationship between the abstract goals. The new event, `cancel1`, for the Cancel sub-goal selects a relationship between a conversation and agents that is in the `cfp` variable and not in the new `cancelStarted` variable and adds it to the `cancelStarted` variable. The variables for the Inform Cancel and Fail Cancel sub-goals are specified as subsets of the `cancelStarted` variable because of the THEN relationship. The XOR relationship between the Inform Cancel and the Fail Cancel sub-goals requires them to be specified as disjoint. The events for the Inform Cancel and Fail Cancel sub-goals select a relationship that is in the `cancelStarted` variable and not in the state variable for the opposing sub-goal and adds it to the state variable for the sub-goal. The `cancelled` event adds a subset of the relationships from the `informCancelled` variable to the `cancelled` variable, $as \subseteq \{c\} \lhd (informCancelled \cup failCancelled)$.

```
callForProposals REFINES callForProposals
  ANY  c, as  WHERE
    c ∈ CONVERSATION
    c ∉ dom(cfp)
    as ∈ CONVERSATION ↔ AGENT
    dom(as) = {c}
    ran(as) ⊆ AGENT
  THEN
    cfp := cfp ∪ as
  END
propose  REFINES  respond
  ANY  c, a  WHERE
    c ↦ a ∈ cfp
    c ↦ a ∉ proposed
    c ↦ a ∉ refused
  THEN
    proposed := proposed ∪ {c ↦ a}
  END

accept  REFINES  select
  ANY  c, as  WHERE
    c ∈ dom(selected1)
    c ∉ dom(accepted)
    as = {c} ◁ selected1
    as ∩ rejected = ∅
  THEN
    accepted := accepted ∪ as
  END
inform
  ANY  c, a  WHERE
    c ↦ a ∈ accepted
    c ↦ a ∉ informed
    c ↦ a ∉ failed
  THEN
    informed := informed ∪ {c ↦ a}
  END
complete  REFINES  complete
  ANY  c, as  WHERE
    c ∈ dom(informed ∪ failed)
    as ⊆ {c} ◁ (informed ∪ failed)
    c ∉ dom(completed)
  THEN
    completed := completed ∪ as
  END
informCancel
  ANY  c, a  WHERE
    c ↦ a ∈ cancelled1
    c ↦ a ∉ informCancelled
    c ↦ a ∉ failCancelled
  THEN
    informCancelled := informCancelled ∪ {c ↦ a}
  END
cancelled  REFINES  cancel
  ANY  c, as  WHERE
    c ∈ (informCancelled ∪ failCancelled)
    as ⊆ {c} (informCancelled ∪ failCancelled)
    c ∉ dom(cancelled)
  THEN
    cancelled := cancelled ∪ as
  END
```

```
refuse  REFINES  respond
  ANY  c, a WHERE
    c ↦ a ∈ cfp
    c ↦ a ∉ refused
    c ↦ a ∉ proposed
  THEN
    refused := refused ∪ {c ↦ a}
  END

select1
  ANY  c, as WHERE
    c ∈ dom(proposed)
    c ∉ dom(selected1)
    as ⊆ {c} ◁ proposed
    c ∉ cancelled1
  THEN
    selected1 := selected1 ∪ as
  END
reject  REFINES  select
  ANY  c, as  WHERE
    c ∈ dom(selected1)
    c ∉ dom(rejected)
    as = ({c} ◁ proposed) \ ({c} ◁ selected1)
    as ∩ accepted = ∅
  THEN
    rejected := rejected ∪ as
  END
fail
  ANY  c, a  WHERE
    c ↦ a ∈ accepted
    c ↦ a ∉ failed
    c ↦ a ∉ informed
  THEN
    failed := failed ∪ {c ↦ a}
  END
cancel1
  ANY  c, as  WHERE
    c ∈ dom(cfp)
    c ∉ dom(cancelStarted)
    as ⊆ {c} ◁ cfp
  THEN
    cancelStarted := cancelStarted ∪ as
  END
failCancel
  ANY  c, a  WHERE
    c ↦ a ∈ cancelled1
    c ↦ a ∉ failCancelled
    c ↦ a ∉ informCancelled
  THEN
    failCancelled := failCancelled ∪ {c ↦ a}
  END
```

FIGURE 7.5: Contract Net: Events of the First Refinement

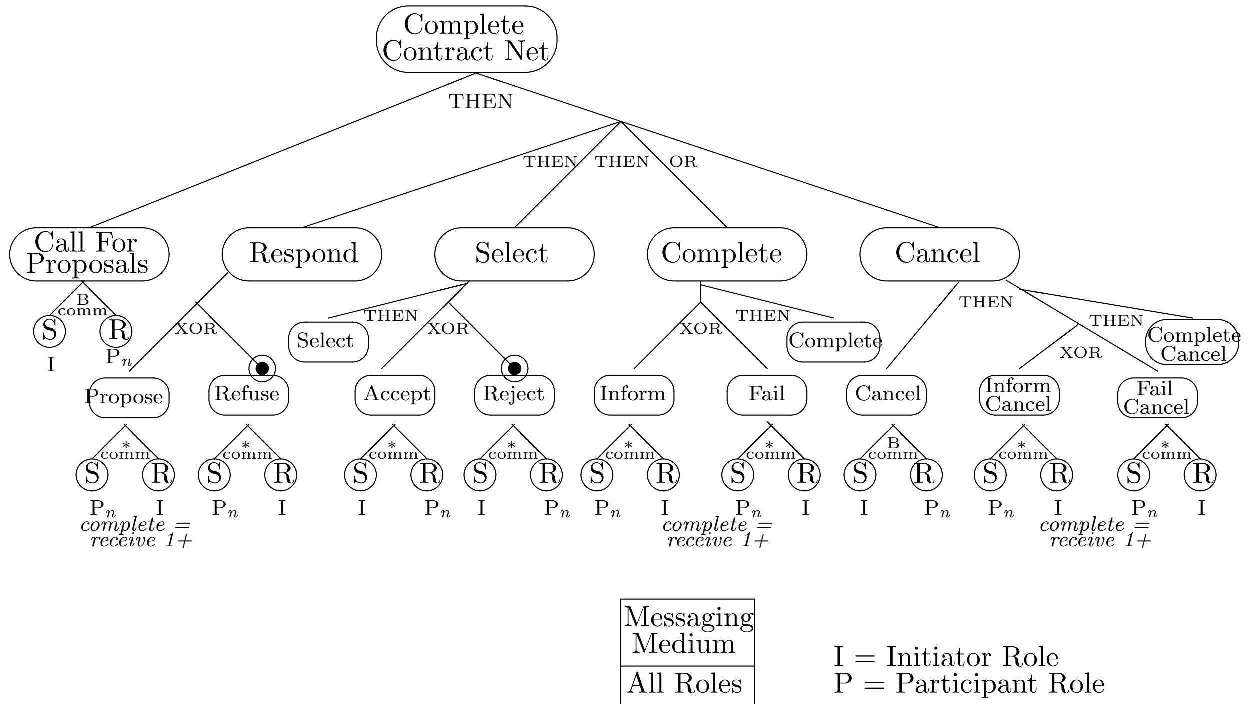

FIGURE 7.6: Goal Model for Contract Net

The next section describes Stage Two of the Process that models the agent roles and their communications as refinements of the system models.

7.3 Stage Two

The second stage of the Process elaborates the goal diagram with communicating goals. The Contract Net case study is an interaction protocol that involves multiple agents at all of the stages of the interaction. The communicating goals elaborated in the goal diagram are broadcast goals. The roles that can be identified for the case study are the initiator of the contract net and the participants. There will be one agent that takes the initiator role for each interaction and multiple agents will take the participant role. A messaging medium resource is required for the agents to communicate.

Figure 7.6 shows the goal diagram that includes the elaboration to the communicating goals. Because of space restrictions the send and receive goals shown in Figure 7.6 have been labelled S for the send goals and R for the receive goals. Some of the sets of sub-goals can occur multiple times and some have been given completion conditions. The completion condition *complete = receive 1+* means that the sub-goals can be considered completed if they have been fulfilled at least once.

The communication goals all involve broadcast relationships and those that are related with a *COMM relationship will require both the send and receive goals to be able to

occur multiple times. Part of the specification of the third refinement is shown below to describe how the elaboration of the goal diagram can be translated into the Event-B model. The full specification of this refinement model can be found in Appendix B.

Figure 7.7 shows the invariant conditions for the first three sets of communicating goals and the role variables. The variables for the communicating goals are suffixed with an 'S' for the send goals and an 'R' for the receive goals. The communicating goals for the Contract Net are all broadcast goals and are specified as many-to-many relationships between `CONVERSATION` and `AGENT`. The COMM relationship requires that the receive goals are all specified as subsets of the send goals. The BCOMM relationship between the communicating goals for the abstract Call For Proposals goal allows the send goal to occur once and the receive goal to occur multiple times. This means that the refinement relationship with the abstract event should be with the send event. The abstract goals can all be allocated to the agent responsible for the send goals. In addition to the specification of the communicating goals for this refinement the agent roles need to be modelled. The roles identified in the goal diagram are the initiator and the participants. The agents that will perform the roles are identified when the call for proposals is first made. The domain of the role variables is specified as the domain of the `cfpS` variable. There is one initiator for each conversation and this is specified as a total function between the domain of `cfpS` and `AGENT`. There can be multiple participants in a conversation and the role variable for participant is specified as a relationship between the domain of `cfpS` and `AGENT`.

$$
\begin{array}{l}
cfpS \subseteq cfp \\
cfpR \subseteq cfpS \\
proposeS \subseteq proposed \\
proposeR \subseteq proposeS \\
refuseS \subseteq refused \\
refuseR \subseteq refuseS \\
initiator \in dom(cfpS) \rightarrow AGENT \\
participant \in dom(cfpS) \leftrightarrow AGENT \\
proposeS,\ refuseS \subseteq cfpR
\end{array}
$$

FIGURE 7.7: Contract Net: State of the Second Refinement

The events shown in Figure 7.8 model the Contract Net case study up to the point where proposals and refusals have been received. The events for the communicating goals have additional variables that represent the agent involved in the stage of the conversation. The role variables are updated in the `sendCfp` event because it represents the left-most set of communicating goals in which both roles are involved. The required role for the agent involved in the event is specified as part of the guards for the remaining events. The `receiveCfp` event can occur multiple times and is introduced as a new event. The send events for the Propose and Refuse communicating goals have a guard that upholds the abstract XOR relationship e.g. $c \mapsto a \notin refused$. The completion conditions specified in the goal diagram for several of the communicating goals require at least one relationship to be present in the variables for the receive goals. This condition for the

Propose communicating goals is shown in the guard for the select event that specifies $c \in dom(proposeR)$, ensuring that there is at least one relationship in the `proposeR` variable with the conversation in the domain.

```
sendCfp  REFINES  callForProposals
  ANY  c, as, a  WHERE
    c ∈ CONVERSATION
    c ∉ dom(cfpS)
    as ∈ CONVERSATION ↔ AGENT
    a ∈ AGENT
    dom(as) = {c}
    ran(as) ⊆ AGENT \ {a}
  THEN
    cfp := cfp ∪ as
    cfpS := cfpS ∪ as
    initiator := initiator ∪ {c ↦ a}
    participant := participant ∪ as
  END

receiveCfp
  ANY  c, a  WHERE
    c ↦ a ∈ cfpS
    c ↦ a ∉ cfpR
    c ↦ a ∈ participant
  THEN
    cfpR := cfpR ∪ {c ↦ a}
  END

sendRefusal REFINES  refuse
  ANY  c, a  WHERE
    c ↦ a ∈ cfpR
    c ↦ a ∉ refused
    c ↦ a ∉ proposed
    c ↦ a ∈ participant
  THEN
    refused := refused ∪ {c ↦ a }
    refuseS := refuseS ∪ {c ↦ a}
  END

sendProposal  REFINES  propose
  ANY  c, a  WHERE
    c ↦ a ∈ cfpR
    c ↦ a ∉ proposed
    c ↦ a ∉ refused
    c ↦ a ∈ participant
  THEN
    proposed := proposed ∪ {c ↦ a}
    proposeS := proposeS ∪ {c ↦ a}
  END

receiveProposal
  ANY  c, a  WHERE
    c ↦ a ∈ proposeS
    c ↦ a ∉ proposeR
    c ↦ a ∈ participant
  THEN
    proposeR := proposeR ∪ {c ↦ a}
  END

receiveRefusal
  ANY  c, a  WHERE
    c ↦ a ∈ refuseS
    c ↦ a ∉ refuseR
    c ↦ a ∈ participant
  THEN
    refuseR := refuseR ∪ {c ↦ a}
  END

select  REFINES  select1
  ANY  c, as  WHERE
    c ∈ dom(proposeR)
    as ⊆ {c} ◁ proposeR
    c ∉ dom(selected1)
  THEN
    selected1 := selected1 ∪ as
  END
```

Figure 7.8: Contract Net: Selected Events From the Second Refinement

All of the events for this stage of the Process have been introduced to the model. The resources identified for the system need to be introduced and then the model needs to be decomposed into the individual component models.

The introduction of the messaging medium resource adds a message record to the context of the model along with fields for the sender, receiver and conversation for the message record. The fourth refinement of the Contract Net case study introduces variables for each type of message being sent that can be used to pass the messages between the events. The information that is accessible as part of each message record allows the

shared variables to be encapsulated by the agent roles that have been allocated the events. Two further role variables can be introduced so each of the agent role components will have a record of the different roles of the agents that they are interacting with.

An extract from the state of the third refinement is shown in Figure 7.9. It shows the specification of the first refinement of the messaging medium for the exchange of call for proposals messages. A variable, `cfpM`, is specified as a subset of MESSAGE. Another invariant uses the join operator to allow the `cfpS` variable to be encapsulated by the agent role that sends the message and the messaging medium variable to be used for coordination.

$$
\begin{array}{l}
cfpM \ \subseteq \ MESSAGE \\
cfpS \ = \ ((cfpM \lhd \ messageConversation)^{-1} \ ; \ receiver) \\
pInitiator \ \in \ dom(cfpR) \ \cup \ dom(cancelR) \ \rightarrow \ AGENT \\
pParticipant \ \in \ dom(cfpR) \ \cup \ dom(cancelR) \ \leftrightarrow \ AGENT \\
pInitiator \ \subseteq \ initiator \\
initiator \ = \ ((cfpM \ \cup \ cancelM) \lhd \ messageConversation)^{-1}; \ sender \\
pParticipant \ \subseteq \ participant \\
participant \ = \ ((cfpM \ \cup \ cancelM) \lhd \ messageConversation)^{-1}; \ receiver)
\end{array}
$$

FIGURE 7.9: Contract Net: Extract From the State of the Third Refinement

Because messages are now being sent and more information is available to each agent the information about the roles of each agent can be encapsulated within each agent. Two more variables that record the role information need to be specified so each role has a record of their role and the agents that perform the opposing roles. In the case study the original role records are encapsulated in the initiator role. The two new variables, `pInitiator` and `pParticipant`, are encapsulated in the participant role. The invariant conditions specify the new role variables as a subset of the original role variables. The roles can be recorded by the participant on the receipt of a call for proposals message. A further invariant for each new role variable, using the join operator, relates the original role variable to the fields of a message e.g. $participant = ((cfpM \cup cancelM) \lhd messageConversation)^{-1}; \ receiver)$. Because it would be possible for a cancel message to be received before the call for proposals message the `cancelM` variable is included in the domain restriction for the invariant conditions. It can then be proven that the information added to the new role variable on the receipt of a call for proposals message is already held in the original role variable and the subset relationship is upheld.

Figure 7.10 shows the `sendCfp` and `receiveCfp` events from the fourth refinement. Event variables have been added to represent the messages being sent and received. When a message is broadcast the guards of the events use a relational image of the message fields to restrict the values of a group of messages e.g. $messageConversation[ms] = \{c\}$. The relational image of a relationship, e.g. $r = \{\{x, y\}, \{x, z\}\}$, produces the range of the relationship where the domain is the given set, e.g. $r[\{x\}] = \{y, z\}$. The `cfpM` variable is used to pass messages between the events and the `cfpS` variable is no longer

accessed in the `receiveCfp` event. The original role variables are no longer accessed by the `receiveCfp` event and the new role variables, `pInitiator` and `pParticipant`, are updated by the information in the message record by the action of the event.

```
sendCfp REFINES sendCfp
  ANY  c, as, a, ms  WHERE
    c ∈ CONVERSATION
    c ∉ dom(cfp)
    as ∈ CONVERSATION ↔ AGENT
    dom(as) = {c}
    a ∈ AGENT
    ran(as) ⊆ AGENT \ {a}
    ms ⊆ MESSAGE
    sender[ms] = {a}
    receiver[ms] = ran(as)
    messageConversation[ms] = {c}
  THEN
    cfp := cfp ∪ as
    cfpS := cfpS ∪ as
    initiator := initiator ∪ {c ↦ a}
    participant := participant ∪ as
    cfpM := cfpM ∪ ms
  END
```

```
receiveCfp REFINES receiveCfp
  ANY  c, a, m  WHERE
    m ∈ cfpM
    c ↦ a ∉ cfpR
    receiver(m) = a
    messageConversation(m) = c
  THEN
    cfpR := cfpR ∪ {c ↦ a}
    pParticipant := pParticipant ∪ {c ↦ a}
    pInitiator := pInitiator ∪ {c ↦ sender(m)}
  END
```

FIGURE 7.10: Contract Net: Example Events of the Fourth Refinement

The fifth refinement model requires the context for the model to be extended to include a specification of the message type. As with the Query-If case study the different types of messages have been declared as disjoint constant subsets of the message record. Figure 7.11 shows the extended context.

```
CONSTANTS
   CFP, PROPOSE, REFUSE, ACCEPT, REJECT, INFORM, FAIL, CANCEL, INFORMCANCEL, FAILCANCEL
AXIOMS
   CFP, PROPOSE, REFUSE, ACCEPT, REJECT, INFORM, FAIL, CANCEL,
   INFORMCANCEL, FAILCANCEL ⊆ MESSAGE
   disjoint(CFP, PROPOSE, REFUSE, ACCEPT, REJECT, INFORM, FAIL, CANCEL,
   INFORMCANCEL, FAILCANCEL)
```

FIGURE 7.11: Contract Net: Extended Context for the Fifth Refinement

An extract from the state of the fifth refinement is shown in Figure 7.12. A single variable, `msgset`, is specified to replace the individual messaging mediums from the abstract model. The gluing invariant relates the abstract `cfpM` variable to all of the messages in `msgset` that are also in the type `CFP`.

The events of the refinement have an additional guard that specifies the type of message and the action of the events are refined to add the messages to the single messaging medium. The `sendCfp` and `receiveCfp` events from the fifth refinement are shown in Figure 7.13.

The variables for the roles have been encapsulated and the messaging medium resource has been specified. The system model can now be decomposed into the components

```
VARIABLES
   cfpS, cfpR, msgset
INVARIANTS
      msgset ⊆ MESSAGE
      cfpM = CFP ∩ msgset = ∅
```

FIGURE 7.12: Contract Net: State of the Fifth Refinement

```
sendCfp REFINES sendCfp
  ANY c, as, a, ms WHERE
    c ∈ CONVERSATION
    c ∉ dom(cfp)
    as ∈ CONVERSATION ↔ AGENT
    a ∈ AGENT
    dom(as) = {c}
    ran(as) ⊆ AGENT \ {a}
    ms ⊆ MESSAGE
    sender[ms] = {a}
    receiver[ms] = ran(as)
    ms ⊆ CFP
    messageConversation[ms] = {c}
  THEN
    cfp := cfp ∪ as
    cfpS := cfpS ∪ as
    initiator := initiator ∪ {c ↦ a}
    participant := participant ∪ as
    msgset := msgset ∪ ms
  END
```

```
receiveCfp REFINES receiveCfp
  ANY c, a, m WHERE
    m ∈ msgset
    c ↦ a ∉ cfpR
    receiver(m) = a
    messageConversation(m) = c
    m ∈ CFP
  THEN
    cfpR := cfpR ∪ {c ↦ a}
    pParticipant := pParticipant ∪ {c ↦ a}
    pInitiator := pInitiator ∪ {c ↦ sender(m)}
  END
```

FIGURE 7.13: Contract Net: Example Events of the Fifth Refinement

for the agent roles and the messaging medium. The decomposition creates abstract machines of the components and a refinement of the system model that synchronises the events of the components to model the system functionality.

Figure 7.14 shows the `sendCfp` event from the refinement model and the events from the initiator role abstract machine and messaging medium abstract machine components that are used to fulfill the event functionality.

The component for the messaging medium differs for this case study by including an extra event for broadcasting that adds a subset of messages to the messaging medium.

The `sendCfp` event from the sixth refinement declares a variable `m` that is assigned the output from the `sendCfp` event in the `initiator` component model and then sent as a parameter to the `bcast` event in the `middleware` component model.

```
REFINEMENT m6  REFINES   m5
  sendCfp  =
    VAR  m  IN
      m  ←  initiator.sendCfp;
      mw.bcast(m)
    END
```

```
MACHINE initiator0
ms1 <--  sendCfp  =
    ANY  c, as, a, ms  WHERE
      c  ∈  CONVERSATION  ∧
      as  ∈  CONVERSATION  ↔  AGENT  ∧
      dom(as)  =  {c}  ∧
      a  ∈  AGENT  ∧
      ran(as)  =  AGENT  \  {a}  ∧
      c  ∉  dom(cfp)  ∧
      ms  ⊆  MESSAGE  ∧
      sender[ms]  =  {a}  ∧
      receiver[ms]  =  ran(as)  ∧
      messageConversation[ms]  =  {c}  ∧
      m  ⊆  CFP
    THEN
      cfp  :=   cfp  ∪  as  ||
      cfpS  :=   cfpS  ∪  as  ||
      initiator  :=  initiator  ∪  {c  ↦  a}  ||
      participant  :=  participant  ∪  as  ||
      ms1  :=  ms
    END
```

```
MACHINE middleware0
 bcast (ms) =
    WHEN
      ms  ⊆  MESSAGE
    THEN
      msgset  :=  msgset  ∪  ms
    END
```

Figure 7.14: Contract Net: Synchronising the Sixth Refinement Model and the Abstract Component Models

Chapter 8

Fault-Tolerance Modelling Patterns for Multi-Agent Systems

This chapter introduces a set of patterns for modelling fault-tolerance in Event-B models of multi-agent systems. How fault-tolerance can be modelled in multi-agents systems is discussed followed by how modelling patterns can be used in Event-B. A modified version of the Contract Net case study is introduced. Each of the patterns are described and provide an example pattern based on the Contract Net case study. Related work on the use of patterns in Event-B and in multi-agent system development is discussed.

Design patterns are a method of capturing and communicating design expertise that can be re-used in further designs. The fault-tolerance of a system is its ability to manage faults that will enable it to continue to function as it was designed. A set of fault-tolerance patterns have been developed to help specify fault-tolerance in Event-B models of multi-agent systems. The Contract Net case study will be used to illustrate the application of the patterns. The FIPA specification that was used as inspiration for the design of the Contract Net case study includes stages of the interaction that are designed to cope with faults that arise during multi-agent negotiation. This means that some of the identified patterns are already part of the case study specification. The specification of these fault-tolerance techniques was used as the starting point for several of the identified patterns.

The fault-tolerance patterns consist of three elements. Each of the elements can be used separately. Their separation should make it possible to apply the patterns to other event-based formal specification methods. The formal elements make the pattern more specific to the Event-B method. An Event-B extract from the Contract Net case study provides a detailed example of how the pattern can be applied to an Event-B development. The extract specifications have the potential to be re-used in other developments. The patterns model an abstraction of fault-tolerance and are intended to ensure that the

faults and the methods for coping with the faults are included in the specification of a multi-agent system.

8.1 Fault-Tolerance in Agent Interaction

A fault-tolerant system is one that can continue to function as it was designed in the presence of faults (Anderson and Lee (1981)). A multi-agent system has to be able to cope with the faults that can occur in any distributed system. Alongside the fault-tolerance requirements of a distributed system the complexity of agent interactions raises further requirements. A multi-agent system needs to be able to manage the communications required for the agents to coordinate their knowledge and actions. A multi-agent system also needs to be able to cope with the complex behaviour of autonomous agents.

Fault-tolerance in distributed systems requires that the system can cope with faults in communication and faults in the behaviour of the distributed components. The system must be able to continue to function if there is a failure in communication between nodes or if a node fails and ceases to communicate. The system must also be able to cope if a node in the system is prevented from completing a task it has been delegated.

Fault-tolerance in multi-agent systems also requires that the system can cope with the rational and autonomous behaviour of agents, which can make their behaviour unpredictable, and the many dynamic interactions required for the system to function. Rational agents will stop pursuing a goal if they believe that the goal has already been achieved or that it cannot be achieved. An agent that is autonomous is not required to complete any tasks requested by other agents. The task may conflict with its existing goals and, therefore, not be desirable for the agent to complete. The heterogeneity and dynamic interactions of a multi-agent system may lead to agents receiving messages that they do not understand or that are out of expected order. The agents must be able to handle such faults in interactions and communicate their reactions to these faults.

The fault-tolerance described above ensures that an agent will continue to be able to provide a service regardless of any failures in a particular interaction. A conversation between two agents may lead to the failure of an agent to fulfill a goal, but that agent will still be able to perform its role in another conversation. Patterns for fault-tolerance at the conversation level, e.g. having a replicated agent that will continue to operate in place of a failed agent, is a possible subject for future work.

Types of patterns and collections of patterns are described in Chapter 5 Section 5.7. The patterns presented in this chapter are not a catalogue because they offer solutions to one aspect of design rather than a selection of aspects. They are not a pattern language because they cannot be combined independently from a development model and offer no alternative solutions. The patterns are most similar to refinement patterns. They are

different in that they provide an abstract pattern that needs to be applied to, or already present in, the abstract machine of the development and are not intended to be applied solely as a refinement step, but integrated into the refinement chain.

8.2 Fault-Tolerance Patterns for Event-B Models of Multi-Agent Systems

There is an issue that arises when developing patterns for a refinement-based development method. For the pattern to be applied in a refinement there must be some abstraction of its function in the abstract model of the system. Without this the mechanism introduced by the pattern it will be difficult to integrate into the basic functioning of the system model. An alternative is to create patterns that can be applied only at the abstract machine level.

Fault-tolerance is not necessarily a feature of a system that is appropriate to model in detail at the most abstract level. It is often a part of the communication infrastructure or a component of individual nodes and, therefore, will be modelled in refinement. The patterns presented in this chapter include an example Event-B specification from a refinement model.

Removing the concept of refinement in an Event-B modelling pattern will limit the usefulness of the pattern for providing a complete solution. Including a complete refinement chain as part of the pattern would predefine the possible refinement steps required for the pattern to be applied. A developer may not be able to then apply the pattern to a development with a dissimilar refinement chain. Providing an example specification of a refinement and not providing support for modelling an abstraction will leave the developer with the task of finding an abstraction, and then refining it to the refinement level at which the pattern has been applied. This approach would only provide an incomplete solution of the pattern and may lead to the incorrect application of the pattern.

Each pattern includes a description, interaction diagram and Event-B extracts from the Contract Net case study. The description for each of the patterns includes a name, problem statement and solution statement. The description can be applied to any event-based specification. The problem statement outlines the issues that are present in a multi-agent system for which the application of the pattern will model a solution. The solution describes that steps that can be taken to solve the problem in an event-based specification. The interaction diagrams show how the different agent roles in the system interact to perform the fault-tolerant behaviour. Several of the patterns are accompanied by diagrams that show the variations required for one-to-many interaction. The Event-B extracts include an abstraction of the pattern and a more concrete pattern that are examples taken from the Contract Net case study. The abstraction of the pattern

will need to be present in the Event-B abstract machine. The developer may need to make intermediate refinement steps between the abstract pattern and the more concrete pattern. The pattern is an extract from a refinement model and the refinement chain of the model may need to be adapted to integrate the application of the pattern.

8.3 Applying the Patterns

Figure 8.1 shows how the patterns can be applied to the refinement chain of an existing Event-B model. The extends relationship shown is similar to those found in Back (2005), but has not been formally defined. The extends relationship requires the addition to, or modification of, the events and variables in the model for the interaction specified in the pattern to be included in the extended model. If the events and variables required for the pattern already exist in the extended model no additions or modifications are necessary. The Event-B examples include gluing invariants that relate the abstract variables to the concrete variables. These can potentially be re-used in the extended refinement chain to help the developer specify the refines relationship.

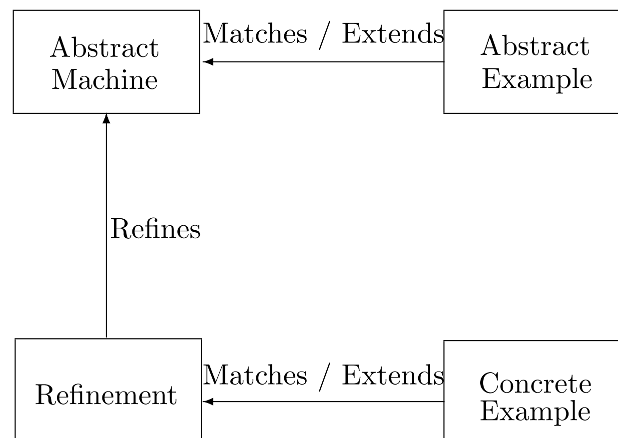

FIGURE 8.1: Applying the Patterns to an Existing Model

The patterns have each been applied in separate developments to the initial chain. This ensures that there is no dependency between the patterns and, therefore, the order in which they are applied has no importance. All of the patterns have also been applied sequentially to the initial chain. This is to provide assurance that there are no conflicts between the patterns. Figure 8.2 illustrates how a collection of patterns can be applied to an Event-B refinement chain. Applying $Pattern_i$ to the initial chain produces $Chain2$ and applying $Pattern_j$ to $Chain2$ produces $Chain3$. $Pattern_j$ could be applied before $Pattern_i$ to produce the same result ($Chain3$). A possible direction for future work would be to find a method to prove the orthogonality of the patterns.

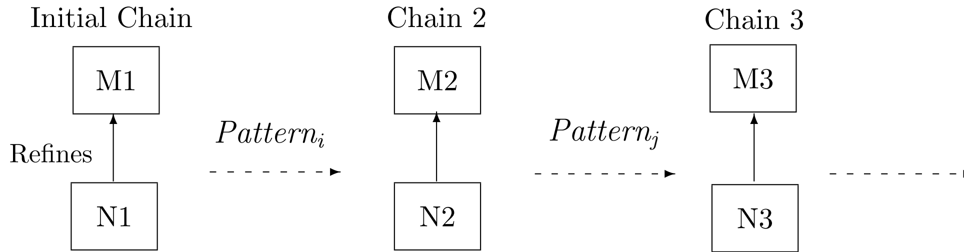

FIGURE 8.2: Effect of Applying Patterns

The selection of patterns provided in this chapter have been chosen because they sufficient and necessary to provide a basic level of fault-tolerance for each interaction in a multi-agent system. This fault-tolerance will allow an agent to continue to function despite any faults that may arise in a particular conversation. The patterns were selected by reviewing the interaction protocols provided by FIPA and analysing the potential faults in the different stages of the interactions. The potential faults found from the analysis are:

1. The agent controlling the conversation changes its goals and requires the conversation to be cancelled.
2. A participating agent stops responding.
3. A participating agent refuses to participate.
4. There are no available participants.
5. A participating agent cannot complete a task that it has committed to.
6. Arbitrary messages / Protocol error.

Fault-tolerance for the first fault is provided by the Cancel pattern. Applying the Cancel pattern to a model of a multi-agent system adds an interaction to the model that allows the agent that controls the conversation to cancel an interaction. The fault-tolerance for the second and fourth fault is provided by the Timeout pattern. Applying the Timeout pattern will add a deadline to the model and events to allow the dealine to be enforced. This pattern may be required to be applied several times in a complex interaction. The third fault is provided for by the Refuse pattern. This pattern allows an agent to refuse a request by adding to the model the events required for the agent to make the refusal. The fifth potential fault is dealt with by the Failure pattern. Applying the Failure pattern provides events in the model to allow the participant to inform the requesting agent of a failure allowing the agent to then take steps to recover from that failure. Fault-tolerance for the final fault can be provided by the Not-Understood pattern. The pattern provides an interaction that informs both parties of the fault that has occurred.

The set of patterns described in this chapter cover all of the potential faults in multi-agent interaction as found in the analysis described above. Each pattern does not influence the application of the other patterns and this allows a pattern to be applied several times, if required, to a model of an interaction. All of the patterns may not be needed in all interactions. They can be applied individually and each pattern does not require the application of the other patterns. This allows the developer to apply only the patterns that are sufficient for the particular development.

8.4 Initial Development Chain for the Contract Net

To present the fault-tolerance patterns a simplified version of the Contract Net case study is used as an initial chain. The patterns will each be applied to the model in the following sections.

The development presented here includes an abstract model and one refinement model. The abstract model models conversations between agents and the refinement introduces the agents involved in the conversation to the model. Initially only successful conversations of the contract net interaction protocol are modelled. The abstract model shown in Figure 8.3 includes four variables that represent states that the conversation will move through. The variables are not modelled as disjoint sets. Instead, the order of the conversation is enforced by specifying the variable for each state as a subset of the previous state. The `cfp` variable represents the state after a call for proposals has been initiated by an agent. The `responded` variable represents the participating agents responding to the call for proposals. The `selected` variable represents the initiator choosing one or more proposals to accept. The `informed` variable models the state where the selected agents have informed the initiator of the successful completion of the task.

The events of the abstract machine move the conversation through the different states as the conversation progresses. The `callForProposals` event adds a conversation to the `cfp` state. The `respond` event takes a conversation that is in the `cfp` state and puts it in the `responded` state. The `select` event takes a conversation that is in the `responded` state and adds it the the `selected` state. The `inform` event takes a conversation that is in the `selected` state and adds it to the `informed` state to complete the conversation.

The refinement of the abstract model incorporates the interaction between the agents that are involved in the conversation. The invariants for the refinement model are shown in Figure 8.4. The variables of the model represent messages being sent and received by the agents in the system. The variables that represent a message being sent are suffixed with an 'S' and those that represent a message being received are suffixed with an 'R'. The conversation is between multiple agents and so the variables are specified as relationships between a set of conversations and a set of agents. For example, $c \mapsto a \in cfpS$ means that agent a has been sent a call for proposals message within conversation

```
INVARIANTS
        cfp ⊆ CONVERSATION
        responded ⊆ cfp
        selected ⊆ responded
        informed ⊆ selected
EVENTS
        callForProposals
           ANY c WHERE
               c ∈ CONVERSATION
               c ∉ cfp
           THEN
               cfp := cfp ∪ {c}
           END
        respond
           ANY c WHERE
               c ∈ cfp
               c ∉ responded
           THEN
               responded := responded ∪ {c}
           END
        select
           ANY c WHERE
               c ∈ responded
               c ∉ selected
           THEN
               selected := selected ∪ {c}
           END
        inform
           ANY c WHERE
               c ∈ selected
               c ∉ informed
           THEN
               informed := informed ∪ {c}
           END
```

Figure 8.3: Abstract Machine of the Initial Chain

c and $c \mapsto a \in cfpR$ means that agent a has received a call for proposals message within conversation c. A message must be sent before it can be received and this is modelled by specifying a subset relationship between the sent variables and the received variables, e.g. $cfpR \subseteq cfpS$. Some of the variables from the abstract machine are replaced by the message variables in the refinement. The last two invariants are the gluing invariant and specify the refinement relationships between the abstract variables that represent the state of the conversation and the concrete variables that model messages being broadcast.

The events of the refinement are shown in Figure 8.5. The `sendCfp` event refines the abstract `callForProposals` event. It models the broadcast of a call for proposals message from agent a to all other agents ($AGENT \setminus \{a\}$) by a set of relationships, as, between a conversation and the agents in the system and adds it to the `cfpS` state. The `receiveCfp` event models a message being received by an agent by selecting a relationship, $c \mapsto a$, that is in the `cfpS` state and adding it to the `cfpR` state. The `sendProposal` event can occur when there is a relationship in the `cfpR` state and the proposal is sent when

$$
\begin{array}{l}
cfpS,\ proposeS,\ acceptS,\ rejectS,\ informS \ \in \ CONVERSATION \ \leftrightarrow \ AGENT \\
cfpR \ \subseteq \ cfpS \\
proposeR \ \subseteq \ proposeS \\
acceptR \ \subseteq \ acceptS \\
rejectR \ \subseteq \ rejectS \\
informR \ \subseteq \ informS \\
proposeS \ \subseteq \ cfpR \\
acceptS \ \subseteq \ proposeR \\
informS \ \subseteq \ acceptR \\
cfp \ = \ dom(cfpS) \\
selected \ = \ dom(acceptS)
\end{array}
$$

FIGURE 8.4: Invariants of the Refinement of the Initial Chain

the relationship is added to the `proposeS` state. The `receiveProposal` event adds a relationship that is in the `proposeS` state to the `proposeR` state. The `responded` event is a refinement of the abstract `respond` event and represents the initiator receiving the required responses. The `select` event broadcasts two different messages. One group of agents, as, will receive an accept message in response to their proposal and another group of agents, ar, will receive a reject message. The `receiveAccept` and `receiveReject` events represent those messages being received by the participants. The `sendInform` event models an agent that has received an accept message sending an inform message following the successful completion of their task. The `receiveInform` event represents this message being received. The final `informed` event refines the abstract `inform` event and models the initiator concluding that the contract has been successfully completed following the receipt of at least one inform message.

The set of fault-tolerance patterns presented in this chapter model solutions for faults that arise in multi-agent systems. This includes faults that are found in ordinary distributed systems. The Timeout pattern prevents an agent from indefinitely waiting for a communication. This allows the agent to cope with faults in either the communication medium, or other nodes or agents in the system. The failure of a node to complete a delegated task is modelled by the Failure pattern. A rational agent altering its goals is modelled by the Cancel pattern. The Refuse pattern allows the system to cope with an agent deciding not to participate in an interaction. The Not-Understood pattern models the reaction of agents to unexpected communications. With the patterns specified in an Event-B development the developer can then refine the models to include more detail on how the system or individual agent will manage these faults.

```
sendCfp  REFINES callForProposals
  ANY  c, as, a   WHERE
    c ∈ CONVERSATION
    c ∉ dom(cfpS)
    as ∈ CONVERSATION ↔ AGENT
    a ∈ AGENT
    dom(as) = {c}
    ran(as) = AGENT \ {a}
  THEN
    cfpS := cfpS ∪ as
  END
sendProposal
  ANY  c, a   WHERE
    c ↦ a ∈ cfpR
    c ↦ a ∉ proposeS
  THEN
    proposeS := proposeS ∪ {c ↦ a}
  END
responded  REFINES respond
  ANY  c   WHERE
    c ∈ dom(proposeS)
    c ∉ responded
  THEN
    responded := responded ∪ {c}
  END

receiveAccept
  ANY  c, a   WHERE
    c ↦ a ∈ acceptS
    c ↦ a ∉ acceptR
  THEN
    acceptR := acceptR ∪ {c ↦ a}
  END
sendInform
  ANY  c, a   WHERE
    c ↦ a ∈ acceptR
    c ↦ a ∉ informS
  THEN
    informS := informS ∪ {c ↦ a}
  END
informed  REFINES inform
  ANY  c   WHERE
    c ∈ dom(informR)
    c ∉ informed
  THEN
    informed := informed ∪ {c}
  END
```

```
receiveCfp
  ANY  c, a   WHERE
    c ↦ a ∈ cfpS
    c ↦ a ∉ cfpR
  THEN
    cfpR := cfpR ∪ {c ↦ a}
  END
receiveProposal
  ANY  c, a   WHERE
    c ↦ a ∈ proposeS
    c ↦ a ∉ proposeR
  THEN
    proposeR := proposeR ∪ {c ↦ a}
  END
select  REFINES select
  ANY  c, as, ar   WHERE
    c ∈ dom(proposeR)
    c ∉ dom(acceptS)
    c ∉ dom(rejectS)
    as ⊆ {c} ◁ proposeR
    as ≠ ∅
    ar = {c} ◁ proposeR \ as
    c ∈ responded
  THEN
    acceptS := acceptS ∪ as
    rejectS := rejectS ∪ ar
  END
receiveReject
  ANY  c, a   WHERE
    c ↦ a ∈ rejectS
    c ↦ a ∉ rejectR
  THEN
    rejectR := rejectR ∪ {c ↦ a}
  END
receiveInform
  ANY  c , a   WHERE
    c ↦ a ∈ informS
    c ↦ a ∉ informR
  THEN
    informR := informR ∪ {c ↦ a}
  END
```

Figure 8.5: Events of the Refinement of the Initial Chain

8.5 Timeout Pattern

Name:Timeout
Problem: An agent may become deadlocked during a conversation whilst waiting for replies. Specifying a deadline will allow the agent to continue the conversation as if it were expecting no more replies.
Solution: Specify a state for the conversation that models a deadline passing. Add an event to the specification that will change the state of the conversation from before the deadline to after the deadline. Split the event for receiving the replies into two. One event will have a guard that is true before the deadline and one will have a guard that is true after the deadline. The event after the deadline will lead to the rejection of any replies.

In the case of a communication failure, or the failure of another agent or node in the system, an agent that continues to wait for a response to a communication may wait an excessively long time or may never receive the reply. This is not practical for most systems, especially a multi-agent system that may be expected to be able to adapt under such circumstances. An agent should be able to decide to either continue the conversation without waiting for a response or to resolve its goal in another way, when it becomes likely that a response will not be forthcoming. An agent may be required to make a decision on how long it should wait depending on its goals for the efficiency of its current task.

The Timeout pattern prevents an agent from becoming deadlocked whilst waiting for a reply. It does this by modelling a deadline after which the behaviour of the system changes. The interaction diagrams in Figure 8.6 shows the messages that are exchanged between the roles involved in the conversation. The Timeout pattern requires that any replies received after the deadline will be automatically rejected. The agent that has the role of initiating the request will be responsible for enforcing the deadline. Diagram A shows a successful one-to-one interaction with the response to a request being received by the initiator before the deadline. In this case the reply from the initiator will depend on the initiator's decision about the response. Diagram B shows the initiator's deadline occurring before the response is received and in this case the reply from the initiator is a rejection of the response. Diagram C shows how a one-to-many interaction can affect the Timeout pattern. Responses are received from different participating agents before and after the deadline has passed. Those received before the deadline will elicit replies that depend on a decision that is made by the initiating agent. Those received after will result in a rejection.

Figure 8.7 shows the `callForProposals`, `respond` and `failure` events that are required for the Timeout pattern to be applied in the abstract machine. The `callForProposals` and `respond` events are already present in the initial chain. The failure event has been added to model the system responding when no proposals are received before the

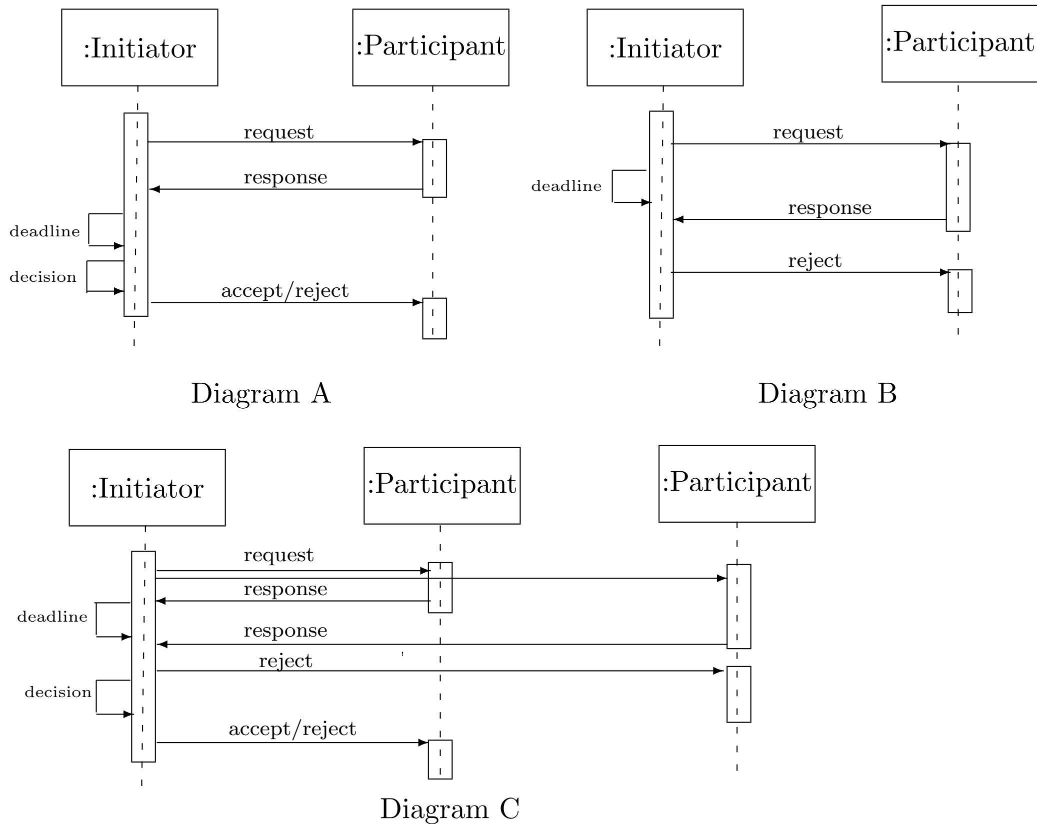

FIGURE 8.6: Interaction Diagram for Timeout Pattern

deadline. The pattern for the timeout could be more general than that taken from the contract net case study. Any request by an agent that waits for a response could use the Timeout pattern to ensure that the requesting agent does not wait indefinitely. The abstract pattern in Figure 8.7 conforms to this general request-response pattern.

```
INVARIANTS
      failed ⊆ cfp
EVENTS
  callForProposals                    respond
      ANY c WHERE                         ANY c WHERE
        c ∈ CONVERSATION                    c ∈ cfp
        c ∉ cfp                             c ∉ responded
      THEN                                THEN
        cfp := cfp ∪ {c}                    responded := responded ∪ {c}
      END                                 END
  failure
      ANY c WHERE
        c ∈ cfp
        c ∉ failed
        c ∉ informed
      THEN
        failed := failed ∪ {c}
      END
```

FIGURE 8.7: Abstract Events for the Timeout Pattern in the Contract Net

The invariant conditions for the refinement are shown in Figure 8.8. To create the

states for before and after the deadline two variables have been added to the model; `beforeTimeout` and `afterTimeout`. The pattern could have been specified with just the `afterTimeout` variable. Both variables were included to make the affect of the deadline clear in the model. The `beforeTimeout` variable is specified as a subset of the domain of the `cfpS` variable so the timeout cannot occur before the conversation has begun. Variables have been added to the model to represent the proposals that are received after the deadline, `proposeRD`, the reject messages sent in response to these proposals, `rejectSD`, and then received, `rejectRD`. The `failedCfp` variable refines the abstract `failed` variable to model the state when the deadline has passed, $failedCfp \subseteq afterTimeout$, and no proposals have been received, $failedCfp \cap dom(proposeR) = \varnothing$.

$$
\begin{array}{l}
beforeTimeout \ \subseteq \ dom(cfpS) \\
afterTimeout \ \subseteq \ beforeTimeout \\
proposeRD \ \subseteq \ proposeS \\
rejectSD \ \subseteq \ proposeRD \\
rejectRD \ \subseteq \ rejectSD \\
failedCfp \ \subseteq \ afterTimeout \\
failedCfp \ \cap \ dom(proposeR) \ = \ \varnothing \\
failed \ = \ failedCfp
\end{array}
$$

Figure 8.8: Invariants for the Refinement of the Timeout Pattern in the Contract Net

Events have been added to the initial chain and existing events have been modified to apply the Timeout pattern. The new and modified events are shown in Figure 8.9 where the names of the new events, and the modifications to existing events, are underlined. The `sendCfp` event has an additional action that adds the conversation to the `beforeTimeout` state. The guard of the `receiveProposal` event has been strengthened so that it can only occur when the conversation is not in the `afterTimeout` state. The new `deadline` event moves the conversation from the state `beforeTimeout` into the state `afterTimeout`. The new `receiveProposal2` event can only occur when the conversation is in the `afterTimeout` state. The action of the event adds the relationship from the `proposeS` state to the new `proposeRD` state. The new `sendReject` event will take a relationship that is in the `proposeRD` state and add it to the `rejectSD` state. This models the initiator responding with a reject message to any proposals received after the timeout. The new `receiveReject2` event will take a relationship that is in the `rejectSD` state and add it to `rejectRD` state. Instead of adding this as a new event a developer could merge it with the existing `receiveReject` event from the initial chain. The new `failToPropose` event refines the abstract `failure` event that was added to the abstract model for the Timeout pattern. It can occur after the deadline has passed and no proposals have been received.

```
sendCfp  REFINES callForProposals
   ANY  c, as, a   WHERE
      c ∈ CONVERSATION
      c ∉ dom(cfpS)
      as ∈ CONVERSATION ↔ AGENT
      a ∈ AGENT
      dom(as) = {c}
      ran(as) = AGENT \ {a}
   THEN
      cfpS := cfpS ∪ as
      beforeTimeout := beforeTimeout ∪ {c}
   END
deadline
   ANY  c   WHERE
      c ∈ beforeTimeout
      c ∉ afterTimeout
   THEN
      afterTimeout := afterTimeout ∪ {c}
   END
sendRejectD
   ANY  c , a   WHERE
      c ↦ a ∈ proposeRD
      c ↦ a ∉ rejectSD
   THEN
      rejectSD := rejectSD ∪ {c ↦ a}
   END

failToPropose  REFINES failure
   ANY  c  WHERE
      c ∉ dom(proposeR)
      c ∈ afterTimeout
      c ∉ failedCfp
   THEN
      failedCfp := failedCfp ∪ {c}
   END
```

```
receiveProposal
  ANY  c, a   WHERE
      c ↦ a ∈ proposeS
      c ↦ a ∉ proposeR
      c ∉ afterTimeout
  THEN
      proposeR := proposeR
          ∪ {c ↦ a}
  END
receiveProposalD
  ANY  c, a   WHERE
      c ↦ a ∈ proposeS
      c ↦ a ∉ proposeR
      c ↦ a ∉ proposeRD
      c ∈ afterTimeout
  THEN
      proposeRD := proposeRD
          ∪ {c ↦ a}
  END
receiveRejectD
  ANY  c, a   WHERE
      c ↦ a ∈ rejectSD
      c ↦ a ∉ rejectRD
  THEN
      rejectRD := rejectRD
          ∪ {c ↦ a}
  END
```

FIGURE 8.9: Concrete Events for the Timeout Pattern in the Contract Net

8.6 Refuse Pattern

Name: Refuse
Problem: An agent cannot support the action requested.
Solution: Add an event for an agent to send a refuse message in response to a request and an event for an agent to receive a refuse message.

Not all agents that receive a request will be able to fulfill it. The request may be in conflict with the agent's own goals. This could be simply due to the agent being overloaded and not wishing to take on any more tasks, or it could be that the agent is competing against the requestor and it would not be in their interest to help. Software design does not always implement the concept of a refusal. Object-based systems use the term 'design by contract' to describe an obligation held by an object that it cannot alter at runtime (Meyer (1997)). The autonomy of agents means that the obligations

between agents are weaker than in design by contract and a multi-agent system must be designed to cope when an agent refuses to undertake a request.

The Refuse pattern differs from the Timeout pattern. In the Timeout pattern the agent that sends the initial message, the initiator, makes the decision in specifying the deadline that may lead to the failure of the conversation. The initiator then copes with the fault. In the Refuse pattern it is the receiver of the initial message that makes the decision to refuse the request and may trigger the failure of the conversation. The initiator must then cope with the fault that has been triggered by the response from the participant.

The Refuse pattern allows an agent to respond to a request that it cannot support, that is not correctly requested or that the requesting agent is not authorised to request. An agent is allowed a choice when responding to a request. The agent can either agree to fulfill the request or it can refuse.

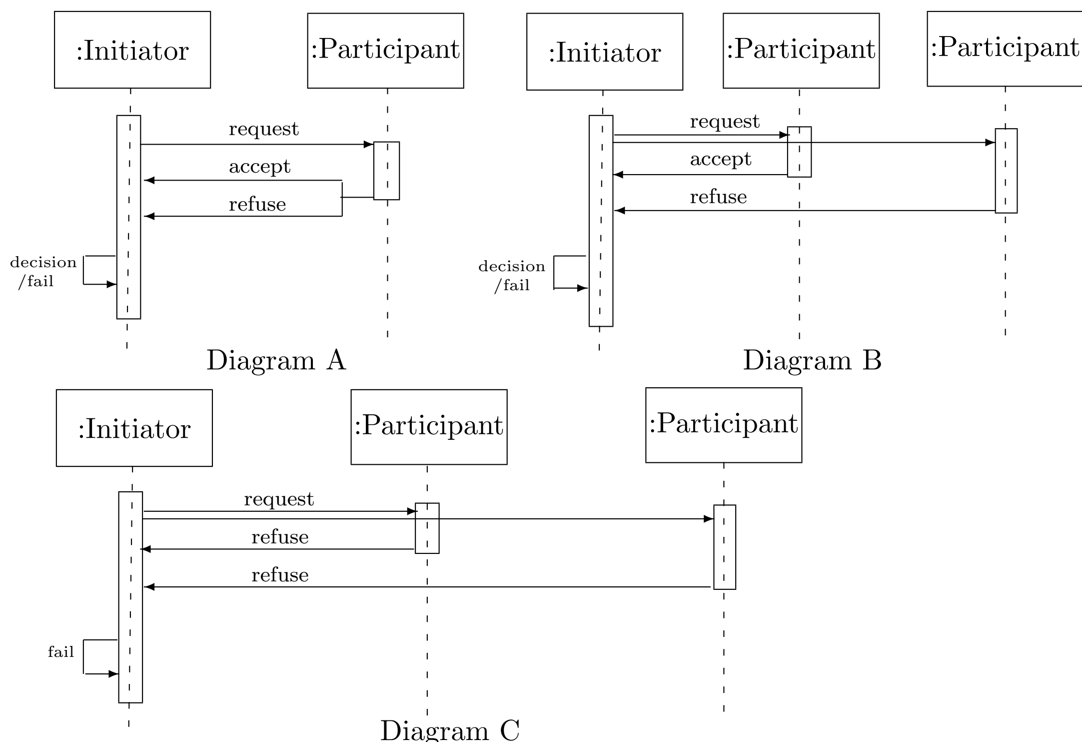

FIGURE 8.10: Interaction Diagram for Refuse Pattern

Figure 8.10 shows interaction diagrams for the Refuse pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a request to a participant agent. The participant agent can respond with either an accept or refuse message. The initiator will then make a decision and the interaction may fail if the accept message is not suitable or a refuse message was sent. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of accept and refuse messages are received in response to the request. Diagram C shows the case where only refuse messages are received and the only outcome is a failure of the interaction.

The events that are required in the abstract machine for the Refuse pattern are the same as those shown in Figure 8.7 for the Timeout pattern. To model the refuse pattern in the refinement of the initial chain three variables and three events have been added. In the contract net case study the refusals are modelled so they are equivalent to the proposals. The invariants in Figure 8.11 specify variables that model sending and receiving refuse messages. An additional invariant specifies that the proposals and refusals for a conversation cannot be from the same agent, $refuseS \cap proposeS = \varnothing$. The `failedCommit` variable models that state of the conversation when all of the replies are refusals. This variable refines the abstract `failed` variable.

$$
\begin{array}{l}
refuseS \subseteq cfpR \\
refuseR \subseteq refuseS \\
refuseS \cap proposeS = \varnothing \\
failedCommit \subseteq dom(refuseR) \\
failedCommit \cap dom(proposeR) = \varnothing \\
failed = failedCommit
\end{array}
$$

FIGURE 8.11: Concrete Invariants for the Refuse Pattern in the Contract Net

The events for the pattern are shown in Figure 8.12. The guard of the `sendProposal` event from the initial chain has been modified to prevent an agent that has made a refusal for the conversation from also making a proposal. The `sendRefusal` event adds a relationship that is in the `cfpR` variable to the `refuseS` variable. The `receiveRefusal` event takes a relationship that is in the `refuseS` variable and adds it to the `refuseR` variable. The `failToCommit` event models the case when all of the responses are refusals and the conversation fails. This is a refinement of the abstract `failure` event.

```
sendProposal
    ANY c, a WHERE
      c ↦ a ∈ cfpR
      c ↦ a ∉ proposeS
      c ↦ a ∉ refuseS
    THEN
      proposeS := proposeS ∪ {c ↦ a}
    END

sendRefusal
    ANY c, a WHERE
      c ↦ a ∈ cfpR
      c ↦ a ∉ proposeS
      c ↦ a ∉ refuseS
    THEN
      refuseS := refuseS
            ∪ {c ↦ a}
    END

receiveRefusal
    ANY c, a WHERE
      c ↦ a ∈ refuseS
      c ↦ a ∉ refuseR
    THEN
      refuseR := refuseR ∪ {c ↦ a}
    END

failToCommit REFINES failure
    ANY c WHERE
      c ∈ dom(refuseR)
      c ∉ dom(proposeR)
      c ∉ failedCommit
    THEN
      failedCommit :=
          failedCommit ∪ {c}
    END
```

FIGURE 8.12: Concrete Events for the Refuse Pattern in the Contract Net

8.7 Cancel Pattern

Name: Cancel
Problem: The requesting agent no longer requires an action to be performed.
Solution: Add an event to the specification for an agent to send a cancel message to an agent that has agreed to perform an action on its behalf. Add an event for that agent to receive a cancel message. Further events need to be added to allow the agent to reply with either an inform, if they have cancelled the action, or a failure, if they have not, and for those messages to be received.

Once an agent has requested an action they can then request that it is cancelled. An agent that exhibits rational behaviour may change its goals because the goal conflicts with other goals, the agent no longer desires that the goal is fulfilled or the agent no longer believes that the goal can be fulfilled (Ferber (1999)). For the initiating agent to ensure that its beliefs about its environment are consistent it needs to know if the agents to whom it has delegated tasks have managed to undo any actions that they have performed. The responses of the agents may affect the actions that the initiating agent takes in response to its change of goals. This potential remedial action by the participating agents makes the Cancel pattern different from the Timeout pattern where the participants do not have to take any action after the deadline has passed.

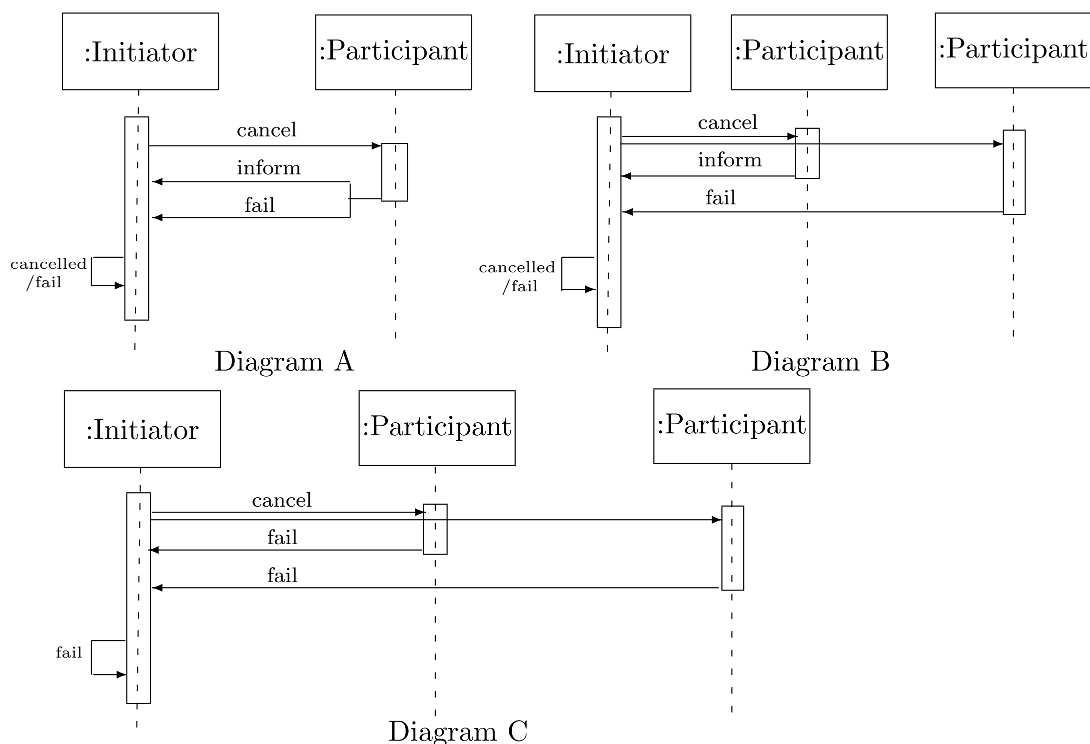

Figure 8.13: Interaction Diagram for Cancel Pattern

The Cancel pattern allows the agent that initiated the conversation to cancel the conversation at any point. The Cancel pattern will cancel a single request in a one-to-one conversation and will broadcast the cancellation in a one-to-many conversation to cancel

all of the requests. Figure 8.13 shows interaction diagrams for the Cancel pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a cancel message to a participant agent. The participant agent can respond with either an inform message if they have successfully cancelled or a fail message if they have not. The initiator will then act according to its knowledge about the state of the system. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of inform and fail messages are received in response to the cancel message. Diagram C shows the case where only fail messages are received and the cancelling of the action fails.

The Cancel pattern requires a new variable and event to be added to the abstract machine of the initial chain. The abstract pattern example in Figure 8.14 shows the `cancel` event moving the conversation into the `cancelled` state.

INVARIANTS

$cancelled \subseteq cfp$

EVENTS

cancel

ANY c WHERE

$c \in cfp$

$c \notin cancelled$

THEN

$cancelled := cancelled \cup \{c\}$

END

FIGURE 8.14: Abstract Events for the Cancel Pattern in the Contract Net

The Cancel Pattern is modelled in the refinement as a collection of events that can occur at any point in the conversation. Events model a cancel message being sent from the initiating agent and received by the other agents involved. Events are also required to model the participating agents responding to the cancel request to inform the initiating agent whether they have managed to cancel their actions.

$cancelS \subseteq cfpS$
$cancelR \subseteq cancelS$
$informCancelS \subseteq cancelR$
$informCancelR \subseteq informCancelS$
$failCancelS \subseteq cancelR$
$failCancelR \subseteq failCancelS$
$informCancelled \subseteq dom(informCancelR)$
$failCancelled \subseteq dom(failCancelR)$
$informCancelS \cap failCancelS = \varnothing$
$informCancelled \cap failCancelled = \varnothing$
$cancelled = informCancelled \cup failCancelled$

FIGURE 8.15: Invariants for the Refinement of the Cancel Pattern in the Contract Net

Figure 8.15 shows the invariant conditions from the Event-B example of the Cancel pattern. The variables represent the states of the system as messages are sent and received. The `cancelS` variable is a subset of the `cfpS` variable so a conversation cannot be cancelled before it has begun. All of the other variables are specified as subsets according to the order of the messages that they represent being sent and received. `InformCancelS` and `failCancelS` are specified so the same agent cannot send an inform

and fail message in the same conversation, $informCancelS \cap failCancelS = \varnothing$. The `informCancelled` and `failCancelled` variables are specified so the conversation cannot be in both states, $informCancelled \cap failCancelled = \varnothing$. The final invariant condition is the gluing invariant that relates the abstract `cancel` variable to a conjunction of the `informCancelled` and `failCancelled` variables.

Figure 8.16 shows the events that have been added to the initial refinement model to specify the Cancel pattern. The `sendCancel` event can be triggered by the initiating agent at any point in the conversation. The cancel message is broadcast to every agent involved in the conversation, $as = \{c\} \lhd cfpS$. The `receiveCancel` event allows the participants to receive the cancel message. The `sendInformCancel` and `sendFailCancel` events model the participants sending a message to the initiator about the success or failure of the cancellation. The `receiveInformCancel` and `receiveFailCancel` events model the initiator receiving the message. The last two events, `informCancelled` and `failCancelled`, refine the abstract `cancel` event and model the initiator evaluating of the success of the cancellation. The guards for the two events specify that at least one inform or fail cancel message has been received. The developer may want to strengthen these guards. For example, the guard of the `informCancelled` event could be strengthened to specify that all of the agents have replied with an inform message, $\{c\} \lhd informCancelR = AGENT \setminus \{a\}$, or that no fail messages have been received, $c \notin dom(failCancelR)$.

8.8 Failure Pattern

Name: Failure
Problem: An agent is prevented from carrying out an agreed action.
Solution: Add an event for an agent to send a failure message after they have committed to performing an action on behalf of another agent. Add an event for an agent to receive a failure message after a commitment has been made and an event for the system to respond to the failure.

An agent that makes a commitment to perform an action may be prevented from carrying it out. The agent that requested the action should be informed of this failure so that its beliefs do not become inconsistent.

Figure 8.17 shows interaction diagrams for the Failure pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a message that requests an action to a participant agent. The participant agent can respond with either an inform message, if they have successfully carried out the action, or a fail message, if they have not. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of inform and fail messages are received in response to the accept message. The initiator will then be able to evaluate whether the task was carried out successfully.

```
sendCancel
  ANY c , as WHERE
    c ∈ dom(cfpS)
    as = {c} ◁ cfpS
    c ∉ dom(cancelS)
  THEN
    cancelS := cancelS ∪ as
  END
```

```
receiveCancel
  ANY c, a WHERE
    c ↦ a ∈ cancelS
    c ↦ a ∉ cancelR
  THEN
    cancelR := cancelR ∪
               {c ↦ a}
  END
```

```
sendInformCancel
  ANY c, a WHERE
    c ↦ a ∈ cancelR
    c ↦ a ∉ informCancelS
    c ↦ a ∉ failCancelS
  THEN
    informCancelS := informCancelS
        ∪ {c ↦ a}
  END
```

```
sendFailCancel
  ANY c, a WHERE
    c ↦ a ∈ cancelR
    c ↦ a ∉ failCancelS
    c ↦ a ∉ informCancelS
  THEN
    failCancelS := failCancelS
        ∪ {c ↦ a}
  END
```

```
receiveInformCancel
  ANY c, a WHERE
    c ↦ a ∈ informCancelS
    c ↦ a ∉ informCancelR
  THEN
    informCancelR := informCancelR
        ∪ {c ↦ a}
  END
```

```
receiveFailCancel
  ANY c, a WHERE
    c ↦ a ∈ failCancelS
    c ↦ a ∉ failCancelR
  THEN
    failCancelR := failCancelR
        ∪ {c ↦ a}
  END
```

```
informCancelled REFINES cancel
  ANY c WHERE
    c ∈ dom(informCancelR)
    c ∉ informCancelled
    c ∉ failCancelled
  THEN
    informCancelled := informCancelled
                       ∪ {c}
  END
```

```
failCancelled REFINES cancel
  ANY c WHERE
    c ∈ dom(failCancelR)
    c ∉ failCancelled
    c ∉ informCancelled
  THEN
    failCancelled := failCancelled
                     ∪ {c}
  END
```

Figure 8.16: Concrete Events for the Cancel Pattern in the Contract Net

Diagram C shows the case where only fail messages are received. The Failure pattern is similar to the Refuse pattern where the responding agent has a choice of two replies that affect the outcome of the interaction differently. It occurs at a different point in the conversation. The Refuse pattern is used before a commitment is made and the Failure pattern is required after a commitment has been made. The Failure pattern differs from the Timeout pattern in the same way as the Refuse pattern. In the Failure pattern the initiator may have to take action in response to a fault triggered by the participant, whereas in the Timeout pattern the fault is triggered by the action of the initiator setting the deadline.

Figure 8.18 shows the events from the abstract machine that are related to the Failure pattern. The Failure pattern specifies the failure of the conversation after the selection of the proposals has been made. Either the `inform` event or the `failure` event can complete the conversation.

Figure 8.19 shows the invariants added for the Failure pattern. The Event-B models the agents involved in the contract net interaction protocol sending failure messages instead of inform messages after they have had their proposal accepted. The conversation

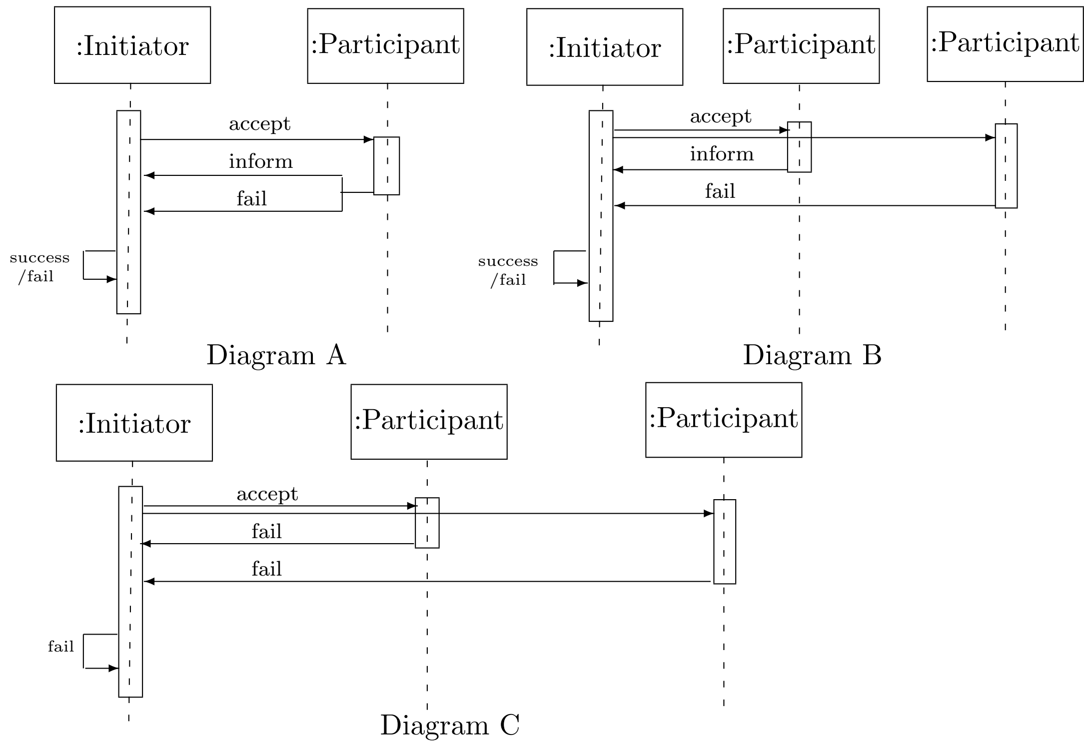

FIGURE 8.17: Interaction Diagram for Failure Pattern

```
select
    ANY  c  WHERE
      c ∈ responded
      c ∉ selected
      c ∉ failed
    THEN
      selected := selected ∪ {c}
    END
```

```
inform
    ANY  c  WHERE
      c ∈ selected
      c ∉ informed
      c ∉ failed
    THEN
      informed := informed ∪ {c}
    END
```

```
failure
    ANY  c  WHERE
      c ∈ cfp
      c ∉ failed
      c ∉ informed
    THEN
      failed := failed ∪ {c}
    END
```

FIGURE 8.18: Abstract Events for the Failure Pattern in the Contract Net

cannot succeed and fail and this is modelled by an invariant condition that specifies the intersection of the informed and failed variables as empty, $informed \cap failed1 = \varnothing$.

$$
\begin{aligned}
&failS \subseteq acceptR \\
&failR \subseteq failS \\
&failed1 \subseteq dom(failR) \\
&informed \cap failed1 = \varnothing \\
&failed = failed1
\end{aligned}
$$

FIGURE 8.19: Invariants for the Refinement of the Failure Pattern in the Contract Net

Figure 8.20 shows the three events that are added to the initial concrete model. The

`sendFail` event models a participant having received an accept message that instructs it to carry out a task, $c \mapsto a \in acceptR$, sending a failure message in response. The `receiveFail` event models the initiator receiving the failure message. The `failed` event refines the abstract `failure` event and can occur after a failure message has been received.

```
sendFail
    ANY c, a WHERE
        c ↦ a ∈ acceptR
        c ↦ a ∉ failS
        c ↦ a ∉ informS
    THEN
        failS := failS ∪ {c ↦ a}
    END
failed REFINES failure
    ANY c WHERE
        c ∈ dom(failR)
        c ∉ failed1
        c ∉ informed
    THEN
        failed1 := failed1 ∪ {c}
    END

receiveFail
    ANY c, a WHERE
        c ↦ a ∈ failS
        c ↦ a ∉ failR
    THEN
        failR := failR ∪ {c ↦ a}
    END
```

FIGURE 8.20: Concrete Events for the Failure Pattern in the Contract Net

8.9 Not-Understood Pattern

Name:Not-Understood
Problem: An agent receives a message that it does not expect or does not recognise.
Solution: Specify an event for receiving a message with an unknown or unexpected performative. Specify the action as replying with a not-understood message. Specify an event for receiving a not-understood message.

The autonomy of the agents means that there is no guarantee of their behaviour and the non-hierarchical nature of multi-agent systems often means that there is no single point of control. For agents in a multi-agent system to maintain a correct understanding of their environment they need to communicate with the other agents in the system to be aware of the actions of the other agents. This can create a large number of messages being passed between agents for them to be able to negotiate, query and inform. The possible heterogeneity of the agents means that they may have a different understanding of interaction protocols. The possibility of receiving arbitrary messages increases with each of these factors and the system needs to be able to cope with such faults.

The concept of the not-understood message is described in FIPA (2002c). The not-understood message communicates that the sending agent has received a message that it does not understand. A not-understood message can be sent or received at any point in the conversation.

It is suggested in FIPA (2002c) that the action taken in response to a not-understood message should be different when the conversation involves broadcast messages and sub-protocols than that taken as part of a one-to-one conversation. It may be inappropriate to cancel the conversation when there are multiple agents performing sub-protocols. Each response to a not-understood message should be evaluated depending on the status of the conversation and is not specified by the Not-Understood pattern.

The Not-Understood pattern involves agents receiving an arbitrary message, responding with a not-understood message and agents receiving a not-understood message. The action taken by the agent to cope with the potential fault is not modelled and is left for the developer to treat.

Figure 8.21 shows an interaction diagram for the Not-Understood pattern. The interaction diagram shows an interaction between any two agent roles. One agent sends another agent a message that the receiving agent does not understand. The response from the receiving agent will be to reply with a not-understood message. The action taken by the agent that receives the not-understood message depends on their role in the conversation and the stage of the conversation.

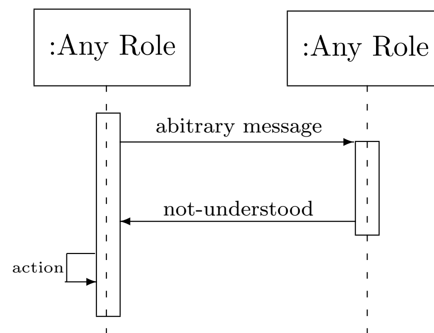

FIGURE 8.21: Interaction Diagram for Not-Understood Pattern

To model the Not-Understood pattern two events have been added to the initial abstract machine for the contract net case study. The events and variables are shown in Figure 8.22. The `arbitraryComm` event models the unrecognised message being received. The `receiveNotUnderstood` event abstractly model an agent receiving a not-understood message.

Figure 8.23 shows the extract from the Event-B refinement model that models the Not-Understood pattern in the Contract Net case study. The `unknownR` variable represents an arbitrary message being received and the `notUnderstoodS` variable represents a not-understood message being sent in response to the receipt of an arbitrary message. The `notUnderstoodR` variable represents a not-understood message being received.

The `receiveArbitraryComm` event models the receipt of a message that is not understood by the receiving agent. The `sendNotUnderstood` event models a not-understood

```
INVARIANTS
  recUnknown ⊆ CONVERSATION
  recNotUnderstood ⊆ cfp
EVENTS
  arbitraryComm
     ANY c  WHERE
       c ∈ cfp
     THEN
       recUnknown := recUnknown ∪ {c}
     END
  receiveNotUnderstood
     ANY c  WHERE
       c ∈ cfp
     THEN
       recNotUnderstood :=
          recNotUnderstood ∪ {c}
     END
```

Figure 8.22: Abstract Events for the Not-Understood Pattern in the Contract Net

message being sent in response to the receipt of this message. The `receiveNotUnderstood` event models an agent receiving a not-understood message. Further refinements of the example will model the agent's reactions to receiving the not-understood message. An initiator agent may decide to cancel the conversation or they may decide that the conversation has failed. The decisions by the agents will depend on the stage of the conversation when the not-understood message is received. This is left for the developer to decide and model.

```
INVARIANTS
        unknownR ⊆ cfpS
        notUnderstoodS ⊆ unknownR
        recUnknown = dom(notUnderstoodS)
        notUnderstoodR ⊆ notUnderstoodS
        recNotUnderstood = dom(notUnderstoodR)

EVENTS
receiveArbitraryComm
     ANY c, a  WHERE
       c ↦ a ∈ cfpS
     THEN
       unknownR := unknownR ∪ {c ↦ a}
     END
sendNotUnderstood  REFINES arbitraryComm
     ANY c, a  WHERE
       c ↦ a ∈ unknownR
       c ↦ a ∉ notUnderstoodS
     THEN
       notUnderstoodS := notUnderstoodS ∪ {c ↦ a}
     END
receiveNotUnderstood  REFINES receiveNotUnderstood
     ANY c, a  WHERE
       c ↦ a ∈ notUnderstoodS
       c ↦ a ∉ notUnderstoodR
     THEN
       notUnderstoodR := notUnderstoodR ∪ {c ↦ a}
     END
```

Figure 8.23: Concrete Invariants and Events for the Refinement of the Not-Understood Pattern in the Contract Net

8.10 Related Work

This section describes work that is related to the ideas presented in this chapter. This work outlines approaches for constructing patterns in the B-Method and Event-B. Other work of interest are design patterns for multi-agent systems particularly patterns that can be integrated with goal models that are used in multi-agent system design. Other fault-tolerance techniques for multi-agent systems have been investigated and are summarised in this section.

The B-Method is used in Blazy et al. (2003) to specify patterns, such as those identified in Gamma et al. (1995), as abstract machines. The pattern machines are instantiated by including another B model in the machine using the B-Method's inclusion mechanism. Pattern models can be composed to create a new pattern by using the inclusion mechanism to construct a new machine from the separate patterns. These patterns are specified at a single level of abstraction and are based on object-oriented development methods.

A set of patterns that solve design problems that are common when using the B-Method has been produced in Chan et al. (2007). The patterns they present include a pattern to associate multiple B machines, a pattern to produce unique objects and patterns for creating sub and super-types of B machines. The patterns are implemented as either extracts of B machines or a description of how different mechanisms from the B-Method can be used to solve a described problem alongside an example of the patterns use. As with those described above, these patterns attempt to introduce some object-oriented concepts into B machines, are at a single level of abstraction and mainly address structural relationships between machines.

A refinement pattern for modelling time constraints in Event-B is presented in Cansell and Méry (2007). They produce the pattern by constructing a generic Event-B model that specifies the time constraints as a superposition refinement. This model can be re-used to produce new refinements of the model to which the pattern is being applied. The authors suggest that it would be possible to prove the pattern model and the proof obligations generated by the pattern would not need to be discharged for the development model.

There are several methods for the use of patterns in the development of multi-agent systems. They are described and used with informal models. Coordination patterns, including a pattern of the contract net protocol, are presented in Deugo et al. (2001), patterns for mobile agent design are presented in Aridor and Lange (1998), and Schelfthout et al. (2002) present patterns for implementing agents in object-oriented architectures. A strategy for constructing and using design patterns for agent systems that uses goals can be found in Weiss (2003). The patterns can be combined using a pattern language to construct a multi-agent system design.

The patterns presented in this chapter provide fault-tolerance for the agents so they can continue to provide a service. Further strategies for managing faults in agent conversations include adapting general fault-tolerance techniques, such as replication Fedoruk and Deters (2002), redundancy Kumar and Cohen (2000) and checkpoints Wang et al. (2004), to multi-agent systems. Creating patterns for the specification of these fault-tolerance strategies in multi-agent systems is a possible direction for future work.

The patterns presented in this chapter provide an abstraction of the pattern so the affects of applying the pattern can be integrated into the Event-B refinement chain. The related work described above use patterns either as a superposition refinement to an Event-B model or as a component to a model. Integrating a pattern into the refinement chain of a development offers the advantages of making the pattern a fundamental part of the development. It is present in the abstraction of the model and can be analysed at all levels of abstraction.

8.11 Summary

Event-B is a method that is suited to the specification of multi-agent systems as it has been developed for modelling reactive and distributed systems. The patterns presented above allow the developer to specify fault-tolerant behaviour in an Event-B model of a multi-agent system.

The patterns are presented as three elements: a description, interaction diagrams and Event-B examples. The Event-B examples make the patterns specific to Event-B development. The other elements of the pattern are more generic and could be suitable for other event-based formal methods. The inclusion of an abstraction of the pattern creates a pattern that can be fully integrated into the refinement chain of a development. The Event-B extracts included from the Contract Net case study show how the patterns can be applied to the model of a complex multi-agent system. They also provide a reusable specification of the pattern at a single level of refinement. The full specification of the simplified Contract Net case study, with all of the patterns applied, is provided in Appendix C.

Chapter 9

Conclusion

The increase in complexity of distributed software systems requires that a method for modelling such systems can model abstractions, such as agents, that are intended to manage this complexity. Event-B is a method for modelling reactive and distributed systems and should be capable of modelling agent-related concepts. The method presented in this thesis does not extend Event-B, but instead uses informal models and the available modelling elements to capture the concepts required to construct models of multi-agent systems.

The agent concepts used in the Incremental Development Process were taken from a review of AOSE methodologies. The review found that the most common concepts, besides agents, were goals, interaction and roles. The goals of the multi-agent system are expressed as the events of an Event-B model, the interaction of the agents is specified by the transitions between the events of the model and the roles of the agents are modelled by the encapsulation and decomposition of state and events into components.

The Process aims to make the development of multi-agent systems in Event-B more accessible by using informal models as a guide to the development of the formal models. The goal diagram is one that is used in AOSE and should be a familiar tool to developers of multi-agent systems. The guidance provided should make the transformation from the goal diagrams to the formal models straightforward. The guidance is intended to make the discharging of proof obligations easier and to exploit the automatic provers in the RODIN toolkit.

The case studies presented in Chapter 6 and 7 are examples of the types of interactions that will occur in a multi-agent system. The two case studies were chosen because they offer two different levels of complexity in their interactions. The Query-If case study is an interaction between two agents that requires only a few messages to reach a conclusion. The Contract Net case study is an interaction that can involve many agents and several stages. The Query-If case study provides an example that illustrates the application of

the Process. The Contract Net case study shows that the Process can be successfully applied to more complex interactions. The case study models were verified using the RODIN platform and the B4Free project manager. An example of a proof obligation being dicharged interactively using the RODIN platform can be found in Appendix B.

Decomposing the case study models results in the abstract models of the components that are capable of fulfilling the two roles of the interaction. These components can then be refined to include a specification of the internal design of the agent. The decomposition of the system into components increases the possibility of re-using parts of a development. The resource components are examples of components that can be re-used between developments.

Event-B modelling elements are already defined. When translating between the goal model and Event-B models events and state variables can be used to model goals. An Event-B model is a model of a state transition system and can be used to model the interaction. Variables can model resources and the decomposition of the system model into components can model the agent roles. It was unnecessary to add new modelling elements to Event-B to model these concepts. The translation can be carried out by defining these relationships between the Event-B modelling elements and agent concepts and providing guidance in the form of heuristics.

The style of specification used makes specifying an order to the events straightforward. An event cannot move the model into a state that is defined as a subset of another state unless the model is already in the parent state. Using subsets, in this way, also creates a model of the order of the interaction in the specification of the variables. The order of the interaction can be seen by analysing the invariant definitions of the state variables.

Refinement in Event-B relates well to goal elaboration because new events can be added to an Event-B refinement and single events can be refined into multiple events. The XOR relationship is a useful model for goal refinement in Event-B as it describes how a single event can be refined into multiple events. The OR relationship where the behaviour of the refinement is less restrictive than the abstract is more difficult to model in Event-B.

A restriction on the Event-B method was caused by the use of endpoint goals. A refinement model that can be prevented from completing transitions that were possible in the abstract model cannot be verified to satisfy liveness properties. The endpoint goals are an introduction of a safety property. They ensure that the model can cope with the interaction being ending prematurely. The liveness properties of the models have not been covered in this work.

9.1 Limitations

A limitation of the Process is that translating the goal diagrams into the Event-B models requires a systematic approach from the developer especially when the system is large with complex relationships between the goals.

A further limitation is that the Process does not model the full internal specification of the agent components. The Process finishes with a specification of the roles required to fulfill the interaction of the system, but not a specification of how the agent will undertake the required tasks, including the reasoning required.

9.2 Contributions

The main contribution of this thesis is the Incremental Development Process. The Process uses goal diagrams to capture the requirements of the system and to capture the modelling decisions for the system through refinement of the goals. The relationships structure the interactions of the agents. The use of the goal diagrams models the system based on the concepts of goals and interactions that are fundamental to models of agent-based systems. The guidelines for the translation of the informal models into Event-B models allow the modelling decisions and the agent concepts to be specified in the Event-B models using elements that are already part of the Event-B method. The decomposition of the models into the roles and resource components of the system capture another concept that is fundamental to multi-agent systems, that of agent roles. The component specifications can then be further developed and have potential to be re-used. The use of informal and formal models in the Process makes the use of a formal method more accessible by separating the modelling decisions from the formal models.

Further contributions include:

- The identification of the concepts that are fundamental to the modelling of multi-agent systems. Goals represent the motivations of the individual agents that can be shared by the system for the agents to cooperate as a system. Interaction protocols are how the agents cooperate to fulfill their goals. Roles are the basis on which each agent's functioning as part of the system is defined.
- The evaluation of the methods for managing the complexity of developing multi-agent systems showed how abstraction and refinement, and decomposition are methods that are appropriate for this task.
- The Process includes a method for constructing an Event-B refinement chain from models of the goals of a multi-agent system. This allows the modelling decisions for the system to be analysed informally before they are translated into the formal models. Separating the modelling decisions as an informal task should make the formal modelling of multi-agent systems in Event-B more accessible to developers, particularly those unfamiliar with formal methods. Goal models are a technique that can be used to informally model the requirements of a system. Using goals models as the basis for the formal models creates a method that integrates agent motivations at the most abstract level.
- The use of the events and variables in Event-B to model the agents fulfilling their goals. This use shows how state transitions can be used to represent motivations guiding an interaction.
- Using decomposition to model agent roles. The role, or roles, of an agent can be determined by, and used to control, the agents function within the system. Decomposing the system by the events that perform the function to fulfill each identified role provides a specification of that role.
- The case studies used in this thesis are a useful method for explaining the Process and showing its application to multi-agent systems. As models of interactions that are used in multi-agent systems they also have the potential for re-use.
- Chapter 8 describes a set of patterns that capture the modelling decisions required to provide different aspects of fault-tolerance into an Event-B model of a multi-agent system.
- The work on the fault-tolerance patterns also provides an appraisal of how the interactions of multi-agent systems can be used to provide fault-tolerance for the agents in the system.
- The development of the patterns required an evaluation of how patterns can be applied in Event-B. The method of integrating the patterns into the refinement

tree of an Event-B development should make the application of the patterns easier. This integration also makes the patterns a fundamental part of the system that is being developed.

9.3 Comparison of the Process to the Gaia and Tropos Methodologies

This section provides a comparison of the Process to two of the AOSE methodologies evaluated in Chapter 3. This comparison is intended provide more detail on the similarities and differences between the Process and these methodologies. These similarities and differences show how the Process has been inspired by the methodologies. They also show that there is room in the field of AOSE for using Event-B to model multi-agent systems. The Gaia and Tropos methodologies were chosen because of their greater influence on the development of the Process.

The analysis phase of Gaia models multi-agent systems as roles and their interactions. The design phase models more concrete entities such as agents. This comparison of the Process and Gaia will, therefore, cover the analysis phase of Gaia.

Gaia begins by identifying the roles of the system. The Process begins by identifying the goals and then identifying the roles that will be allocated the goals. This makes Gaia more focused on roles as the organisational structure of the system, whereas the Process models the roles as performing the actions of the system and the organisational structure is a by-product of the interactions of the roles. The use of goals as the primary entity allows the integration of Goal Oriented Requirements Engineering techniques. The Gaia methodology does not include requirements capture.

Similarly to the Process, Gaia starts from the system view and takes these abstract models and makes them more concrete. The system is constructed from the roles and their interactions. The Process includes goals, in addition to roles and interactions, in the system view.

Gaia constructs several models of different views of the system e.g. role model and interaction model. The Process uses one model of the system at different levels of abstraction and formality. Gaia also has more detailed recording of the attributes of each of the modelling elements e.g. roles include descriptions of their protocols and permissions etc., whereas in the Process the roles are defined through the allocation of goals and the decomposition of the variables and events for the role. There is a need to balance the complexity of the models with the detail required to make the models complete. This is particularly important with formal models because of the effort required to verify them. Using several different formal models would require formal rules to relate the different views to one another so that each of the models can

be proven consistent. The models used in Gaia are informal and do not need to be formally verifiable.

In Gaia the dependencies between roles are modelled by their interactions. The Process models dependencies between goals and then allocates goals to roles. The interaction required to fulfill the goals is modelled by the dependencies (elaboration relationships) between the goals. This is because the Process primarily takes a goal based view of the system.

The main inspiration taken from the Gaia methodology when developing the Process is the use of roles as an abstraction for agents. The Gaia methodology particularly highlights the need for such abstractions when modelling multi-agent systems.

The Process can be compared to the Tropos methodology as far as the architectural design phase of Tropos. The further phases in the methodology refine the models to include details of the implementation platform and this is beyond the scope of the Process.

Tropos begins with modelling the requirements of the system using goal models. The requirements phase begins with the system and the environment that influences the system. Unlike the Process, Tropos distinguishes between soft and hard goals. The use of the models to identify and analyse requirements makes this distinction useful because of the high-level of abstraction required. Plans describe how the agent will act and can be executed to fulfill the agent's goals. These are modelled alongside goals, using the same method, to provide a further view of the analysis. As with Gaia this can produce models of multiple views of the system.

Tropos uses the term actor to model both agents and roles and does not distinguish the two concepts at the same level of abstraction. Actors are identified with the goals in the goal models and refined as the system requirements are analysed. The Process introduces agent roles after the goals have been elaborated.

Tropos goal models use dependency relationships between the actors and goals, and other actors as a way to structure the organisation of the system. The positive or negative contribution of goals to one another are modelled. The THEN and XOR relationships in the Process goal models can be seen as showing a positive contribution between goals, with a THEN relationship, or a negative contribution, with an XOR relationship. This difference is because the relationships in the goal diagrams for the Process are based on the state transition system that will be modelled in the Event-B models.

The architectural design phase in Tropos builds the system organisation from the identified actors using their goals and dependencies to structure the actor relationships. The actors can be decomposed and the system analysed from different levels of abstraction. The actors can then be decomposed into sub-actors along with the decomposition of

allocated goals. This is not a feature of the goal models of the Process, but re-applying the Process to a component role should result in a similar effect.

The capabilities identified and assigned for the actor dependencies in the architectural design phase of Tropos could be compared to the events that are decomposed for each role during the decomposition of the system in the Process. The events of each of the roles specify the actions that each of the role components is capable of at the modelled level of abstraction.

Tropos uses formal models alongside the informal models of the system requirements. These formal models can be model checked to ensure that the system will meet its requirements and is not inconsistent. The formal analysis of the system is intended to support the development and refinement of the informal models. The Process uses the informal models to support the development of the formal models. The informal models are used to make modelling decisions that can then be translated into the Event-B formal models. These models can then be formally verified.

The Tropos methodology inspired an investigation into Goal Oriented Requirements Engineering and, in turn, the use of goal models in the Process. It also demonstrates how formal models can be constructed from informal representations of agent concepts.

9.4 Comparison of Goal Diagrams

The review of Goal Oriented Requirements Engineering in Chapter 3 Section 3.4.2 provides an overview of the methods described in this section. The intention of this section is to compare the reviewed methods with the techniques used in the Process.

The KAOS method uses AND and OR relationships between goals. The AND relationship requires all of the sub-goals to be satisfied for the parent goal to be satisfied. The OR relationship allows alternative sets of sub-goals to be satisfied in order for the parent goal to be satisfied. The preliminary goal graph has soft goals that have links that are modelled to contribute to, or conflict with, the linked goals. KAOS uses a separate model to create responsibility links between agents and goals and to commit the agent to satisfying a goal within the restrictions placed by conditions such as preconditions (Lamsweerde (2001)). KAOS uses a meta-model to define the language used to create the formal models (Letier (2001)). The meta-model includes concepts used in the different diagrams such as goal, agent and the links between them. Concepts such as pre- and post-conditions are also defined in the meta model. These concepts are then assigned temporal logic predicates and the model can be formally verified.

i* uses dependency links to model relationships between agents, other agents and goals. In the goal model, called the strategic rationale model, there are means-end links that show the alternative ways for achieving a goal. This is similar to an OR relationship.

There are also decomposition links that decompose a task into sub-tasks. This is similar to an AND relationship. Tropos uses heuristics to translate between i* diagrams and the two Formal Tropos models. The outer layer model specifies the agents and other entities and the dependencies between them and their goals. The inner layer model specifies the constraints on the entities as predicates that model properties such as invariant and fulfillment.

REF uses dependency links and AND and OR refinement links in goal models (Donzelli (2004)). The AND refinement requires all goals or tasks to be satisfied and the OR refinement requires that at least one of the associated goals or tasks is satisfied. Currently there is no work linking REF models to formal methods.

The goals in the goal diagram used in the Process are not distinguished as either hard or soft goals as they can be in other frameworks (Yu (1997); Lamsweerde (2001); Bresciani and Donzelli (2003)). This is because the abstract goals can be viewed as soft goals that are eventually elaborated into more concrete or hard goals. Forcing an identification of the point where soft goals turn into hard goals would complicate the construction of the goal diagram. The goals that can be allocated to the agent roles must be hard goals, but the distinction is not required for the Process. Tasks are not distinguished separately from the goals in the goal model because the Event-B models that are constructed are at a relatively high level of abstraction and tasks should be introduced later through refinement of the component models.

To be able to use the goal diagrams as a guide for the construction of the Event-B models a relationship between the formal and informal models had to be found. There has to be enough information in the goal diagram to describe the Event-B model. This affects the way the goal relationships are modelled.

The goal elaboration relationships used in the methods described above are AND and OR relationships. Including the concept of interaction in the goal diagrams means that the order in which the goals are executed is important. The relationship that defines an order is the THEN relationship. Event-B is a model of a state transition system and can be used to model ordered interactions. The THEN relationship is a non-commutative AND relationship and is more useful for the Process than the AND relationship.

A further detail given to some goal relationships are links that express contributions or conflicts between goals on the satisfaction of the parent goal. The endpoint goals in the goal diagrams used in the Process provide a similar detail based on the goals not the relationships.

9.5 Comparison of Informal to Formal Model Translation

There are several examples of work that relate informal goal models to formal models as well as examples of work that translate from an informal model to Event-B models.

There are two methods available to translate i* goal models into formal models as part of the development process. The i* method of goal modelling forms the basis of the first stage of the Tropos development method (Fuxman et al. (2004)). Translation guidelines are provided that are intended to extract the information that is implicit in the i* goal model and create a Formal Tropos model. The Formal Tropos model has an outer layer that describes the actors, goals, tasks and resources and their relationships. The inner layer uses temporal logic to provide invariant, creation and fulfillment conditions for each of the elements. A tool is provided to translate the Formal Tropos model into a language that can be animated and verified with model checking. The Formal Tropos specification can then be refined. Krishna et al. (2004) have produced a methodology for co-evolving i* goal models and Z schemas. The elements of an i* diagram, such as goal and agent, are specified in a Z schema. Guidelines are provided to help the developer then add the elements of their specific goal diagram to the schema model. The Z model is then refined to model the system in greater detail. Guidelines are also provided to add any changes made to the i* diagram into the abstract Z model and then propagate these changes into the refinements. This makes it possible for the i* model and Z schema models to be kept synchronised through iterations of the development process.

The KAOS method for goal-oriented requirements engineering (Lamsweerde (2001)) employs four sub-models. These model the goals, objects, agents and the operations that are derived from the goals. The goal model specifies the hard and soft goals identified for the system and they are related using AND and OR refinement relationships. The relationships are also given different levels to signify how much each goal can contribute or conflict with the related goal. The goals are characterised as either achieve, maintain, cease or avoid goals. The domain elements identified in the models can then be formalised by defining their behaviour using temporal logic. Agents are assigned the responsibility of the goals using AND and OR relationships and from this assignment the variables that can be assigned to the agents can be identified. The operations of the system can then be specified that will satisfy the goals of the system.

Poppleton (2007) describes a method for constructing a specification in Event-B based on informal requirements that are described as features of the system. Abstract machines of each of the features modelled in a feature model are constructed, which are then composed using rules that are intended to be implemented as tool support in future work. The features can then be refined separately allowing re-use. It is suggested that the presented rules for composing the features into a system model can be automated.

UML-B (Snook and Butler (2006)) is one example of work to translate between the UML and B. Annotated UML class and state models are developed and used to generate a B model in a subset of the B syntax. The B model can provide semantics for the UML model. Refinement of the B model is provided through specially adapted UML class and packages entities. UML-B is intended to help developers use formal methods by providing an accessible modelling notation that can be formally verified. UML-B has tools for modelling and translating between the UML and B models. There is currently ongoing work to create a plug-in for the RODIN tool-set that will translate UML2.0 models into Event-B (Butler (2006)).

The refinement diagrams described in Back (2005) allow component and refinement models to be structured diagrammatically using dependency, refinement and extension relationships. The diagrams are based on Lattice Theory and this allows them to be reasoned about and proofs developed using additional diagrams. The resulting diagrams model the architecture of a system and show how each development of the system has refined the original architecture.

Compared to the related work that translates between informal goal models and formal models the goal models used in the Process are simplified to make the Process both more accessible and to make the translation to the Event-B models relatively systematic. The verification of the models in the Process is not focused on the requirements as it is in Tropos and KAOS.

The work presented in Poppleton (2007) does not translate between the feature model and the abstract machines of the separate features. It does provide a set of rules that describe a method to compose the separate models of features. The Process provides rules for translation between the informal goal model and the Event-B models and the development method includes decomposition of the system model rather than composition of separate requirement machines.

The work on UML-B is mature and has tool support that allows automatic translation between the UML-B models and the B models. The UML entities have been adapted to fit with the translation to B notation. The goal models of the Process have been developed to be translated into the Event-B models, but the translation is not systematic enough to be automated. Current work to integrate UML and Event-B in the RODIN toolset (Snook and Butler (2008)) uses a meta-model to integrate the Event-B modelling elements with a UML-style modelling language. This is more related to the guidance given to translate the entities of the goal model to the Event-B models because the modelling elements in the graphical notation have been developed to represent the formal elements. Work is being carried out integrating refinement into UML-B. Each UML-B class diagram models a single level of abstraction, whereas the goal models of the Process model an Event-B refinement chain.

The refinement diagrams in Back (2005) are used to develop architectural models of systems. The Process develops behavioural models of multi-agent systems and refines them. The purpose of the refinement diagrams is to help ensure the correctness of changes to the architecture introduced by changing requirements. The purpose of the Process is to capture agent concepts in Event-B models of multi-agent systems.

9.6 Future Work

There is potential for the Process to become more systematic in the translation between the goal diagram and the Event-B models. A set of systematic transformations between the informal and formal models could be automated by a software program. This would further ease the burden on the developer by removing the need for them to apply the transformations and by potentially allowing them to work without expert knowledge of the formal models. It may be possible to capture the final steps of the Process as a series of refinement patterns.

The Process produces Event-B abstract machines of the agent roles required by the system. These abstract machines can be further refined to model how the internal reasoning of the agent works to achieve the goals that it has been allocated. Extending the Process to help the developer refine the agent components to include agent reasoning will make the Process a more complete solution for the formal modelling of multi-agent systems in Event-B. This may require additional modelling elements, that could have attributes based on logics, that represent knowledge and belief, to be added to the Event-B method to enable the internal representation of agents to be modelled.

The approach to developing the fault-tolerance patterns presented in this thesis can be applied to other common aspects of multi-agent and distributed system development. Standard platform agents or resources are examples of aspects of multi-agent systems that could be constructed as patterns using the same method.

Another useful avenue for future work would be to gather feedback from developers and experts in AOSE on using the Process in other case studies and developments. This can be facilitated through publication of the work from this thesis in conferences and workshops.

Appendix A

Query-If Case Study Event-B Models

A.1 Context

CONTEXT context

SETS

CONVERSATION
AGENT

END

A.2 m0 - Abstract Machine

MACHINE m0

SEES context

VARIABLES

$queried$
$completed$
$refused$

INVARIANTS

$inv1 : queried \in CONVERSATION \nrightarrow AGENT$
$inv2 : completed \subseteq queried$
$inv3 : refused \subseteq queried$
$inv4 : completed \cap refused = \varnothing$

EVENTS

INITIALISATION

BEGIN
$act1 : queried := \varnothing$
$act2 : completed := \varnothing$
$act3 : refused := \varnothing$
END

EVENT query
ANY
c
a
WHERE
$grd1 : c \in CONVERSATION$
$grd2 : a \in AGENT$
$grd3 : c \notin dom(queried)$
THEN
$act1 : queried := queried \cup \{c \mapsto a\}$
END

EVENT complete

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queried$

$grd2 : c \mapsto a \notin completed$

$grd3 : c \mapsto a \notin refused$

THEN

$act1 : completed := completed \cup \{c \mapsto a\}$

END

EVENT refuse

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queried$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin completed$

THEN

$act1 : refused := refused \cup \{c \mapsto a\}$

END

END

A.3 m1 - First Refinement

MACHINE m1

REFINES m0

SEES context

VARIABLES

$queried$
$refused$
$accepted$
$informed$
$failed$

INVARIANTS

$inv1 : accepted \subseteq queried$
$inv2 : informed \subseteq accepted$
$inv3 : failed \subseteq accepted$
$inv4 : accepted \cap refused = \varnothing$
$inv5 : completed = informed \cup failed$
$inv6 : informed \cap failed = \varnothing$

EVENTS

INITIALISATION

BEGIN
$act1 : queried := \varnothing$
$act3 : refused := \varnothing$
$act2 : accepted := \varnothing$
$act4 : informed := \varnothing$
$act5 : failed := \varnothing$
END

EVENT query
REFINES query
ANY
c

a

WHERE

$grd1 : c \in CONVERSATION$

$grd2 : a \in AGENT$

$grd3 : c \notin dom(queried)$

THEN

$act1 : queried := queried \cup \{c \mapsto a\}$

END

EVENT refuse

REFINES refuse

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queried$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin accepted$

THEN

$act1 : refused := refused \cup \{c \mapsto a\}$

END

EVENT accept

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queried$

$grd2 : c \mapsto a \notin accepted$

$grd3 : c \mapsto a \notin refused$

THEN

$act1 : accepted := accepted \cup \{c \mapsto a\}$

END

EVENT inform

REFINES complete

ANY

c

a

WHERE

$grd1 : c \mapsto a \in accepted$

$grd2 : c \mapsto a \notin informed$
$grd3 : c \mapsto a \notin failed$

THEN

$act1 : informed := informed \cup \{c \mapsto a\}$

END

EVENT fail
REFINES complete

ANY

c
a

WHERE

$grd1 : c \mapsto a \in accepted$
$grd2 : c \mapsto a \notin failed$
$grd3 : c \mapsto a \notin informed$

THEN

$act1 : failed := failed \cup \{c \mapsto a\}$

END

END

A.4 m2 - Second Refinement

MACHINE m2

REFINES m1

SEES context

VARIABLES

queried
refused
accepted
informed
failed
queryS
queryR
refuseS
refuseR
acceptS
acceptR
informS
informR
failS
failR
initiator
participant

INVARIANTS

$inv1 : queryS \subseteq queried$
$inv2 : queryR \subseteq queryS$
$inv3 : refuseS \subseteq queryR$
$inv4 : refuseR \subseteq refuseS$
$inv5 : acceptS \subseteq queryR$
$inv6 : acceptR \subseteq acceptS$
$inv7 : informS \subseteq acceptS$
$inv8 : informR \subseteq informS$
$inv9 : failS \subseteq acceptS$
$inv10 : failR \subseteq failS$
$inv12 : initiator \in dom(queryS) \rightarrow AGENT$

$inv13 : participant \in dom(queryS) \rightarrow AGENT$
$inv14 : refuseS \subseteq refused$
$inv15 : acceptS \subseteq accepted$
$inv16 : informS \subseteq informed$
$inv17 : failS \subseteq failed$

EVENTS

INITIALISATION

BEGIN
$act1 : queried := \varnothing$
$act3 : refused := \varnothing$
$act2 : accepted := \varnothing$
$act4 : informed := \varnothing$
$act5 : failed := \varnothing$
$act6 : queryS := \varnothing$
$act7 : queryR := \varnothing$
$act8 : refuseS := \varnothing$
$act9 : refuseR := \varnothing$
$act10 : acceptS := \varnothing$
$act11 : acceptR := \varnothing$
$act12 : informS := \varnothing$
$act13 : informR := \varnothing$
$act14 : failS := \varnothing$
$act15 : failR := \varnothing$
$act16 : initiator := \varnothing$
$act17 : participant := \varnothing$
END

EVENT sendQuery
REFINES query
ANY
c
a
ar
WHERE
$grd1 : c \in CONVERSATION$
$grd2 : a \in AGENT$
$grd5 : ar \in AGENT$
$grd4 : c \notin dom(queried)$

$grd3 : a \neq ar$

THEN

$act2 : queryS := queryS \cup \{c \mapsto a\}$

$act1 : initiator := initiator \cup \{c \mapsto a\}$

$act3 : participant := participant \cup \{c \mapsto ar\}$

$act4 : queried := queried \cup \{c \mapsto a\}$

END

EVENT receiveQuery

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queryS$

$grd2 : c \mapsto a \notin queryR$

$grd4 : c \mapsto a \in initiator$

THEN

$act1 : queryR := queryR \cup \{c \mapsto a\}$

END

EVENT sendRefuse
REFINES refuse

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queryR$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin accepted$

$grd4 : c \mapsto a \in initiator$

THEN

$act1 : refuseS := refuseS \cup \{c \mapsto a\}$

$act2 : refused := refused \cup \{c \mapsto a\}$

END

EVENT receiveRefuse

ANY

c

a

WHERE

$grd1 : c \mapsto a \in refuseS$

$grd2 : c \mapsto a \notin refuseR$

$grd3 : c \mapsto a \in initiator$

THEN

$act1 : refuseR := refuseR \cup \{c \mapsto a\}$

END

EVENT sendAccept
REFINES accept

ANY

c

a

WHERE

$grd1 : c \mapsto a \in queryR$

$grd2 : c \mapsto a \notin accepted$

$grd3 : c \mapsto a \notin refused$

$grd4 : c \mapsto a \in initiator$

THEN

$act1 : acceptS := acceptS \cup \{c \mapsto a\}$

$act2 : accepted := accepted \cup \{c \mapsto a\}$

END

EVENT receiveAccept

ANY

c

a

WHERE

$grd1 : c \mapsto a \in acceptS$

$grd2 : c \mapsto a \notin acceptR$

$grd3 : c \mapsto a \in initiator$

THEN

$act1 : acceptR := acceptR \cup \{c \mapsto a\}$

END

EVENT sendInform
REFINES inform

ANY

c

a

WHERE

$grd5 : c \mapsto a \in accepted$

$grd2 : c \mapsto a \notin informed$

$grd3 : c \mapsto a \notin failed$

$grd4 : c \mapsto a \in initiator$

THEN

$act1 : informS := informS \cup \{c \mapsto a\}$

$act2 : informed := informed \cup \{c \mapsto a\}$

END

EVENT receiveInform

ANY

c

a

WHERE

$grd1 : c \mapsto a \in informS$

$grd2 : c \mapsto a \notin informR$

$grd3 : c \mapsto a \in initiator$

THEN

$act1 : informR := informR \cup \{c \mapsto a\}$

END

EVENT sendFail

REFINES fail

ANY

c

a

WHERE

$grd5 : c \mapsto a \in accepted$

$grd2 : c \mapsto a \notin failed$

$grd3 : c \mapsto a \notin informed$

$grd4 : c \mapsto a \in initiator$

THEN

$act1 : failS := failS \cup \{c \mapsto a\}$

$act2 : failed := failed \cup \{c \mapsto a\}$

END

EVENT receiveFail

ANY

c

a

WHERE

$grd1 : c \mapsto a \in failS$

$grd2 : c \mapsto a \notin failR$

$grd3 : c \mapsto a \in initiator$

THEN

$act1 : failR := failR \cup \{c \mapsto a\}$

END

END

A.5 Context 2

CONTEXT context2

REFINES context

SETS

MESSAGE

CONSTANTS

sender
receiver
messageConversation

AXIOMS

$axm1 : sender \in MESSAGE \rightarrow AGENT$
$axm2 : receiver \in MESSAGE \rightarrow AGENT$
$axm3 : messageConversation \in MESSAGE \rightarrow CONVERSATION$

END

A.6 m3 - Third Refinement

MACHINE m3

REFINES m2

SEES context2

VARIABLES

$queried$
$refused$
$accepted$
$informed$
$failed$
$queryS$
$queryR$
$refuseS$
$refuseR$
$acceptS$
$acceptR$
$informS$
$informR$
$failS$
$failR$
$initiator$
$participant$
$queryM$
$refuseM$
$acceptM$
$informM$
$failM$
$pInitiator$
$pParticipant$

INVARIANTS

$inv1 : queryM \subseteq MESSAGE$
$inv2 : refuseM \subseteq MESSAGE$
$inv3 : acceptM \subseteq MESSAGE$
$inv4 : informM \subseteq MESSAGE$

$inv5 : failM \subseteq MESSAGE$
$inv6 : queryS = (queryM \lhd messageConversation)^{-1};\ sender$
$inv7 : refuseS = (refuseM \lhd messageConversation)^{-1};\ receiver$
$inv8 : acceptS = (acceptM \lhd messageConversation)^{-1};\ receiver$
$inv9 : informS = (informM \lhd messageConversation)^{-1};\ receiver$
$inv10 : failS = (failM \lhd messageConversation)^{-1};\ receiver$
$inv11 : pInitiator \in dom(queryR) \rightarrow AGENT$
$inv12 : pParticipant \in dom(queryR) \rightarrow AGENT$
$inv13 : pInitiator \subseteq initiator$
$inv14 : pParticipant \subseteq participant$
$inv15 : initiator = (queryM \lhd messageConversation)^{-1};\ sender$
$inv16 : participant = (queryM \lhd messageConversation)^{-1};\ receiver$

EVENTS

INITIALISATION

BEGIN

$act1 : queried := \varnothing$
$act3 : refused := \varnothing$
$act2 : accepted := \varnothing$
$act4 : informed := \varnothing$
$act5 : failed := \varnothing$
$act6 : queryS := \varnothing$
$act7 : queryR := \varnothing$
$act8 : refuseS := \varnothing$
$act9 : refuseR := \varnothing$
$act10 : acceptS := \varnothing$
$act11 : acceptR := \varnothing$
$act12 : informS := \varnothing$
$act13 : informR := \varnothing$
$act14 : failS := \varnothing$
$act15 : failR := \varnothing$
$act16 : initiator := \varnothing$
$act17 : participant := \varnothing$
$act18 : queryM := \varnothing$
$act19 : refuseM := \varnothing$
$act20 : acceptM := \varnothing$
$act21 : informM := \varnothing$
$act22 : failM := \varnothing$
$act23 : pInitiator := \varnothing$

$act24 : pParticipant := \varnothing$

END

EVENT sendQuery

REFINES sendQuery

ANY

c

a

ar

m

WHERE

$grd1 : c \in CONVERSATION$

$grd2 : a \in AGENT$

$grd4 : c \notin dom(queried)$

$grd3 : a \neq ar$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : receiver(m) = ar$

$grd8 : messageConversation(m) = c$

THEN

$act2 : queryS := queryS \cup \{c \mapsto a\}$

$act1 : initiator := initiator \cup \{c \mapsto a\}$

$act3 : participant := participant \cup \{c \mapsto ar\}$

$act4 : queried := queried \cup \{c \mapsto a\}$

$act5 : queryM := queryM \cup \{m\}$

END

EVENT receiveQuery

REFINES receiveQuery

ANY

c

a

m

WHERE

$grd1 : m \in queryM$

$grd2 : c \mapsto a \notin queryR$

$grd3 : sender(m) = a$

$grd6 : messageConversation(m) = c$

THEN

$act1 : queryR := queryR \cup \{c \mapsto a\}$

$act2 : pInitiator := pInitiator \cup \{c \mapsto a\}$

$act3 : pParticipant := pParticipant \cup \{c \mapsto receiver(m)\}$

END

EVENT sendRefuse

REFINES sendRefuse

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in queryR$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin accepted$

$grd4 : c \mapsto a \in pInitiator$

$grd5 : m \in MESSAGE$

$grd6 : receiver(m) = a$

$grd7 : sender(m) = pParticipant(c)$

$grd8 : messageConversation(m) = c$

THEN

$act1 : refuseS := refuseS \cup \{c \mapsto a\}$

$act2 : refused := refused \cup \{c \mapsto a\}$

$act3 : refuseM := refuseM \cup \{m\}$

END

EVENT receiveRefuse

REFINES receiveRefuse

ANY

c

a

m

WHERE

$grd1 : m \in refuseM$

$grd2 : c \mapsto a \notin refuseR$

$grd3 : c \mapsto a \in initiator$

$grd4 : sender(m) = participant(c)$

$grd5 : receiver(m) = a$

$grd6 : messageConversation(m) = c$

THEN

$act1 : refuseR := refuseR \cup \{c \mapsto a\}$

END

EVENT sendAccept
REFINES sendAccept
ANY
c
a
m
WHERE
$grd1 : c \mapsto a \in queryR$
$grd2 : c \mapsto a \notin accepted$
$grd3 : c \mapsto a \notin refused$
$grd4 : c \mapsto a \in pInitiator$
$grd5 : m \in MESSAGE$
$grd6 : sender(m) = pParticipant(c)$
$grd7 : receiver(m) = a$
$grd8 : messageConversation(m) = c$
THEN
$act1 : acceptS := acceptS \cup \{c \mapsto a\}$
$act2 : accepted := accepted \cup \{c \mapsto a\}$
$act3 : acceptM := acceptM \cup \{m\}$
END

EVENT receiveAccept
REFINES receiveAccept
ANY
c
a
m
WHERE
$grd1 : m \in acceptM$
$grd2 : c \mapsto a \notin acceptR$
$grd3 : c \mapsto a \in initiator$
$grd4 : sender(m) = participant(c)$
$grd5 : receiver(m) = a$
$grd6 : messageConversation(m) = c$
THEN
$act1 : acceptR := acceptR \cup \{c \mapsto a\}$
END

EVENT sendInform
REFINES sendInform
ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in acceptS$

$grd2 : c \mapsto a \notin informed$

$grd3 : c \mapsto a \notin failed$

$grd4 : c \mapsto a \in pInitiator$

$grd5 : c \mapsto a \in accepted$

$grd6 : m \in MESSAGE$

$grd7 : sender(m) = pParticipant(c)$

$grd8 : receiver(m) = a$

$grd9 : messageConversation(m) = c$

THEN

$act1 : informS := informS \cup \{c \mapsto a\}$

$act2 : informed := informed \cup \{c \mapsto a\}$

$act3 : informM := informM \cup \{m\}$

END

EVENT receiveInform

REFINES receiveInform

ANY

c

a

m

WHERE

$grd1 : m \in informM$

$grd2 : c \mapsto a \notin informR$

$grd3 : c \mapsto a \in initiator$

$grd4 : sender(m) = participant(c)$

$grd5 : receiver(m) = a$

$grd6 : messageConversation(m) = c$

THEN

$act1 : informR := informR \cup \{c \mapsto a\}$

END

EVENT sendFail

REFINES sendFail

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in acceptS$

$grd2 : c \mapsto a \notin failed$

$grd3 : c \mapsto a \notin informed$

$grd4 : c \mapsto a \in pInitiator$

$grd5 : c \mapsto a \in accepted$

$grd6 : m \in MESSAGE$

$grd7 : sender(m) = pParticipant(c)$

$grd8 : receiver(m) = a$

$grd9 : messageConversation(m) = c$

THEN

$act1 : failS := failS \cup \{c \mapsto a\}$

$act2 : failed := failed \cup \{c \mapsto a\}$

$act3 : failM := failM \cup \{m\}$

END

EVENT receiveFail
REFINES receiveFail

ANY

c

a

m

WHERE

$grd1 : m \in failM$

$grd2 : c \mapsto a \notin failR$

$grd3 : c \mapsto a \in initiator$

$grd4 : sender(m) = participant(c)$

$grd5 : receiver(m) = a$

$grd6 : messageConversation(m) = c$

THEN

$act1 : failR := failR \cup \{c \mapsto a\}$

END

END

A.7 Context 3

CONTEXT context3

REFINES context2

SETS

$PROPOSITION$

CONSTANTS

$QUERY$
$ACCEPT$
$REFUSE$
$INFORM$
$FAIL$
$question$
$answer$

AXIOMS

$axm1 : QUERY \subseteq MESSAGE$
$axm2 : ACCEPT \subseteq MESSAGE$
$axm3 : REFUSE \subseteq MESSAGE$
$axm4 : INFORM \subseteq MESSAGE$
$axm5 : FAIL \subseteq MESSAGE$
$axm5 : question \in QUERY \rightarrow PROPOSITION$
$axm5 : answer \in INFORM \rightarrow BOOL$
$axm6 : QUERY \cap ACCEPT = \varnothing$
$axm7 : QUERY \cap REFUSE = \varnothing$
$axm8 : QUERY \cap INFORM = \varnothing$
$axm9 : QUERY \cap FAIL = \varnothing$
$axm10 : ACCEPT \cap REFUSE = \varnothing$
$axm11 : ACCEPT \cap INFORM = \varnothing$
$axm12 : ACCEPT \cap FAIL = \varnothing$
$axm13 : REFUSE \cap INFORM = \varnothing$
$axm14 : REFUSE \cap FAIL = \varnothing$
$axm15 : INFORM \cap FAIL = \varnothing$

END

A.8 m4 - Fourth Refinement

MACHINE m4

REFINES m3

SEES context3

VARIABLES

$queried$
$refused$
$accepted$
$informed$
$failed$
$queryS$
$queryR$
$refuseS$
$refuseR$
$acceptS$
$acceptR$
$informS$
$informR$
$failS$
$failR$
$initiator$
$participant$
$pInitiator$
$pParticipant$
$msgset$

INVARIANTS

$inv1 : msgset \subseteq MESSAGE$
$inv2 : queryM = msgset \cap QUERY$
$inv3 : refuseM = msgset \cap REFUSE$
$inv4 : acceptM = msgset \cap ACCEPT$
$inv5 : informM = msgset \cap INFORM$
$inv6 : failM = msgset \cap FAIL$

EVENTS

INITIALISATION

BEGIN

$act1 : queried := \varnothing$

$act3 : refused := \varnothing$

$act2 : accepted := \varnothing$

$act4 : informed := \varnothing$

$act5 : failed := \varnothing$

$act6 : queryS := \varnothing$

$act7 : queryR := \varnothing$

$act8 : refuseS := \varnothing$

$act9 : refuseR := \varnothing$

$act10 : acceptS := \varnothing$

$act11 : acceptR := \varnothing$

$act12 : informS := \varnothing$

$act13 : informR := \varnothing$

$act14 : failS := \varnothing$

$act15 : failR := \varnothing$

$act16 : initiator := \varnothing$

$act17 : participant := \varnothing$

$act23 : pInitiator := \varnothing$

$act24 : pParticipant := \varnothing$

$act18 : msgset := \varnothing$

END

EVENT sendQuery

REFINES sendQuery

ANY

c

a

ar

m

WHERE

$grd1 : c \in CONVERSATION$

$grd2 : a \in AGENT$

$grd4 : c \notin dom(queried)$

$grd3 : a \neq ar$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : receiver(m) = ar$

$grd8 : messageConversation(m) = c$
$grd9 : m \in QUERY$
$grd10 : question(m) \in PROPOSITION$

THEN

$act2 : queryS := queryS \cup \{c \mapsto a\}$
$act1 : initiator := initiator \cup \{c \mapsto a\}$
$act3 : participant := participant \cup \{c \mapsto ar\}$
$act4 : queried := queried \cup \{c \mapsto a\}$
$act5 : msgset := msgset \cup \{m\}$

END

EVENT receiveQuery
REFINES receiveQuery

ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin queryR$
$grd3 : sender(m) = a$
$grd6 : messageConversation(m) = c$
$grd4 : m \in QUERY$
$grd5 : question(m) \in PROPOSITION$

THEN

$act1 : queryR := queryR \cup \{c \mapsto a\}$
$act2 : pInitiator := pInitiator \cup \{c \mapsto a\}$
$act3 : pParticipant := pParticipant \cup \{c \mapsto receiver(m)\}$

END

EVENT sendRefuse
REFINES sendRefuse

ANY

c
a
m

WHERE

$grd1 : c \mapsto a \in queryR$
$grd2 : c \mapsto a \notin refused$
$grd3 : c \mapsto a \notin accepted$
$grd4 : c \mapsto a \in pInitiator$

$grd5 : m \in MESSAGE$
$grd6 : receiver(m) = a$
$grd7 : sender(m) = pParticipant(c)$
$grd8 : messageConversation(m) = c$
$grd9 : m \in REFUSE$

THEN

$act1 : refuseS := refuseS \cup \{c \mapsto a\}$
$act2 : refused := refused \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveRefuse
REFINES receiveRefuse

ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin refuseR$
$grd3 : c \mapsto a \in initiator$
$grd4 : sender(m) = participant(c)$
$grd5 : receiver(m) = a$
$grd6 : messageConversation(m) = c$
$grd7 : m \in REFUSE$

THEN

$act1 : refuseR := refuseR \cup \{c \mapsto a\}$

END

EVENT sendAccept
REFINES sendAccept

ANY

c
a
m

WHERE

$grd1 : c \mapsto a \in queryR$
$grd2 : c \mapsto a \notin accepted$
$grd3 : c \mapsto a \notin refused$
$grd4 : c \mapsto a \in pInitiator$
$grd5 : m \in MESSAGE$

$grd6 : sender(m) = pParticipant(c)$
$grd7 : receiver(m) = a$
$grd8 : messageConversation(m) = c$
$grd9 : m \in ACCEPT$

THEN

$act1 : acceptS := acceptS \cup \{c \mapsto a\}$
$act2 : accepted := accepted \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveAccept
REFINES receiveAccept

ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin acceptR$
$grd3 : c \mapsto a \in initiator$
$grd4 : sender(m) = participant(c)$
$grd5 : receiver(m) = a$
$grd6 : messageConversation(m) = c$
$grd7 : m \in ACCEPT$

THEN

$act1 : acceptR := acceptR \cup \{c \mapsto a\}$

END

EVENT sendInform
REFINES sendInform

ANY

c
a
m

WHERE

$grd1 : c \mapsto a \in acceptS$
$grd2 : c \mapsto a \notin informed$
$grd3 : c \mapsto a \notin failed$
$grd4 : c \mapsto a \in pInitiator$
$grd5 : c \mapsto a \in accepted$
$grd6 : m \in MESSAGE$

$grd7 : sender(m) = pParticipant(c)$
$grd8 : receiver(m) = a$
$grd9 : messageConversation(m) = c$
$grd10 : m \in INFORM$
$grd11 : answer(m) \in BOOL$

THEN

$act1 : informS := informS \cup \{c \mapsto a\}$
$act2 : informed := informed \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveInform
REFINES receiveInform

ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin informR$
$grd3 : c \mapsto a \in initiator$
$grd4 : sender(m) = participant(c)$
$grd5 : receiver(m) = a$
$grd6 : messageConversation(m) = c$
$grd7 : m \in INFORM$
$grd9 : answer(m) \in BOOL$

THEN

$act1 : informR := informR \cup \{c \mapsto a\}$

END

EVENT sendFail
REFINES sendFail

ANY

c
a
m

WHERE

$grd1 : c \mapsto a \in acceptS$
$grd2 : c \mapsto a \notin failed$
$grd3 : c \mapsto a \notin informed$
$grd4 : c \mapsto a \in pInitiator$

$grd5 : c \mapsto a \in accepted$
$grd6 : m \in MESSAGE$
$grd7 : sender(m) = pParticipant(c)$
$grd8 : receiver(m) = a$
$grd9 : messageConversation(m) = c$
$grd10 : m \in FAIL$

THEN

$act1 : failS := failS \cup \{c \mapsto a\}$
$act2 : failed := failed \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveFail
REFINES receiveFail

ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin failR$
$grd3 : c \mapsto a \in initiator$
$grd4 : sender(m) = participant(c)$
$grd5 : receiver(m) = a$
$grd6 : messageConversation(m) = c$
$grd7 : m \in FAIL$

THEN

$act1 : failR := failR \cup \{c \mapsto a\}$

END

END

A.9 m5 - Fifth Refinement

REFINEMENT m5

REFINES m4

SEES context3

INCLUDES initiator.initiator0, participant.participant0, mw.middleware0

INVARIANT

$$
\begin{aligned}
&queried = initiator.queried \land \\
&refused = participant.refused \land \\
&accepted = participant.accepted \land \\
&informed = participant.informed \land \\
&failed = participant.failed \land \\
&queryS = initiator.queryS \land \\
&acceptS = participant.acceptS \land \\
&refuseS = participant.refuseS \land \\
&informS = participant.informS \land \\
&failureS = participant.failureS \land \\
&queryR = participant.queryR \land \\
&acceptR = initiator.acceptR \land \\
&refuseR = initiator.refuseR \land \\
&informR = initiator.informR \land \\
&failureR = initiator.failureR \land \\
&initiator = initiator.initiator \land \\
&participant = initiator.participant \land \\
&pInitiator = participant.pInitiator \land \\
&pParticipant = participant.pParticipant \land \\
&msgset = mw.msgset
\end{aligned}
$$

EVENTS

EVENT sendQuery =

VAR

m

IN

$m \leftarrow initiator.sendQuery;$

$mw.send(m)$

END;

EVENT receiveQuery =

ANY

a

WHERE

$a \in AGENT$

THEN

VAR

m

IN

$m \leftarrow mw.receive(a);$

$participant.receiveQuery(m)$

END

END;

EVENT sendAccept =

VAR

m

IN

$m \leftarrow participant.sendAccept;$

$mw.send(m)$

END;

EVENT receiveAccept =

ANY

a

WHERE

$a \in AGENT$

THEN

VAR

m

IN

$m \leftarrow mw.receive(a);$

$initiator.receiveAccept(m)$

END

END;

EVENT sendRefuse =
  VAR
    m
  IN
    $m \leftarrow participant.sendRefuse;$
    $mw.send(m)$
  END;

EVENT receiveRefuse =
  ANY
    a
  WHERE
    $a \in AGENT$
  THEN
    VAR
      m
    IN
      $m \leftarrow mw.receive(a);$
      $initiator.receiveRefuse(m)$
    END
  END;

EVENT sendInform =
  VAR
    m
  IN
    $m \leftarrow participant.sendInform;$
    $mw.send(m)$
  END;

EVENT receiveInform =
  ANY
    a
  WHERE
    $a \in AGENT$
  THEN
    VAR

m **IN**
$m \leftarrow mw.receive(a);$
$initiator.receiveInform(m)$
END
END;

EVENT sendFail =
VAR
m
IN
$m \leftarrow participant.sendFail;$
$mw.send(m)$
END;

EVENT receiveFail =
ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$initiator.receiveFail(m)$
END
END

END

A.10 Initiator - Component Model

MODEL initiator0

SEES context

VARIABLES queried, queryS, acceptR, refuseR, informR, failureR, initiator, participant

INVARIANT

$queried \in CONVERSATION \nrightarrow AGENT \wedge$
$queryS \in CONVERSATION \nrightarrow AGENT \wedge$
$acceptR \in CONVERSATION \nrightarrow AGENT \wedge$
$refuseR \in CONVERSATION \nrightarrow AGENT \wedge$
$informR \in CONVERSATION \nrightarrow AGENT \wedge$
$failureR \in CONVERSATION \nrightarrow AGENT \wedge$
$initiator \in dom(queryS) \rightarrow AGENT \wedge$
$participant \in dom(queryS) \rightarrow AGENT$

INITIALISATION

$queried := \varnothing \;||$
$queryS := \varnothing \;||$
$acceptR := \varnothing \;||$
$refuseR := \varnothing \;||$
$informR := \varnothing \;||$
$failureR := \varnothing \;||$
$initiator := \varnothing \;||$
$participant := \varnothing$

EVENTS

EVENT mr $\leftarrow$ sendQuery =

ANY

c
a
ar
m

WHERE

$c \in CONVERSATION \wedge$
$a \in AGENT \wedge$
$ar \in AGENT \wedge$
$c \notin dom(queried) \wedge$
$a \neq ar \wedge$
$m \in MESSAGE \wedge$
$sender(m) = a \wedge$
$receiver(m) = ar \wedge$
$messageConversation(m) = c \wedge$
$m \in QUERY \wedge$
$question(m) \in PROPOSITION$

THEN

$queryS := queryS \cup \{c \mapsto a\} \parallel$
$initiator := intiator \cup \{c \mapsto a\} \parallel$
$participant := participant \cup \{c \mapsto ar\} \parallel$
$queried := queried \cup \{c \mapsto a\} \parallel$
$mr := m$
END;

EVENT receiveAccept(m) =
PRE
$m \in MESSAGE$
THEN
ANY
c
a
WHERE
$c \mapsto a \notin acceptR \wedge$
$c \mapsto a \in initiator \wedge$
$sender(m) = participant(c) \wedge$
$receiver(m) = a \wedge$
$messageConversation(m) = c \wedge$
$m \in ACCEPT$
THEN
$acceptR := acceptR \cup \{c \mapsto a\}$
END
END;

EVENT receiveRefuse(m) =
PRE
$m \in MESSAGE$
THEN
ANY
c
a
WHERE
$c \mapsto a \notin refuseR \wedge$
$c \mapsto a \in initiator \wedge$
$sender(m) = participant(c) \wedge$
$receiver(m) = a \wedge$
$messageConversation(m) = c \wedge$

$m \in REFUSE$

THEN

$refuseR := refuseR \cup \{c \mapsto a\}$

END

END;

EVENT receiveInform(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin informR \wedge$

$c \mapsto a \in initiator \wedge$

$sender(m) = participant(c) \wedge$

$receiver(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in INFORM \wedge$

$answer(m) \in BOOL$

THEN

$informR := informR \cup \{c \mapsto a\}$

END

END;

EVENT receiveFail(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin failR \wedge$

$c \mapsto a \in initiator \wedge$

$sender(m) = participant(c) \wedge$

$receiver(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in FAIL$
THEN
$failureR := failureR \cup \{c \mapsto a\}$
END
END

END

A.11 Participant - Component Model

MODEL participant0

SEES context

VARIABLES refused, accepted, informed, failed, acceptS, refuseS, informS, failureS, queryR, pInitiator, pParticipant

INVARIANT
$refused \in CONVERSATION \nrightarrow AGENT \wedge$
$accepted \in CONVERSATION \nrightarrow AGENT \wedge$
$informed \in CONVERSATION \nrightarrow AGENT \wedge$
$failed \in CONVERSATION \nrightarrow AGENT \wedge$
$acceptS \in CONVERSATION \nrightarrow AGENT \wedge$
$refuseS \in CONVERSATION \nrightarrow AGENT \wedge$
$informS \in CONVERSATION \nrightarrow AGENT \wedge$
$failureS \in CONVERSATION \nrightarrow AGENT \wedge$
$queryR \in CONVERSATION \nrightarrow AGENT \wedge$
$pInitiator \in dom(queryR) \rightarrow AGENT \wedge$
$pParticipant \in dom(queryR) \rightarrow AGENT$

INITIALISATION
$refused := \varnothing \,||$
$accepted := \varnothing \,||$
$informed := \varnothing \,||$
$failed := \varnothing \,||$
$acceptS := \varnothing \,||$
$refuseS := \varnothing \,||$
$informS := \varnothing \,||$
$failureS := \varnothing \,||$
$queryR := \varnothing \,||$
$pInitiator := \varnothing \,||$

$pParticipant := \varnothing$

EVENTS

EVENT receiveQuery(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin queryR \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in QUERY \wedge$

$question(m) \in PROPOSITION$

THEN

$queryR := queryR \cup \{c \mapsto a\} \parallel$

$pParticipant := pParticipant \cup \{c \mapsto receiver(m)\} \parallel$

$pInitiator := pInitiator \cup \{c \mapsto a\}$

END

END;

EVENT mr $\leftarrow$ sendAccept =

ANY

c

a

WHERE

$c \mapsto a \in queryR \wedge$

$c \mapsto a \notin accepted \wedge$

$c \mapsto a \notin refused \wedge$

$c \mapsto a \in pInitiator \wedge$

$m \in MESSAGE \wedge$

$sender(m) = pParticipant(c) \wedge$

$receiver(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in ACCEPT$

THEN

$acceptS := acceptS \cup \{c \mapsto a\} \parallel$

$accepted := accepted \cup \{c \mapsto a\} \parallel$

$mr := m$

END;

EVENT mr ← sendRefuse =

ANY

c

a

WHERE

$c \mapsto a \in queryR \wedge$

$c \mapsto a \notin refused \wedge$

$c \mapsto a \notin accepted \wedge$

$c \mapsto a \in pInitiator \wedge$

$m \in MESSAGE \wedge$

$receiver(m) = a \wedge$

$sender(m) = pParticipant(c) \wedge$

$messageConversation(m) = c \wedge$

$m \in REFUSE$

THEN

$refuseS := refuseS \cup \{c \mapsto a\} \ ||$

$refused := refused \cup \{c \mapsto a\} \ ||$

$mr := m$

END;

EVENT mr ← sendInform =

ANY

c

a

WHERE

$c \mapsto a \in acceptS \wedge$

$c \mapsto a \notin informed \wedge$

$c \mapsto a \notin failed \wedge$

$c \mapsto a \in pInitiator \wedge$

$c \mapsto a \in accepted \wedge$

$m \in MESSAGE \wedge$

$sender(m) = pParticipant(c) \wedge$

$receiver(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in INFORM \wedge$

$answer(m) \in BOOL$

THEN

$informS := informS \cup \{c \mapsto a\} \;||$

$informed := informed \cup \{c \mapsto a\} \;||$

$mr := m$

END;

EVENT mr $\leftarrow$ sendFailure =

ANY

c

a

WHERE

$c \mapsto a \in acceptS \wedge$

$c \mapsto a \notin failed) \wedge$

$c \mapsto a \notin informed \wedge$

$c \mapsto a \in pInitiator \wedge$

$c \mapsto a \in accepted \wedge$

$m \in MESSAGE \wedge$

$sender(m) = pParticipant(c) \wedge$

$receiver(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in FAIL$

THEN

$failS := failS \cup \{c \mapsto a\} \;||$

$failed := failed \cup \{c \mapsto a\} \;||$

$mr := m$

END

END

A.12 Middleware - Component Model

MODEL middleware0

SEES context

VARIABLES msgset

INVARIANT

$msgset \subseteq MESSAGE$

INITIALISATION

$msgset := \varnothing$

EVENTS

EVENT send(m) =

PRE

$m \in MESSAGE$

THEN

$msgset := msgset \cup \{m\}$

END;

EVENT m $\leftarrow$ receive(a) =

PRE

$a \in AGENT$

THEN

ANY

$m0$

WHERE

$m0 \in MESSAGE \wedge$

$receiver(m0) = a \wedge$

$m0 \in msgset$

THEN

$m := m0$

END

END

END

Appendix B

Contract Net Case Study Event-B Models

B.1 Context

CONTEXT context

SETS

CONVERSATION
AGENT

END

B.2 m0 - Abstract Machine

MACHINE m0

SEES context

VARIABLES

cfp
$responded$
$selected$
$completed$
$cancelled$

INVARIANTS

$inv1 : cfp \in CONVERSATION \leftrightarrow AGENT$
$inv2 : responded \subseteq cfp$
$inv3 : selected \subseteq responded$
$inv4 : completed \subseteq selected$
$inv5 : cancelled \subseteq cfp$

EVENTS

INITIALISATION

BEGIN
$act1 : cfp := \varnothing$
$act2 : responded := \varnothing$
$act3 : selected := \varnothing$
$act4 : completed := \varnothing$
$act5 : cancelled := \varnothing$
END

EVENT callForProposals
ANY
c
as
WHERE
$grd1 : c \in CONVERSATION$
$grd2 : c \notin dom(cfp)$

$grd3 : as \in CONVERSATION \leftrightarrow AGENT$

$grd4 : dom(as) = \{c\}$

$grd5 : ran(as) \subseteq AGENT$

THEN

$act1 : cfp := cfp \cup as$

END

EVENT respond

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfp$

$grd2 : c \mapsto a \notin responded$

THEN

$act1 : responded := responded \cup \{c \mapsto a\}$

END

EVENT select

ANY

c

as

WHERE

$grd3 : c \in dom(responded)$

$grd1 : as \subseteq responded$

$grd2 : as \cap selected = \varnothing$

THEN

$act1 : selected := selected \cup as$

END

EVENT complete

ANY

c

as

WHERE

$grd3 : c \in dom(selected)$

$grd1 : as \subseteq \{c\} \lhd selected$

$grd2 : c \notin dom(completed)$

THEN

$act1 : completed := completed \cup as$

END

EVENT cancel

ANY

c

as

WHERE

$grd1 : c \in dom(cfp)$

$grd2 : c \notin dom(cancelled)$

$grd3 : as \subseteq \{c\} \lhd cfp$

THEN

$act1 : cancelled := cancelled \cup as$

END

END

B.3 m1 - First Refinement

MACHINE m1

REFINES m0

SEES context

VARIABLES

cfp
$proposed$
$refused$
$accepted$
$rejected$
$informed$
$failed$
$cancelStarted$
$informCancelled$
$failCancelled$
$selected1$
$completed$
$cancelled$

INVARIANTS

$inv1 : proposed \subseteq cfp$
$inv2 : refused \subseteq cfp$
$inv3 : proposed \cap refused = \varnothing$
$inv18 : selected1 \subseteq proposed$
$inv4 : accepted \subseteq selected1$
$inv5 : rejected \subseteq proposed$
$inv6 : accepted \cap rejected = \varnothing$
$inv7 : informed \subseteq accepted$
$inv8 : failed \subseteq accepted$
$inv9 : informed \cap failed = \varnothing$
$inv10 : cancelStarted \subseteq cfp$
$inv11 : informCancelled \subseteq cancelStarted$
$inv12 : failCancelled \subseteq cancelStarted$
$inv13 : informCancelled \cap failCancelled = \varnothing$
$inv14 : responded = proposed \cup refused$
$inv15 : selected = accepted \cup rejected$

EVENTS

INITIALISATION

BEGIN
$act1 : cfp := \varnothing$
$act6 : proposed := \varnothing$
$act7 : refused := \varnothing$
$act8 : accepted := \varnothing$
$act9 : rejected := \varnothing$
$act10 : informed := \varnothing$
$act11 : failed := \varnothing$
$act12 : cancelled := \varnothing$
$act13 : informCancelled := \varnothing$
$act14 : failCancelled := \varnothing$
$act2 : selected1 := \varnothing$
$act3 : completed := \varnothing$
$act4 : cancelStarted := \varnothing$
END

EVENT callForProposals

REFINES callForProposals

ANY

c

as

WHERE

$grd1 : c \in CONVERSATION$

$grd2 : c \notin dom(cfp)$

$grd3 : as \in CONVERSATION \leftrightarrow AGENT$

$grd4 : dom(as) = \{c\}$

$grd5 : ran(as) \subseteq AGENT$

THEN

$act1 : cfp := cfp \cup as$

END

EVENT propose

REFINES respond

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfp$

$grd2 : c \mapsto a \notin proposed$

$grd3 : c \mapsto a \notin refused$

THEN

$act1 : proposed := proposed \cup \{c \mapsto a\}$

END

EVENT refuse

REFINES respond

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfp$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin proposed$

THEN

$act1 : refused := refused \cup \{c \mapsto a\}$

END

EVENT select1

ANY

c

as

WHERE

$grd1 : c \in dom(proposed)$

$grd2 : as \subseteq \{c\} \lhd proposed$

$grd3 : c \notin dom(selected1)$

THEN

$act1 : selected1 := selected1 \cup as$

END

EVENT accept
REFINES select

ANY

c

as

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(accepted)$

$grd3 : as \subseteq \{c\} \lhd selected1$

$grd4 : as \cap rejected = \varnothing$

THEN

$act1 : accepted := accepted \cup as$

END

EVENT reject
REFINES select

ANY

c

as

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(rejected)$

$grd3 : as = (\{c\} \lhd proposed) \setminus (\{c\} \lhd selected1)$

$grd4 : as \cap accepted = \varnothing$

THEN

$act1 : rejected := rejected \cup as$

END

EVENT inform

ANY

c

a

WHERE

$grd1 : c \mapsto a \in accepted$

$grd2 : c \mapsto a \notin informed$

$grd3 : c \mapsto a \notin failed$

THEN

$act1 : informed := informed \cup \{c \mapsto a\}$

END

EVENT fail

ANY

c

a

WHERE

$grd1 : c \mapsto a \in accepted$

$grd2 : c \mapsto a \notin failed$

$grd3 : c \mapsto a \notin informed$

THEN

$act1 : failed := failed \cup \{c \mapsto a\}$

END

EVENT complete

REFINES complete

ANY

c

as

WHERE

$grd1 : c \in dom(informed \cup failed)$

$grd2 : as \subseteq \{c\} \lhd (informed \cup failed)$

$grd3 : c \notin dom(completed)$

THEN

$act1 : completed := completed \cup as$

END

EVENT cancel1

ANY

c

as

WHERE

$grd1 : c \in dom(cfp)$

$grd2 : as \subseteq \{c\} \lhd cfp$
$grd3 : c \notin dom(cancelStarted)$
THEN
$act1 : cancelStarted := cancelStarted \cup as$
END

EVENT informCancel
ANY
c
a
WHERE
$grd1 : c \mapsto a \in cancelStarted$
$grd2 : c \mapsto a \notin informCancelled$
$grd3 : c \mapsto a \notin failCancelled$
THEN
$act1 : informCancelled := informCancelled \cup \{c \mapsto a\}$
END

EVENT failCancel
ANY
c
a
WHERE
$grd1 : c \mapsto a \in cancelStarted$
$grd2 : c \mapsto a \notin failCancelled$
$grd3 : c \mapsto a \notin informCancelled$
THEN
$act1 : failCancelled := failCancelled \cup \{c \mapsto a\}$
END

EVENT cancelled
REFINES cancel
ANY
c
as
WHERE
$grd1 : c \in dom(informCancelled \cup failCancelled)$
$grd2 : as \subseteq \{c\} \lhd (informCancelled \cup failCancelled)$
$grd3 : c \notin dom(cancelled)$
THEN
$act1 : cancelled := cancelled \cup as$

END

END

B.4 m2 - Second Refinement

MACHINE m2

REFINES m1

SEES context

VARIABLES

cfp
$proposed$
$refused$
$selected1$
$accepted$
$rejected$
$informed$
$failed$
$cancelStarted$
$informCancelled$
$failCancelled$
$cfpS$
$cfpR$
$proposeS$
$proposeR$
$refuseS$
$refuseR$
$acceptS$
$acceptR$
$rejectS$
$rejectR$
$informS$
$informR$
$failS$

$failR$
$cancelS$
$cancelR$
$informCancelS$
$informCancelR$
$failCancelS$
$failCancelR$
$completed$
$cancelled$
$initiator$
$participant$

INVARIANTS

$inv1 : cfpS \subseteq cfp$
$inv2 : cfpR \subseteq cfpS$
$inv3 : proposeS \subseteq proposed$
$inv4 : proposeR \subseteq proposeS$
$inv5 : refuseS \subseteq refused$
$inv6 : refuseR \subseteq refuseS$
$inv7 : acceptS \subseteq accepted$
$inv8 : acceptR \subseteq acceptS$
$inv9 : rejectS \subseteq rejected$
$inv10 : rejectR \subseteq rejectS$
$inv11 : informS \subseteq informed$
$inv12 : informR \subseteq informS$
$inv13 : failS \subseteq failed$
$inv14 : failR \subseteq failS$
$inv15 : cancelS \subseteq cancelStarted$
$inv16 : cancelR \subseteq cancelS$
$inv17 : informCancelS \subseteq informCancelled$
$inv18 : informCancelR \subseteq informCancelS$
$inv19 : failCancelS \subseteq failCancelled$
$inv20 : failCancelR \subseteq failCancelS$
$inv21 : initiator \in dom(cfpS) \rightarrow AGENT$
$inv22 : participant \in dom(cfpS) \leftrightarrow AGENT$
$inv23 : proposeS \subseteq cfpR$
$inv24 : refuseS \subseteq cfpR$
$inv25 : acceptS \subseteq selected1$
$inv26 : rejectS \subseteq proposed$
$inv27 : informS \subseteq acceptR$

$inv28 : failS \subseteq acceptR$
$inv29 : informCancelS \subseteq cancelR$
$inv30 : failCancelS \subseteq cancelR$

EVENTS

INITIALISATION

BEGIN

$act1 : cfp := \varnothing$
$act6 : proposed := \varnothing$
$act7 : refused := \varnothing$
$act8 : accepted := \varnothing$
$act9 : rejected := \varnothing$
$act10 : informed := \varnothing$
$act11 : failed := \varnothing$
$act12 : cancelled := \varnothing$
$act13 : informCancelled := \varnothing$
$act14 : failCancelled := \varnothing$
$act2 : selected1 := \varnothing$
$act3 : cfpS := \varnothing$
$act4 : cfpR := \varnothing$
$act5 : proposeS := \varnothing$
$act15 : proposeR := \varnothing$
$act16 : refuseS := \varnothing$
$act17 : refuseR := \varnothing$
$act18 : acceptS := \varnothing$
$act19 : acceptR := \varnothing$
$act20 : rejectS := \varnothing$
$act21 : rejectR := \varnothing$
$act22 : informS := \varnothing$
$act23 : informR := \varnothing$
$act24 : failS := \varnothing$
$act25 : failR := \varnothing$
$act26 : cancelS := \varnothing$
$act27 : cancelR := \varnothing$
$act28 : informCancelS := \varnothing$
$act29 : informCancelR := \varnothing$
$act30 : failCancelS := \varnothing$
$act31 : failCancelR := \varnothing$
$act32 : completed := \varnothing$

$act33 : cancelStarted := \varnothing$

$act34 : initiator := \varnothing$

$act35 : participant := \varnothing$

END

EVENT sendCfp

REFINES callForProposals

ANY

c

as

a

WHERE

$grd1 : c \in CONVERSATION$

$grd2 : c \notin dom(cfp)$

$grd3 : as \in CONVERSATION \leftrightarrow AGENT$

$grd4 : dom(as) = \{c\}$

$grd6 : a \in AGENT$

$grd5 : ran(as) \subseteq AGENT \setminus \{a\}$

THEN

$act1 : cfp := cfp \cup as$

$act2 : cfpS := cfpS \cup as$

$act3 : initiator := initiator \cup \{c \mapsto a\}$

$act4 : participant := participant \cup as$

END

EVENT receiveCfp

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpS$

$grd2 : c \mapsto a \notin cfpR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : cfpR := cfpR \cup \{c \mapsto a\}$

END

EVENT sendProposal

REFINES propose

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin proposed$

$grd3 : c \mapsto a \notin refused$

$grd4 : c \mapsto a \in participant$

THEN

$act1 : proposed := proposed \cup \{c \mapsto a\}$

$act2 : proposeS := proposeS \cup \{c \mapsto a\}$

END

EVENT receiveProposal

ANY

c

a

WHERE

$grd1 : c \mapsto a \in proposeS$

$grd2 : c \mapsto a \notin proposeR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : proposeR := proposeR \cup \{c \mapsto a\}$

END

EVENT sendRefusal

REFINES refuse

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin proposed$

$grd4 : c \mapsto a \in participant$

THEN

$act1 : refused := refused \cup \{c \mapsto a\}$

$act2 : refuseS := refuseS \cup \{c \mapsto a\}$

END

EVENT receiveRefusal

ANY

c

a

WHERE

$grd1 : c \mapsto a \in refuseS$

$grd2 : c \mapsto a \notin refuseR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : refuseR := refuseR \cup \{c \mapsto a\}$

END

EVENT select

REFINES select1

ANY

c

as

WHERE

$grd1 : c \in dom(proposeR)$

$grd2 : as \subseteq \{c\} \lhd proposeR$

$grd3 : c \notin dom(selected1)$

THEN

$act1 : selected1 := selected1 \cup as$

END

EVENT sendAccept

REFINES accept

ANY

c

as

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(accepted)$

$grd3 : as \subseteq \{c\} \lhd selected1$

$grd4 : as \cap rejected = \varnothing$

$grd5 : as \subseteq participant$

THEN

$act1 : accepted := accepted \cup as$

$act2 : acceptS := acceptS \cup as$

END

EVENT receiveAccept

ANY

c

a

WHERE

$grd1 : c \mapsto a \in acceptS$

$grd2 : c \mapsto a \notin acceptR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : acceptR := acceptR \cup \{c \mapsto a\}$

END

EVENT sendReject
REFINES reject

ANY

c

as

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(rejected)$

$grd3 : as = (\{c\} \lhd proposed) \setminus (\{c\} \lhd selected1)$

$grd4 : as \cap accepted = \varnothing$

$grd5 : as \subseteq participant$

THEN

$act1 : rejected := rejected \cup as$

$act2 : rejectS := rejectS \cup as$

END

EVENT receiveReject

ANY

c

a

WHERE

$grd1 : c \mapsto a \in rejectS$

$grd2 : c \mapsto a \notin rejectR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : rejectR := rejectR \cup \{c \mapsto a\}$

END

EVENT sendInform
REFINES inform

ANY

c

a

WHERE

$grd1 : c \mapsto a \in acceptR$

$grd2 : c \mapsto a \notin informed$

$grd3 : c \mapsto a \notin failed$

$grd4 : c \mapsto a \in participant$

THEN

$act1 : informed := informed \cup \{c \mapsto a\}$

$act2 : informS := informS \cup \{c \mapsto a\}$

END

EVENT receiveInform

ANY

c

a

WHERE

$grd1 : c \mapsto a \in informS$

$grd2 : c \mapsto a \notin informR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : informR := informR \cup \{c \mapsto a\}$

END

EVENT sendFail
REFINES fail

ANY

c

a

WHERE

$grd1 : c \mapsto a \in acceptR$

$grd2 : c \mapsto a \notin failed$

$grd3 : c \mapsto a \notin informed$

$grd4 : c \mapsto a \in participant$

THEN

$act1 : failed := failed \cup \{c \mapsto a\}$

$act2 : failS := failS \cup \{c \mapsto a\}$

END

EVENT receiveFail

ANY

c

a

WHERE

$grd1 : c \mapsto a \in failS$

$grd2 : c \mapsto a \notin failR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : failR := failR \cup \{c \mapsto a\}$

END

EVENT complete

REFINES complete

ANY

c

as

WHERE

$grd1 : c \in dom(informR \cup failR)$

$grd2 : as \subseteq \{c\} \lhd (informR \cup failR)$

$grd3 : c \notin dom(completed)$

THEN

$act1 : completed := completed \cup as$

END

EVENT sendCancel

REFINES cancel1

ANY

c

as

WHERE

$grd1 : c \in dom(cfp)$

$grd2 : as \subseteq \{c\} \lhd cfp$

$grd3 : c \notin dom(cancelStarted)$

$grd4 : as \subseteq participant$

THEN

$act1 : cancelStarted := cancelStarted \cup as$

$act2 : cancelS := cancelS \cup as$

END

EVENT receiveCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cancelS$

$grd2 : c \mapsto a \notin cancelR$

THEN

$act1 : cancelR := cancelR \cup \{c \mapsto a\}$

END

EVENT sendInformCancel
REFINES informCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin informCancelled$

$grd3 : c \mapsto a \notin failCancelled$

$grd4 : c \mapsto a \in participant$

THEN

$act1 : informCancelled := informCancelled \cup \{c \mapsto a\}$

$act2 : informCancelS := informCancelS \cup \{c \mapsto a\}$

END

EVENT receiveInformCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in informCancelS$

$grd2 : c \mapsto a \notin informCancelR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : informCancelR := informCancelR \cup \{c \mapsto a\}$

END

EVENT sendFailCancel
REFINES failCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin failCancelled$

$grd3 : c \mapsto a \notin informCancelled$

$grd4 : c \mapsto a \in participant$

THEN

$act1 : failCancelled := failCancelled \cup \{c \mapsto a\}$

$act2 : failCancelS := failCancelS \cup \{c \mapsto a\}$

END

EVENT receiveFailCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in failCancelS$

$grd2 : c \mapsto a \notin failCancelR$

$grd3 : c \mapsto a \in participant$

THEN

$act1 : failCancelR := failCancelR \cup \{c \mapsto a\}$

END

EVENT cancelled

REFINES cancelled

ANY

c

as

WHERE

$grd1 : c \in dom(informCancelR \cup failCancelR)$

$grd2 : as \subseteq \{c\} \lhd (informCancelR \cup failCancelR)$

$grd3 : c \notin dom(cancelled)$

THEN

$act1 : cancelled := cancelled \cup as$

END

END

B.5 Context 2

CONTEXT context2

REFINES context

SETS

$MESSAGE$

CONSTANTS

$sender$
$receiver$
$messageConversation$

AXIOMS

$axm1 : sender \in MESSAGE \rightarrow AGENT$
$axm2 : receiver \in MESSAGE \rightarrow AGENT$
$axm3 : messageConversation \in MESSAGE \rightarrow CONVERSATION$

END

B.6 m3 - Third Refinement

MACHINE m3

REFINES m2

SEES context2

VARIABLES

cfp
$proposed$
$refused$
$selected1$

accepted
rejected
informed
failed
cancelStarted
informCancelled
failCancelled
cfpS
cfpR
proposeS
proposeR
refuseS
refuseR
acceptS
acceptR
rejectS
rejectR
informS
informR
failS
failR
cancelS
cancelR
informCancelS
informCancelR
failCancelS
failCancelR
completed
cancelled
initiator
participant
cfpM
proposeM
refuseM
acceptM
rejectM
informM
failM
cancelM
informCancelM
failCancelM

$pParticipant$
$pInitiator$

INVARIANTS

$inv1 : cfpM \subseteq MESSAGE$
$inv2 : cfpS = (cfpM \lhd messageConversation)^{-1};\ receiver$
$inv3 : proposeM \subseteq MESSAGE$
$inv4 : proposeS = (proposeM \lhd messageConversation)^{-1};\ sender$
$inv5 : refuseM \subseteq MESSAGE$
$inv6 : refuseS = (refuseM \lhd messageConversation)^{-1};\ sender$
$inv7 : acceptM \subseteq MESSAGE$
$inv8 : acceptS = (acceptM \lhd messageConversation)^{-1};\ receiver$
$inv9 : rejectM \subseteq MESSAGE$
$inv10 : rejectS = (rejectM \lhd messageConversation)^{-1};\ receiver$
$inv11 : informM \subseteq MESSAGE$
$inv12 : informS = (informM \lhd messageConversation)^{-1};\ sender$
$inv13 : failM \subseteq MESSAGE$
$inv14 : failS = (failM \lhd messageConversation)^{-1};\ sender$
$inv15 : cancelM \subseteq MESSAGE$
$inv16 : cancelS = (cancelM \lhd messageConversation)^{-1};\ receiver$
$inv17 : informCancelM \subseteq MESSAGE$
$inv18 : informCancelS = (informCancelM \lhd messageConversation)^{-1};\ sender$
$inv19 : failCancelM \subseteq MESSAGE$
$inv20 : failCancelS = (failCancelM \lhd messageConversation)^{-1};\ sender$
$inv21 : pParticipant \subseteq participant$
$inv22 : pParticipant \in dom(cfpR) \cup dom(cancelR) \leftrightarrow AGENT$
$inv23 : participant = ((cfpM \cup cancelM) \lhd messageConversation)^{-1};\ receiver$
$inv24 : pInitiator \subseteq initiator$
$inv25 : pInitiator \in dom(cfpR) \cup dom(cancelR) \rightarrow AGENT$
$inv26 : initiator = ((cfpM \cup cancelM) \lhd messageConversation)^{-1};\ sender$

EVENTS

INITIALISATION

BEGIN

$act1 : cfp := \varnothing$
$act6 : proposed := \varnothing$
$act7 : refused := \varnothing$
$act8 : accepted := \varnothing$

$act9 : rejected := \varnothing$
$act10 : informed := \varnothing$
$act11 : failed := \varnothing$
$act12 : cancelled := \varnothing$
$act13 : informCancelled := \varnothing$
$act14 : failCancelled := \varnothing$
$act2 : selected1 := \varnothing$
$act3 : cfpS := \varnothing$
$act4 : cfpR := \varnothing$
$act5 : proposeS := \varnothing$
$act15 : proposeR := \varnothing$
$act16 : refuseS := \varnothing$
$act17 : refuseR := \varnothing$
$act18 : acceptS := \varnothing$
$act19 : acceptR := \varnothing$
$act20 : rejectS := \varnothing$
$act21 : rejectR := \varnothing$
$act22 : informS := \varnothing$
$act23 : informR := \varnothing$
$act24 : failS := \varnothing$
$act25 : failR := \varnothing$
$act26 : cancelS := \varnothing$
$act27 : cancelR := \varnothing$
$act28 : informCancelS := \varnothing$
$act29 : informCancelR := \varnothing$
$act30 : failCancelS := \varnothing$
$act31 : failCancelR := \varnothing$
$act32 : completed := \varnothing$
$act33 : cancelStarted := \varnothing$
$act34 : initiator := \varnothing$
$act35 : participant := \varnothing$
$act36 : cfpM := \varnothing$
$act37 : proposeM := \varnothing$
$act38 : refuseM := \varnothing$
$act39 : acceptM := \varnothing$
$act40 : rejectM := \varnothing$
$act41 : informM := \varnothing$
$act42 : failM := \varnothing$
$act43 : cancelM := \varnothing$
$act44 : informCancelM := \varnothing$
$act45 : failCancelM := \varnothing$

$act46 : pParticipant := \varnothing$

$act47 : pInitiator := \varnothing$

END

EVENT sendCfp

REFINES sendCfp

ANY

c

as

a

ms

WHERE

$grd1 : c \in CONVERSATION$

$grd2 : c \notin dom(cfp)$

$grd3 : as \in CONVERSATION \leftrightarrow AGENT$

$grd4 : dom(as) = \{c\}$

$grd6 : a \in AGENT$

$grd5 : ran(as) \subseteq AGENT \setminus \{a\}$

$grd7 : ms \subseteq MESSAGE$

$grd8 : sender[ms] = \{a\}$

$grd9 : receiver[ms] = ran(as)$

$grd10 : messageConversation[ms] = \{c\}$

THEN

$act1 : cfp := cfp \cup as$

$act2 : cfpS := cfpS \cup as$

$act3 : initiator := initiator \cup \{c \mapsto a\}$

$act4 : participant := participant \cup as$

$act5 : cfpM := cfpM \cup ms$

END

EVENT receiveCfp

REFINES receiveCfp

ANY

c

a

m

WHERE

$grd1 : m \in cfpM$

$grd2 : c \mapsto a \notin cfpR$

$grd4 : receiver(m) = a$

$grd5 : messageConversation(m) = c$

THEN

$act1 : cfpR := cfpR \cup \{c \mapsto a\}$

$act2 : pParticipant := pParticipant \cup \{c \mapsto a\}$

$act3 : pInitiator := pInitiator \cup \{c \mapsto sender(m)\}$

END

EVENT sendProposal
REFINES sendProposal

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin proposed$

$grd3 : c \mapsto a \notin refused$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

THEN

$act1 : proposed := proposed \cup \{c \mapsto a\}$

$act2 : proposeS := proposeS \cup \{c \mapsto a\}$

$act3 : proposeM := proposeM \cup \{m\}$

END

EVENT receiveProposal
REFINES receiveProposal

ANY

c

a

m

WHERE

$grd1 : m \in proposeM$

$grd2 : c \mapsto a \notin proposeR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

THEN

$act1 : proposeR := proposeR \cup \{c \mapsto a\}$

END

EVENT sendRefusal

REFINES sendRefusal

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin proposed$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

THEN

$act1 : refused := refused \cup \{c \mapsto a\}$

$act2 : refuseS := refuseS \cup \{c \mapsto a\}$

$act3 : refuseM := refuseM \cup \{m\}$

END

EVENT receiveRefusal

REFINES receiveRefusal

ANY

c

a

m

WHERE

$grd1 : m \in refuseM$

$grd2 : c \mapsto a \notin refuseR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

THEN

$act1 : refuseR := refuseR \cup \{c \mapsto a\}$

END

EVENT select
REFINES select
ANY
c
as
WHERE
$grd1 : c \in dom(proposeR)$
$grd2 : as \subseteq \{c\} \lhd proposeR$
$grd3 : c \notin dom(selected1)$
THEN
$act1 : selected1 := selected1 \cup as$
END

EVENT sendAccept
REFINES sendAccept
ANY
c
as
ms
WHERE
$grd1 : c \in dom(selected1)$
$grd2 : c \notin dom(accepted)$
$grd3 : as \subseteq \{c\} \lhd selected1$
$grd4 : as \neq \varnothing$
$grd5 : as \cap rejected = \varnothing$
$grd6 : as \subseteq participant$
$grd7 : ms \subseteq MESSAGE$
$grd8 : receiver[ms] = ran(as)$
$grd9 : messageConversation[ms] = \{c\}$
$grd10 : sender[ms] = \{initiator(c)\}$
THEN
$act1 : accepted := accepted \cup as$
$act2 : acceptS := acceptS \cup as$
$act3 : acceptM := acceptM \cup ms$
END

EVENT receiveAccept
REFINES receiveAccept
ANY
c
a

m

WHERE

$grd1 : m \in acceptM$

$grd2 : c \mapsto a \notin acceptR$

$grd3 : c \mapsto a \in pParticipant$

$grd4 : receiver(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto sender(m) \in pInitiator$

THEN

$act1 : acceptR := acceptR \cup \{c \mapsto a\}$

END

EVENT sendReject
REFINES sendReject

ANY

c

as

ms

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(rejected)$

$grd3 : as = (\{c\} \lhd proposed) \setminus (\{c\} \lhd selected1)$

$grd4 : as \cap accepted = \varnothing$

$grd5 : as \subseteq participant$

$grd6 : ms \subseteq MESSAGE$

$grd7 : receiver[ms] = ran(as)$

$grd8 : messageConversation[ms] = \{c\}$

$grd9 : sender[ms] = \{initiator(c)\}$

THEN

$act1 : rejected := rejected \cup as$

$act2 : rejectS := rejectS \cup as$

$act3 : rejectM := rejectM \cup ms$

END

EVENT receiveReject
REFINES receiveReject

ANY

c

a

m

WHERE

$grd1 : m \in rejectM$

$grd2 : c \mapsto a \notin rejectR$

$grd3 : c \mapsto a \in pParticipant$

$grd4 : receiver(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto sender(m) \in pInitiator$

THEN

$act1 : rejectR := rejectR \cup \{c \mapsto a\}$

END

EVENT sendInform

REFINES sendInform

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in acceptR$

$grd2 : c \mapsto a \notin informed$

$grd3 : c \mapsto a \notin failed$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

THEN

$act1 : informed := informed \cup \{c \mapsto a\}$

$act2 : informS := informS \cup \{c \mapsto a\}$

$act3 : informM := informM \cup \{m\}$

END

EVENT receiveInform

REFINES receiveInform

ANY

c

a

m

WHERE

$grd1 : m \in informM$

$grd2 : c \mapsto a \notin informR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$
$grd5 : messageConversation(m) = c$
$grd6 : c \mapsto receiver(m) \in initiator$
THEN
$act1 : informR := informR \cup \{c \mapsto a\}$
END

EVENT sendFail
REFINES sendFail
ANY
c
a
m
WHERE
$grd1 : c \mapsto a \in acceptR$
$grd2 : c \mapsto a \notin failed$
$grd3 : c \mapsto a \notin informed$
$grd4 : c \mapsto a \in pParticipant$
$grd5 : m \in MESSAGE$
$grd6 : sender(m) = a$
$grd7 : messageConversation(m) = c$
$grd8 : c \mapsto receiver(m) \in pInitiator$
THEN
$act1 : failed := failed \cup \{c \mapsto a\}$
$act2 : failS := failS \cup \{c \mapsto a\}$
$act3 : failM := failM \cup \{m\}$
END

EVENT receiveFail
REFINES receiveFail
ANY
c
a
m
WHERE
$grd1 : m \in failM$
$grd2 : c \mapsto a \notin failR$
$grd3 : c \mapsto a \in participant$
$grd4 : sender(m) = a$
$grd5 : messageConversation(m) = c$
$grd6 : c \mapsto receiver(m) \in initiator$

THEN
$act1 : failR := failR \cup \{c \mapsto a\}$
END

EVENT complete
REFINES complete
ANY
c
as
WHERE
$grd1 : c \in dom(informR \cup failR)$
$grd2 : as \subseteq \{c\} \lhd (informR \cup failR)$
$grd3 : c \notin dom(completed)$
THEN
$act1 : completed := completed \cup as$
END

EVENT sendCancel
REFINES sendCancel
ANY
c
as
ms
WHERE
$grd1 : c \in dom(cfpS)$
$grd2 : as \subseteq \{c\} \lhd cfp$
$grd3 : c \notin dom(cancelStarted)$
$grd4 : as \subseteq participant$
$grd5 : ms \subseteq MESSAGE$
$grd6 : receiver[ms] = ran(as)$
$grd7 : messageConversation[ms] = \{c\}$
$grd8 : sender[ms] = \{initiator(c)\}$
THEN
$act1 : cancelStarted := cancelStarted \cup as$
$act2 : cancelS := cancelS \cup as$
$act3 : cancelM := cancelM \cup ms$
END

EVENT receiveCancel
REFINES receiveCancel
ANY

c

a

m

WHERE

$grd1 : m \in cancelM$

$grd2 : c \mapsto a \notin cancelR$

$grd4 : receiver(m) = a$

$grd5 : messageConversation(m) = c$

THEN

$act1 : cancelR := cancelR \cup \{c \mapsto a\}$

$act2 : pParticipant := pParticipant \cup \{c \mapsto a\}$

$act3 : pInitiator := pInitiator \cup \{c \mapsto sender(m)\}$

END

EVENT sendInformCancel

REFINES sendInformCancel

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin informCancelled$

$grd3 : c \mapsto a \notin failCancelled$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

THEN

$act1 : informCancelled := informCancelled \cup \{c \mapsto a\}$

$act2 : informCancelS := informCancelS \cup \{c \mapsto a\}$

$act3 : informCancelM := informCancelM \cup \{m\}$

END

EVENT receiveInformCancel

REFINES receiveInformCancel

ANY

c

a

m

WHERE

$grd1 : m \in informCancelM$

$grd2 : c \mapsto a \notin informCancelR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

THEN

$act1 : informCancelR := informCancelR \cup \{c \mapsto a\}$

END

EVENT sendFailCancel

REFINES sendFailCancel

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin failCancelled$

$grd3 : c \mapsto a \notin informCancelled$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

THEN

$act1 : failCancelled := failCancelled \cup \{c \mapsto a\}$

$act2 : failCancelS := failCancelS \cup \{c \mapsto a\}$

$act3 : failCancelM := failCancelM \cup \{m\}$

END

EVENT receiveFailCancel

REFINES receiveFailCancel

ANY

c

a

m

WHERE

$grd1 : m \in failCancelM$

$grd2 : c \mapsto a \notin failCancelR$

$grd3 : c \mapsto a \in participant$
$grd4 : sender(m) = a$
$grd5 : messageConversation(m) = c$
$grd6 : c \mapsto receiver(m) \in initiator$

THEN

$act1 : failCancelR := failCancelR \cup \{c \mapsto a\}$

END

EVENT cancelled
REFINES cancelled

ANY

c
as

WHERE

$grd1 : c \in dom(informCancelR \cup failCancelR)$
$grd2 : as \subseteq \{c\} \lhd (informCancelR \cup failCancelR)$
$grd3 : c \notin dom(cancelled)$

THEN

$act1 : cancelled := cancelled \cup as$

END

END

B.7 Context 3

CONTEXT context3

REFINES context2

CONSTANTS

CFP
PROPOSE
REFUSE
ACCEPT
REJECT
INFORM
FAIL

$CANCEL$
$INFORMCANCEL$
$FAILCANCEL$

AXIOMS

$axm1 : CFP \subseteq MESSAGE$
$axm2 : PROPOSE \subseteq MESSAGE$
$axm3 : REFUSE \subseteq MESSAGE$
$axm4 : ACCEPT \subseteq MESSAGE$
$axm5 : REJECT \subseteq MESSAGE$
$axm6 : INFORM \subseteq MESSAGE$
$axm7 : FAIL \subseteq MESSAGE$
$axm8 : CANCEL \subseteq MESSAGE$
$axm9 : INFORMCANCEL \subseteq MESSAGE$
$axm10 : FAILCANCEL \subseteq MESSAGE$
$axm11 : CFP \cap PROPOSE = \varnothing$
$axm12 : CFP \cap REFUSE = \varnothing$
$axm13 : CFP \cap ACCEPT = \varnothing$
$axm14 : CFP \cap REJECT = \varnothing$
$axm15 : CFP \cap INFORM = \varnothing$
$axm16 : CFP \cap FAIL = \varnothing$
$axm17 : CFP \cap CANCEL = \varnothing$
$axm18 : CFP \cap INFORMCANCEL = \varnothing$
$axm19 : CFP \cap FAILCANCEL = \varnothing$
$axm20 : PROPOSE \cap REFUSE = \varnothing$
$axm21 : PROPOSE \cap ACCEPT = \varnothing$
$axm22 : PROPOSE \cap REJECT = \varnothing$
$axm23 : PROPOSE \cap INFORM = \varnothing$
$axm24 : PROPOSE \cap FAIL = \varnothing$
$axm25 : PROPOSE \cap CANCEL = \varnothing$
$axm26 : PROPOSE \cap INFORMCANCEL = \varnothing$
$axm27 : PROPOSE \cap FAILCANCEL = \varnothing$
$axm28 : REFUSE \cap ACCEPT = \varnothing$
$axm29 : REFUSE \cap REJECT = \varnothing$
$axm30 : REFUSE \cap INFORM = \varnothing$
$axm31 : REFUSE \cap FAIL = \varnothing$
$axm32 : REFUSE \cap CANCEL = \varnothing$
$axm33 : REFUSE \cap INFORMCANCEL = \varnothing$
$axm34 : REFUSE \cap FAILCANCEL = \varnothing$
$axm35 : ACCEPT \cap REJECT = \varnothing$

$axm36 : ACCEPT \cap INFORM = \varnothing$
$axm37 : ACCEPT \cap FAIL = \varnothing$
$axm38 : ACCEPT \cap CANCEL = \varnothing$
$axm39 : ACCEPT \cap INFORMCANCEL = \varnothing$
$axm40 : ACCEPT \cap FAILCANCEL = \varnothing$
$axm41 : REJECT \cap INFORM = \varnothing$
$axm42 : REJECT \cap FAIL = \varnothing$
$axm43 : REJECT \cap CANCEL = \varnothing$
$axm44 : REJECT \cap INFORMCANCEL = \varnothing$
$axm45 : REJECT \cap FAILCANCEL = \varnothing$
$axm46 : INFORM \cap FAIL = \varnothing$
$axm47 : INFORM \cap CANCEL = \varnothing$
$axm48 : INFORM \cap INFORMCANCEL = \varnothing$
$axm49 : INFORM \cap FAILCANCEL = \varnothing$
$axm50 : FAIL \cap CANCEL = \varnothing$
$axm51 : FAIL \cap INFORMCANCEL = \varnothing$
$axm52 : FAIL \cap FAILCANCEL = \varnothing$
$axm53 : CANCEL \cap INFORMCANCEL = \varnothing$
$axm54 : CANCEL \cap FAILCANCEL = \varnothing$
$axm55 : INFORMCANCEL \cap FAILCANCEL = \varnothing$

END

B.8 m4 - Fourth Refinement

MACHINE m4

REFINES m3

SEES context3

VARIABLES

cfp
$proposed$
$refused$
$selected1$
$accepted$

$rejected$
$informed$
$failed$
$cancelStarted$
$informCancelled$
$failCancelled$
$cfpS$
$cfpR$
$proposeS$
$proposeR$
$refuseS$
$refuseR$
$acceptS$
$acceptR$
$rejectS$
$rejectR$
$informS$
$informR$
$failS$
$failR$
$cancelS$
$cancelR$
$informCancelS$
$informCancelR$
$failCancelS$
$failCancelR$
$completed$
$cancelled$
$initiator$
$participant$
$pParticipant$
$pInitiator$
$msgset$

INVARIANTS

$inv1 : msgset \subseteq MESSAGE$
$inv2 : cfpM = msgset \cap CFP$
$inv3 : proposeM = msgset \cap PROPOSE$
$inv4 : refuseM = msgset \cap REFUSE$
$inv5 : acceptM = msgset \cap ACCEPT$

$inv6 : rejectM = msgset \cap REJECT$
$inv7 : informM = msgset \cap INFORM$
$inv8 : failM = msgset \cap FAIL$
$inv9 : cancelM = msgset \cap CANCEL$
$inv10 : informCancelM = msgset \cap INFORMCANCEL$
$inv11 : failCancelM = msgset \cap FAILCANCEL$

EVENTS

INITIALISATION

BEGIN

$act1 : cfp := \varnothing$
$act6 : proposed := \varnothing$
$act7 : refused := \varnothing$
$act8 : accepted := \varnothing$
$act9 : rejected := \varnothing$
$act10 : informed := \varnothing$
$act11 : failed := \varnothing$
$act12 : cancelled := \varnothing$
$act13 : informCancelled := \varnothing$
$act14 : failCancelled := \varnothing$
$act2 : selected1 := \varnothing$
$act3 : cfpS := \varnothing$
$act4 : cfpR := \varnothing$
$act5 : proposeS := \varnothing$
$act15 : proposeR := \varnothing$
$act16 : refuseS := \varnothing$
$act17 : refuseR := \varnothing$
$act18 : acceptS := \varnothing$
$act19 : acceptR := \varnothing$
$act20 : rejectS := \varnothing$
$act21 : rejectR := \varnothing$
$act22 : informS := \varnothing$
$act23 : informR := \varnothing$
$act24 : failS := \varnothing$
$act25 : failR := \varnothing$
$act26 : cancelS := \varnothing$
$act27 : cancelR := \varnothing$
$act28 : informCancelS := \varnothing$
$act29 : informCancelR := \varnothing$

$act30 : failCancelS := \varnothing$
$act31 : failCancelR := \varnothing$
$act32 : completed := \varnothing$
$act33 : cancelStarted := \varnothing$
$act34 : initiator := \varnothing$
$act35 : participant := \varnothing$
$act46 : pParticipant := \varnothing$
$act47 : pInitiator := \varnothing$
$act36 : msgset := \varnothing$
END

EVENT sendCfp
REFINES sendCfp
ANY
c
as
a
ms
WHERE
$grd1 : c \in CONVERSATION$
$grd2 : c \notin dom(cfp)$
$grd3 : as \in CONVERSATION \leftrightarrow AGENT$
$grd4 : dom(as) = \{c\}$
$grd6 : a \in AGENT$
$grd5 : ran(as) \subseteq AGENT \setminus \{a\}$
$grd7 : ms \subseteq MESSAGE$
$grd8 : sender[ms] = \{a\}$
$grd9 : receiver[ms] = ran(as)$
$grd10 : messageConversation[ms] = \{c\}$
$grd11 : ms \subseteq CFP$
THEN
$act1 : cfp := cfp \cup as$
$act2 : cfpS := cfpS \cup as$
$act3 : initiator := initiator \cup \{c \mapsto a\}$
$act4 : participant := participant \cup as$
$act5 : msgset := msgset \cup ms$
END

EVENT receiveCfp
REFINES receiveCfp
ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin cfpR$
$grd4 : receiver(m) = a$
$grd5 : messageConversation(m) = c$
$grd3 : m \in CFP$

THEN

$act1 : cfpR := cfpR \cup \{c \mapsto a\}$
$act2 : pParticipant := pParticipant \cup \{c \mapsto a\}$
$act3 : pInitiator := pInitiator \cup \{c \mapsto sender(m)\}$

END

EVENT sendProposal
REFINES sendProposal

ANY

c
a
m

WHERE

$grd1 : c \mapsto a \in cfpR$
$grd2 : c \mapsto a \notin proposed$
$grd3 : c \mapsto a \notin refused$
$grd4 : c \mapsto a \in pParticipant$
$grd5 : m \in MESSAGE$
$grd6 : sender(m) = a$
$grd7 : messageConversation(m) = c$
$grd8 : c \mapsto receiver(m) \in pInitiator$
$grd9 : m \in PROPOSE$

THEN

$act1 : proposed := proposed \cup \{c \mapsto a\}$
$act2 : proposeS := proposeS \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveProposal
REFINES receiveProposal

ANY

c

a

m

WHERE

$grd1 : m \in msgset$

$grd2 : c \mapsto a \notin proposeR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

$grd7 : m \in PROPOSE$

THEN

$act1 : proposeR := proposeR \cup \{c \mapsto a\}$

END

EVENT sendRefusal

REFINES sendRefusal

ANY

c

a

m

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin refused$

$grd3 : c \mapsto a \notin proposed$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

$grd9 : m \in REFUSE$

THEN

$act1 : refused := refused \cup \{c \mapsto a\}$

$act2 : refuseS := refuseS \cup \{c \mapsto a\}$

$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveRefusal

REFINES receiveRefusal

ANY

c

a

m

WHERE

$grd1 : m \in msgset$

$grd2 : c \mapsto a \notin refuseR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

$grd7 : m \in REFUSE$

THEN

$act1 : refuseR := refuseR \cup \{c \mapsto a\}$

END

EVENT select
REFINES select

ANY

c

as

WHERE

$grd1 : c \in dom(proposeR)$

$grd2 : as \subseteq \{c\} \lhd proposeR$

$grd3 : c \notin dom(selected1)$

THEN

$act1 : selected1 := selected1 \cup as$

END

EVENT sendAccept
REFINES sendAccept

ANY

c

as

ms

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(accepted)$

$grd3 : as = \{c\} \lhd selected1$

$grd4 : as \neq \varnothing$

$grd5 : as \cap rejected = \varnothing$

$grd6 : as \subseteq participant$

$grd7 : ms \subseteq MESSAGE$

$grd8 : receiver[ms] = ran(as)$

$grd9 : messageConversation[ms] = \{c\}$

$grd10 : sender[ms] = \{initiator(c)\}$

$grd11 : ms \subseteq ACCEPT$

THEN

$act1 : accepted := accepted \cup as$

$act2 : acceptS := acceptS \cup as$

$act3 : msgset := msgset \cup ms$

END

EVENT receiveAccept

REFINES receiveAccept

ANY

c

a

m

WHERE

$grd1 : m \in msgset$

$grd2 : c \mapsto a \notin acceptR$

$grd3 : c \mapsto a \in pParticipant$

$grd4 : receiver(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto sender(m) \in pInitiator$

$grd7 : m \in ACCEPT$

THEN

$act1 : acceptR := acceptR \cup \{c \mapsto a\}$

END

EVENT sendReject

REFINES sendReject

ANY

c

as

ms

WHERE

$grd1 : c \in dom(selected1)$

$grd2 : c \notin dom(rejected)$

$grd3 : as = (\{c\} \lhd proposed) \setminus (\{c\} \lhd selected1)$

$grd4 : as \cap accepted = \varnothing$

$grd5 : as \subseteq participant$

$grd6 : ms \subseteq MESSAGE$

$grd7 : receiver[ms] = ran(as)$

$grd8 : messageConversation[ms] = \{c\}$
$grd9 : sender[ms] = \{initiator(c)\}$
$grd10 : ms \subseteq REJECT$

THEN

$act1 : rejected := rejected \cup as$
$act2 : rejectS := rejectS \cup as$
$act3 : msgset := msgset \cup ms$

END

EVENT receiveReject
REFINES receiveReject

ANY

c
a
m

WHERE

$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin rejectR$
$grd3 : c \mapsto a \in pParticipant$
$grd4 : receiver(m) = a$
$grd5 : messageConversation(m) = c$
$grd6 : c \mapsto sender(m) \in pInitiator$
$grd7 : m \in REJECT$

THEN

$act1 : rejectR := rejectR \cup \{c \mapsto a\}$

END

EVENT sendInform
REFINES sendInform

ANY

c
a
m

WHERE

$grd1 : c \mapsto a \in acceptR$
$grd2 : c \mapsto a \notin informed$
$grd3 : c \mapsto a \notin failed$
$grd4 : c \mapsto a \in pParticipant$
$grd5 : m \in MESSAGE$
$grd6 : sender(m) = a$
$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$
$grd9 : m \in INFORM$
THEN
$act1 : informed := informed \cup \{c \mapsto a\}$
$act2 : informS := informS \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$
END

EVENT receiveInform
REFINES receiveInform
ANY
c
a
m
WHERE
$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin informR$
$grd3 : c \mapsto a \in participant$
$grd4 : sender(m) = a$
$grd5 : messageConversation(m) = c$
$grd6 : c \mapsto receiver(m) \in initiator$
$grd7 : m \in INFORM$
THEN
$act1 : informR := informR \cup \{c \mapsto a\}$
END

EVENT sendFail
REFINES sendFail
ANY
c
a
m
WHERE
$grd1 : c \mapsto a \in acceptR$
$grd2 : c \mapsto a \notin failed$
$grd3 : c \mapsto a \notin informed$
$grd4 : c \mapsto a \in pParticipant$
$grd5 : m \in MESSAGE$
$grd6 : sender(m) = a$
$grd7 : messageConversation(m) = c$
$grd8 : c \mapsto receiver(m) \in pInitiator$

$grd9 : m \in FAIL$

THEN

$act1 : failed := failed \cup \{c \mapsto a\}$

$act2 : failS := failS \cup \{c \mapsto a\}$

$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveFail
REFINES receiveFail

ANY

c

a

m

WHERE

$grd1 : m \in msgset$

$grd2 : c \mapsto a \notin failR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

$grd7 : m \in FAIL$

THEN

$act1 : failR := failR \cup \{c \mapsto a\}$

END

EVENT complete
REFINES complete

ANY

c

as

WHERE

$grd1 : c \in dom(informR \cup failR)$

$grd2 : as \subseteq \{c\} \lhd (informR \cup failR)$

$grd3 : c \notin dom(completed)$

THEN

$act1 : completed := completed \cup as$

END

EVENT sendCancel
REFINES sendCancel

ANY

c
as
ms
WHERE
$grd1 : c \in dom(cfpS)$
$grd2 : as \subseteq \{c\} \lhd cfpS$
$grd3 : c \notin dom(cancelStarted)$
$grd4 : as \subseteq participant$
$grd5 : ms \subseteq MESSAGE$
$grd6 : receiver[ms] = ran(as)$
$grd7 : messageConversation[ms] = \{c\}$
$grd8 : sender[ms] = \{initiator(c)\}$
$grd9 : ms \subseteq CANCEL$
THEN
$act1 : cancelStarted := cancelStarted \cup as$
$act2 : cancelS := cancelS \cup as$
$act3 : msgset := msgset \cup ms$
END

EVENT receiveCancel
REFINES receiveCancel
ANY
c
a
m
WHERE
$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin cancelR$
$grd4 : receiver(m) = a$
$grd5 : messageConversation(m) = c$
$grd3 : m \in CANCEL$
THEN
$act1 : cancelR := cancelR \cup \{c \mapsto a\}$
$act2 : pParticipant := pParticipant \cup \{c \mapsto a\}$
$act3 : pInitiator := pInitiator \cup \{c \mapsto sender(m)\}$
END

EVENT sendInformCancel
REFINES sendInformCancel
ANY
c

a
m
WHERE
$grd1 : c \mapsto a \in cancelR$
$grd2 : c \mapsto a \notin informCancelled$
$grd3 : c \mapsto a \notin failCancelled$
$grd4 : c \mapsto a \in pParticipant$
$grd5 : m \in MESSAGE$
$grd6 : sender(m) = a$
$grd7 : messageConversation(m) = c$
$grd8 : c \mapsto receiver(m) \in pInitiator$
$grd9 : m \in INFORMCANCEL$
THEN
$act1 : informCancelled := informCancelled \cup \{c \mapsto a\}$
$act2 : informCancelS := informCancelS \cup \{c \mapsto a\}$
$act3 : msgset := msgset \cup \{m\}$
END

EVENT receiveInformCancel
REFINES receiveInformCancel
ANY
c
a
m
WHERE
$grd1 : m \in msgset$
$grd2 : c \mapsto a \notin informCancelR$
$grd3 : c \mapsto a \in participant$
$grd4 : sender(m) = a$
$grd5 : messageConversation(m) = c$
$grd6 : c \mapsto receiver(m) \in initiator$
$grd7 : m \in INFORMCANCEL$
THEN
$act1 : informCancelR := informCancelR \cup \{c \mapsto a\}$
END

EVENT sendFailCancel
REFINES sendFailCancel
ANY
c
a

m

WHERE

$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin failCancelled$

$grd3 : c \mapsto a \notin informCancelled$

$grd4 : c \mapsto a \in pParticipant$

$grd5 : m \in MESSAGE$

$grd6 : sender(m) = a$

$grd7 : messageConversation(m) = c$

$grd8 : c \mapsto receiver(m) \in pInitiator$

$grd9 : m \in FAILCANCEL$

THEN

$act1 : failCancelled := failCancelled \cup \{c \mapsto a\}$

$act2 : failCancelS := failCancelS \cup \{c \mapsto a\}$

$act3 : msgset := msgset \cup \{m\}$

END

EVENT receiveFailCancel
REFINES receiveFailCancel

ANY

c

a

m

WHERE

$grd1 : m \in msgset$

$grd2 : c \mapsto a \notin failCancelR$

$grd3 : c \mapsto a \in participant$

$grd4 : sender(m) = a$

$grd5 : messageConversation(m) = c$

$grd6 : c \mapsto receiver(m) \in initiator$

$grd7 : m \in FAILCANCEL$

THEN

$act1 : failCancelR := failCancelR \cup \{c \mapsto a\}$

END

EVENT cancelled
REFINES cancelled

ANY

c

as

WHERE

$grd1 : c \in dom(informCancelR \cup failCancelR)$
$grd2 : as \subseteq \{c\} \lhd (informCancelR \cup failCancelR)$
$grd3 : c \notin dom(cancelled)$

THEN

$act1 : cancelled := cancelled \cup as$

END

END

B.9 m5 - Fifth Refinement

REFINEMENT m5

REFINES m4 **SEES** context3

INCLUDES mw.middleware0, initiator.initiator0, participant.participant0

INVARIANT

$cfp = initiator.cfp \wedge$
$cfpS = initiator.cfpS \wedge$
$cfpR = participant.cfpR \wedge$
$refused = participant.refused \wedge$
$refuseS = participant.refuseS \wedge$
$refuseR = initiator.refuseR \wedge$
$proposed = participant.proposed \wedge$
$proposeS = participant.proposeS \wedge$
$proposeR = initiator.proposeR \wedge$
$selected1 = initiator.selected \wedge$
$accepted = initiator.accepted \wedge$
$acceptS = initiator.acceptS \wedge$
$acceptR = participant.acceptR \wedge$
$rejected = initiator.rejected \wedge$
$rejectS = initiator.rejectS \wedge$
$rejectR = participant.rejectR \wedge$
$informed = participant.informed \wedge$
$informS = participant.informS \wedge$
$informR = initiator.informR \wedge$
$failed = participant.failed \wedge$

$failS = participant.failS \wedge$
$failR = initiator.failR \wedge$
$cancelStarted = initiator.cancelStarted \wedge$
$cancelS = initiator.cancelS \wedge$
$cancelR = participant.cancelR \wedge$
$informCancelled = participant.informCancelled \wedge$
$informCancelS = participant.informCancelS \wedge$
$informCancelR = initiator.informCancelR \wedge$
$failCancelled = participant.failCancelled \wedge$
$failCancelS = participant.failCancelS \wedge$
$failCancelR = initiator.failCancelR \wedge$
$completed = initiator.completed \wedge$
$cancelled = initiator.cancelled \wedge$
$msgset = mw.msgset \wedge$
$initiator = initiator.initiator \wedge$
$participant = initiator.participant \wedge$
$pInitiator = participant.pInitiator \wedge$
$pParticipant = participant.pParticipant$

EVENTS

EVENT sendCfp =
VAR
m
IN
$m \leftarrow initiator.sendCfp;$
$mw.bcast(m)$
END;

EVENT receiveCfp =
ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$participant.receiveCfp(m)$

```
    END
  END;


EVENT sendRefusal =
  VAR
    m
  IN
    m ← participant.sendRefusal;
    mw.send(m)
  END;


EVENT sendProposal =
  VAR
    m
  IN
    m ← participant.sendProposal;
    mw.send(m)
  END;


EVENT receiveProposal =
  ANY
    a
  WHERE
    a ∈ AGENT
  THEN
    VAR
      m
    IN
      m ← mw.receive(a);
      initiator.receiveProposal(m)
    END
  END;


EVENT receiveRefusal =
  ANY
    a
  WHERE
```

$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$initiator.receiveRefusal(m)$
END
END;

EVENT select = initiator.select;

EVENT sendAccept =
VAR
m
IN
$m \leftarrow initiator.sendAccept;$
$mw.bcast(m)$
END;

EVENT sendReject =
VAR
m
IN
$m \leftarrow initiator.sendReject;$
$mw.bcast(m)$
END;

EVENT receiveAccept =
ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN

```
        m ← mw.receive(a);
        participant.receiveAccept(m)
      END
    END;


EVENT receiveReject =
    ANY
      a
    WHERE
      a ∈ AGENT
    THEN
      VAR
        m
      IN
        m ← mw.receive(a);
        participant.receiveReject(m)
      END
    END;


EVENT sendInform =
    VAR
      m
    IN
      m ← participant.sendInform;
      mw.send(m)
    END;


EVENT sendFail =
    VAR
      m
    IN
      m ← participant.sendFail;
      mw.send(m)
    END;


EVENT receiveInform =
    ANY
```

a
WHERE
$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$initiator.receiveInform(m)$
END
END;

EVENT receiveFail =
ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$initiator.receiveFail(m)$
END
END;

EVENT complete = initiator.complete;

EVENT sendCancel =
VAR
m
IN
$m \leftarrow initiator.sendCancel;$
$mw.bcast(m)$
END;

EVENT receiveCancel =

ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$participant.receiveCancel(m)$
END
END;

EVENT sendInformCancel =
VAR
m
IN
$m \leftarrow participant.sendInformCancel;$
$mw.send(m)$
END;

EVENT sendFailCancel =
VAR
m
IN
$m \leftarrow participant.sendFailCancel;$
$mw.send(m)$
END;

EVENT receiveInformCancel =
ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN

$m \leftarrow mw.receive(a);$
$initiator.receiveInformCancel(m)$
END
END;

EVENT receiveFailCancel =
ANY
a
WHERE
$a \in AGENT$
THEN
VAR
m
IN
$m \leftarrow mw.receive(a);$
$initiator.receiveFailCancel(m)$
END
END;

EVENT cancelled = initiator.cancelled

END

B.10 Initiator - Component Model

MODEL initiator0

SEES context

VARIABLES
cfp, cfpS, refuseR, proposeR, selected, accepted, acceptS, rejectS, rejected, informR, failR, completed, cancelStarted, cancelS, informCancelR, failCancelR, cancelled, initiator, participant

INVARIANT
$cfp \in CONVERSATION \leftrightarrow AGENT \wedge$
$cfpS \in CONVERSATION \leftrightarrow AGENT \wedge$
$refuseR \in CONVERSATION \leftrightarrow AGENT \wedge$
$proposeR \in CONVERSATION \leftrightarrow AGENT \wedge$

$selected \in CONVERSATION \leftrightarrow AGENT \wedge$
$accepted \in CONVERSATION \leftrightarrow AGENT \wedge$
$acceptS \in CONVERSATION \leftrightarrow AGENT \wedge$
$rejected \in CONVERSATION \leftrightarrow AGENT \wedge$
$rejectS \in CONVERSATION \leftrightarrow AGENT \wedge$
$informR \in CONVERSATION \leftrightarrow AGENT \wedge$
$failR \in CONVERSATION \leftrightarrow AGENT \wedge$
$cancelStarted \in CONVERSATION \leftrightarrow AGENT \wedge$
$cancelS \in CONVERSATION \leftrightarrow AGENT \wedge$
$informCancelR \in CONVERSATION \leftrightarrow AGENT \wedge$
$failCancelR \in CONVERSATION \leftrightarrow AGENT \wedge$
$cancelled \in CONVERSATION \leftrightarrow AGENT \wedge$
$initiator \in dom(cfpS) \rightarrow AGENT \wedge$
$participant \in dom(cfpS) \leftrightarrow AGENT$

INITIALISATION

$cfp := \varnothing \;||$
$cfpS := \varnothing \;||$
$refuseR := \varnothing \;||$
$proposeR := \varnothing \;||$
$selected := \varnothing \;||$
$accepted := \varnothing \;||$
$acceptS := \varnothing \;||$
$rejected := \varnothing \;||$
$rejectS := \varnothing \;||$
$informR := \varnothing \;||$
$failR := \varnothing \;||$
$completed := \varnothing \;||$
$cancelStarted := \varnothing \;||$
$cancelS := \varnothing \;||$
$informCancelR := \varnothing \;||$
$failCancelR := \varnothing \;||$
$cancelled := \varnothing \;||$
$initiator := \varnothing \;||$
$participant := \varnothing$

EVENTS

EVENT ms1 ← sendCfp =

ANY

c

as

a

ms

WHERE

$c \in CONVERSATION \land$

$c \notin dom(cfp) \land$

$as \in CONVERSATION \leftrightarrow AGENT \land$

$dom(as) = \{c\} \land$

$a \in AGENT \land$

$ran(as) \subseteq AGENT \setminus a \land$

$ms \subseteq MESSAGE \land$

$sender[ms] = a \land$

$receiver[ms] = ran(as) \land$

$messageConversation[ms] = \{c\} \land$

$ms \subseteq CFP$

THEN

$cfp := cfp \cup as \mid\mid$

$cfpS := cfpS \cup as \mid\mid$

$initiator := initiator \cup \{c \mapsto a\} \mid\mid$

$participant := participant \cup as \mid\mid$

$ms1 := ms$

END;

EVENT receiveRefusal(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin refuseR \land$

$c \mapsto a \in participant \land$

$sender(m) = a \land$

$messageConversation(m) = c \land$

$c \mapsto receiver(m) \in initiator \land$

$m \in REFUSE$

THEN

$refuseR := refuseR \cup \{c \mapsto a\}$

END

END;

EVENT receiveProposal(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin proposeR \wedge$

$c \mapsto a \in participant \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$c \mapsto receiver(m) \in initiator \wedge$

$m \in PROPOSE$

THEN

$proposeR := proposeR \cup \{c \mapsto a\}$

END

END;

EVENT select =

ANY

c

as

WHERE

$c \in dom(proposeR) \wedge$

$as \subseteq \{c\} \lhd proposeR \wedge$

$c \notin dom(selected)$

THEN

$selected := selected \cup as$

END;

EVENT ms1 ← sendAccept =

ANY

c

as

ms

WHERE

$c \in dom(selected) \wedge$

$c \notin dom(accepted) \wedge$

$as \subseteq \{c\} \lhd selected \wedge$

$as \neq \varnothing \wedge$

$as \cap rejected = \varnothing \wedge$

$as \subseteq participant \wedge$

$ms \subseteq MESSAGE \wedge$

$receiver[ms] = ran(as) \wedge$

$messageConversation[ms] = \{c\} \wedge$

$sender[ms] = initiator(c) \wedge$

$ms \subseteq ACCEPT$

THEN

$accepted := accepted \cup as \mid\mid$

$acceptS := acceptS \cup as \mid\mid$

$ms1 := ms$

END;

EVENT ms1 $\leftarrow$ sendReject =

ANY

c

as

ms

WHERE

$c \in dom(selected) \wedge$

$c \notin dom(rejected) \wedge$

$as = (\{c\} \lhd proposed) \setminus (\{c\} \lhd selected) \wedge$

$as \cap accepted = \varnothing \wedge$

$as \subseteq participant \wedge$

$ms \subseteq MESSAGE \wedge$

$receiver[ms] = ran(as) \wedge$

$messageConversation[ms] = \{c\} \wedge$

$sender[ms] = initiator(c) \wedge$

$ms \subseteq REJECT$

THEN

$rejected := rejected \cup as \mid\mid$

$rejectS := rejectS \cup as \mid\mid$

$ms1 := ms$

END;

EVENT receiveInform(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin informR \wedge$

$c \mapsto a \in participant \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$c \mapsto receiver(m) \in initiator \wedge$

$m \in INFORM$

THEN

$informR := informR \cup \{c \mapsto a\}$

END

END;

EVENT receiveFail(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin failR \wedge$

$c \mapsto a \in participant \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$c \mapsto receiver(m) \in initiator \wedge$

$m \in FAIL$

THEN

$failR := failR \cup \{c \mapsto a\}$

END

END;

EVENT complete =

ANY

c

as

WHERE

$c \in dom(informR \cup failR) \wedge$

$as \subseteq \{c\} \lhd (informR \cup failR) \wedge$

$c \notin dom(completed)$

THEN

$completed := completed \cup as$

END;

EVENT ms1 ← sendCancel =

ANY

c

as

ms

WHERE

$c \in dom(cfpS) \wedge$

$as = \{c\} \lhd cfpS \wedge$

$c \notin dom(cancelStarted) \wedge$

$as \subseteq participant \wedge$

$ms \subseteq MESSAGE \wedge$

$receiver[ms] = ran(as) \wedge$

$messageConversation[ms] = \{c\} \wedge$

$sender[ms] = intiator(c) \wedge$

$ms \subseteq CANCEL$

THEN

$cancelStarted := cancelStarted \cup as \;||$

$cancelS := cancelS \cup as \;||$

$ms1 := ms$

END;

EVENT receiveInformCancel(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin informCancelR \wedge$

$c \mapsto a \in participant \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$c \mapsto receiver(m) \in initiator \wedge$

$m \in INFORMCANCEL$

THEN

$informCancelR := informCancelR \cup \{c \mapsto a\}$

END

END;

EVENT receiveFailCancel(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin failCancelR \wedge$

$c \mapsto a \in participant \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$c \mapsto receiver(m) \in initiator \wedge$

$m \in FAILCANCEL$

THEN

$failCancelR := failCancelR \cup \{c \mapsto a\}$

END

END;

EVENT cancelled =

ANY

c

as

WHERE

$c \in dom(informCancelR \cup failCancelR) \wedge$

$as \subseteq \{c\} \lhd (informCancelR \cup failCancelR) \wedge$
$c \notin dom(cancelled)$
THEN
$cancelled := cancelled \cup as$
END;

END

B.11 Participant - Component Model

MODEL participant0

SEES context

VARIABLES
cfpR, refuseS, proposeS, acceptR, rejectR, informS, failS, cancelR, informCancelS, failCancelS, pInitiator, pParticipant

INVARIANT

$cfpR \in CONVERSATION \leftrightarrow AGENT \wedge$
$proposed \in CONVERSATION \leftrightarrow AGENT \wedge$
$refused \in CONVERSATION \leftrightarrow AGENT \wedge$
$refuseS \in CONVERSATION \leftrightarrow AGENT \wedge$
$proposeS \in CONVERSATION \leftrightarrow AGENT \wedge$
$acceptR \in CONVERSATION \leftrightarrow AGENT \wedge$
$rejectR \in CONVERSATION \leftrightarrow AGENT \wedge$
$informed \in CONVERSATION \leftrightarrow AGENT \wedge$
$failed \in CONVERSATION \leftrightarrow AGENT \wedge$
$informS \in CONVERSATION \leftrightarrow AGENT \wedge$
$failS \in CONVERSATION \leftrightarrow AGENT \wedge$
$cancelR \in CONVERSATION \leftrightarrow AGENT \wedge$
$informCancelS \in CONVERSATION \leftrightarrow AGENT \wedge$
$failCancelS \in CONVERSATION \leftrightarrow AGENT \wedge$
$pInitiator \in (dom(cfpR) \cup dom(cancelR)) \rightarrow AGENT \wedge$
$pParticipant \in (dom(cfpR) \cup dom(cancelR)) \leftrightarrow AGENT$

INITIALISATION
$cfpR := \varnothing \;||$
$proposed := \varnothing \;||$

$refused := \varnothing \;||$
$refuseS := \varnothing \;||$
$proposeS := \varnothing \;||$
$acceptR := \varnothing \;||$
$rejectR := \varnothing \;||$
$informed := \varnothing \;||$
$failed := \varnothing \;||$
$informS := \varnothing \;||$
$failS := \varnothing \;||$
$cancelR := \varnothing \;||$
$informCancelS := \varnothing \;||$
$failCancelS := \varnothing \;||$
$pInitiator := \varnothing \;||$
$pParticipant := \varnothing$

EVENTS

EVENT receiveCfp(m) =
PRE
$m \in MESSAGE$
THEN
ANY
c
a
WHERE
$c \mapsto a \notin cfpR \wedge$
$receiver(m) = a \wedge$
$messageConversation(m) = c \wedge$
$m \in CFP$
THEN
$cfpR := cfpR \cup \{c \mapsto a\} \;||$
$pInitiator := pInitiator \cup \{c \mapsto sender(m)\} \;||$
$pParticipant := pParticipant \cup \{c \mapsto a\}$
END
END;

EVENT m1 ← sendRefusal =
ANY
c
a

m

WHERE

$c \mapsto a \in cfpR \wedge$
$c \mapsto a \notin refused \wedge$
$c \mapsto a \notin proposed \wedge$
$c \mapsto a \in pParticipant \wedge$
$m \in MESSAGE \wedge$
$sender(m) = a \wedge$
$messageConversation(m) = c \wedge$
$c \mapsto receiver(m) \in pInitiator \wedge$
$m \in REFUSE$

THEN

$refused := refused \cup \{c \mapsto a\} \ ||$
$refuseS := refuseS \cup \{c \mapsto a\} \ ||$
$m1 := m$

END;

EVENT m1 $\leftarrow$ sendProposal =

ANY

c
a
m

WHERE

$c \mapsto a \in cfpR \wedge$
$c \mapsto a \notin proposed \wedge$
$c \mapsto a \notin refused \wedge$
$c \mapsto a \in pParticipant \wedge$
$m \in MESSAGE \wedge$
$sender(m) = a \wedge$
$messageConversation(m) = c \wedge$
$c \mapsto receiver(m) \in pInitiator \wedge$
$m \in PROPOSE$

THEN

$proposed := proposed \cup \{c \mapsto a\} \ ||$
$proposeS := proposeS \cup \{c \mapsto a\} \ ||$
$m1 := m$

END;

EVENT receiveReject(m) =

PRE
$m \in MESSAGE$
THEN
ANY
c
a
WHERE
$c \mapsto a \notin rejectR \wedge$
$c \mapsto a \in pParticipant \wedge$
$receiver(m) = a \wedge$
$messageConversation(m) = c \wedge$
$c \mapsto sender(m) \in pInitiator \wedge$
$m \in REJECT$
THEN
$rejectR := rejectR \cup \{c \mapsto a\}$
END
END;

EVENT receiveAccept(m) =
PRE
$m \in MESSAGE$
THEN
ANY
c
a
WHERE
$c \mapsto a \notin acceptR \wedge$
$c \mapsto a \in pParticipant \wedge$
$receiver(m) = a \wedge$
$messageConversation(m) = c \wedge$
$c \mapsto sender(m) \in pInitiator \wedge$
$m \in ACCEPT$
THEN
$acceptR := acceptR \cup \{c \mapsto a\}$
END
END;

EVENT m1 $\leftarrow$ sendInform =
ANY

c
a
m
WHERE
$c \mapsto a \in acceptR \wedge$
$c \mapsto a \notin informed \wedge$
$c \mapsto a \notin failed \wedge$
$c \mapsto a \in pParticipant \wedge$
$m \in MESSAGE \wedge$
$sender(m) = a \wedge$
$messageConversation(m) = c \wedge$
$c \mapsto receiver(m) \in pInitiator \wedge$
$m \in INFORM$
THEN
$informed := informed \cup \{c \mapsto a\} \parallel$
$informS := informS \cup \{c \mapsto a\} \parallel$
$m1 := m$
END;

EVENT m1 $\leftarrow$ sendFail =
ANY
c
a
m
WHERE
$c \mapsto a \in acceptR \wedge$
$c \mapsto a \notin failed \wedge$
$c \mapsto a \notin informed \wedge$
$c \mapsto a \in pParticipant \wedge$
$m \in MESSAGE \wedge$
$sender(m) = a \wedge$
$messageConversation(m) = c \wedge$
$c \mapsto receiver(m) \in pInitiator \wedge$
$m \in FAIL$
THEN
$failed := failed \cup \{c \mapsto a\} \parallel$
$failS := failS \cup \{c \mapsto a\} \parallel$
$m1 := m$
END;

EVENT receiveCancel(m) =

PRE

$m \in MESSAGE$

THEN

ANY

c

a

WHERE

$c \mapsto a \notin cancelR \wedge$

$c \mapsto a \in pParticipant \wedge$

$receiver(m) = a \wedge$

$messageConversation(m) = c \wedge$

$m \in CANCEL$

THEN

$cancelR := cancelR \cup \{c \mapsto a\} \mid\mid$

$pInitiator := pInitiator \cup \{c \mapsto sender(m)\} \mid\mid$

$pParticipant := pParticipant \cup \{c \mapsto a\}$

END

END;

EVENT m1 ← sendInformCancel =

ANY

c

a

m

WHERE

$c \mapsto a \in cancelR \wedge$

$c \mapsto a \notin informCancelled \wedge$

$c \mapsto a \notin failCancelled \wedge$

$c \mapsto a \in pParticipant \wedge$

$m \in MESSAGE \wedge$

$sender(m) = a \wedge$

$messageConversation(m) = c \wedge$

$c \mapsto receiver(m) \in pInitiator \wedge$

$m \in INFORMCANCEL$

THEN

$informCancelled := informCancelled \cup \{c \mapsto a\} \mid\mid$

$informCancelS := informCancelS \cup \{c \mapsto a\} \mid\mid$

$m1 := m$

END;

EVENT m1 ← sendFailCancel =

ANY

c

a

m

WHERE

$c \mapsto a \in cancelR \land$

$c \mapsto a \notin failCancelled \land$

$c \mapsto a \notin informCancelled \land$

$c \mapsto a \in pParticipant \land$

$m \in MESSAGE \land$

$sender(m) = a \land$

$messageConversation(m) = c \land$

$c \mapsto receiver(m) \in pInitiator \land$

$m \in FAILCANCEL$

THEN

$failCancelled := failCancelled \cup \{c \mapsto a\} \;||$

$failCancelS := failCancelS \cup \{c \mapsto a\} \;||$

$m1 := m$

END

END

B.12 Middleware - Component Model

MODEL middleware0

SEES context

VARIABLES msgset

INVARIANT

$msgset \subseteq MESSAGE$

INITIALISATION

$msgset := \varnothing$

EVENTS

EVENT send(m) =

PRE

$m \in MESSAGE$

THEN

$msgset := msgset \cup m$

END;

EVENT bcast(ms) =

PRE

$ms \subseteq MESSAGE$

THEN

$msgset := msgset \cup ms$

END;

EVENT mr ← receive(a) =

PRE

$a \in AGENT$

THEN

ANY

m

WHERE

$m \in msgset \wedge$

$receiver(m) = a$

THEN

$mr := m$

END

END

END

B.13 Example Proof Obligation

This section provides an example of a proof obligation being discharged using the interactive proof environment of the RODIN platform. The proof obligation comes from the third refinement model for the Contract Net case study. In the figures included below the naming of the models used is different from those above. This is because of the iterations in the development of the models. The second refinement model is called m3 and the third refinement model is named m4.

Figure B.1 shows the main Proving view of the RODIN platform. The top window displays the hypotheses that are currently available for use in discharging the proof obligation. These have come from the guards of the event that has generated the proof obligation. The window below shows the goal that currently needs to be proved for the proof obligation to be discharged. The bottom window is the proof control. This section includes buttons for running the different available provers, adding an hypothesis that can be typed in the text input box, searching for available hypotheses from the model and finding information about the proof obligation.

```
m4    m4
sendProposal/inv4/INV

m∈MESSAGE
sender(m)=a
messageConversation(m)=c
receiver(m)=pInitiator(messageConversation(m))
messageConversation(m) ↦ sender(m)∈cfpR
messageConversation(m) ↦ sender(m)∈pParticipant
¬
  messageConversation(m) ↦ sender(m)∈proposed
¬
  messageConversation(m) ↦ sender(m)∈refused

State
Goal
proposeS∪{messageConversation(m) ↦ sender(m)}=((proposeM∪{m}) ◁ messageConversation)∼;sender

Proof Control    Rodin Problems
```

FIGURE B.1: Proof Obligation

```
Proof Tree | Proof Information
sendProposal/inv4/INV
• Event in m3
  sendProposal:
    ANY c, a WHERE
      grd1: c ↦ a ∈ cfpR
      grd2: c ↦ a ∉ proposed
      grd3: c ↦ a ∉ refused
      grd4: c ↦ a ∈ participant
    THEN
      act1: proposed ≔ proposed ∪ {c ↦ a}
      act2: proposeS ≔ proposeS ∪ {c ↦ a}
    END
• Event in m4
  sendProposal:
    ANY c, a, m WHERE
      grd1: c ↦ a ∈ cfpR
      grd2: c ↦ a ∉ proposed
      grd3: c ↦ a ∉ refused
      grd4: c ↦ a ∈ pParticipant
      grd5: m ∈ MESSAGE
      grd6: sender(m) = a
      grd7: messageConversation(m) = c
      grd8: receiver(m) = pInitiator(c)
    THEN
      act1: proposed ≔ proposed ∪ {c ↦ a}
      act2: proposeS ≔ proposeS ∪ {c ↦ a}
      act3: proposeM ≔ proposeM ∪ {m}
    END
• Invariant in m4
  inv4: proposeS = (proposeM ◁ messageConversation)∼;sender
```

FIGURE B.2: Proof Information

Selecting the 'i' button in the proof control window displays information about the proof obligation. The information for this example proof obligation is shown in Figure B.2. The proof obligation comes from the an invariant condition and the sendProposal event. The proof is required that the gluing invariant, $proposeS = (proposeM \lhd messageConversation)^{-1};\ sender$, is upheld after the occurence of the `sendProposal` event. The gluing invariant describes the relationship between the variables `proposeS` and `proposeM` using the fields of the messages that are in `proposeM`. The proof obligation must show that after the action of the sendProposal event the relationships between conversation and agents in the `proposeS` variable is equal to the conversations and sender agents of the messages in the `proposeM` variable.

The relationship between the `proposeS` and `proposeM` variables is not currently in the selected hypothesis window and needs to be added for the provers to be able to discharge the proof obligation. To find the required hypothesis the variable name `proposeS` was typed into the text input box and the search hypotheses button was selected. All of the

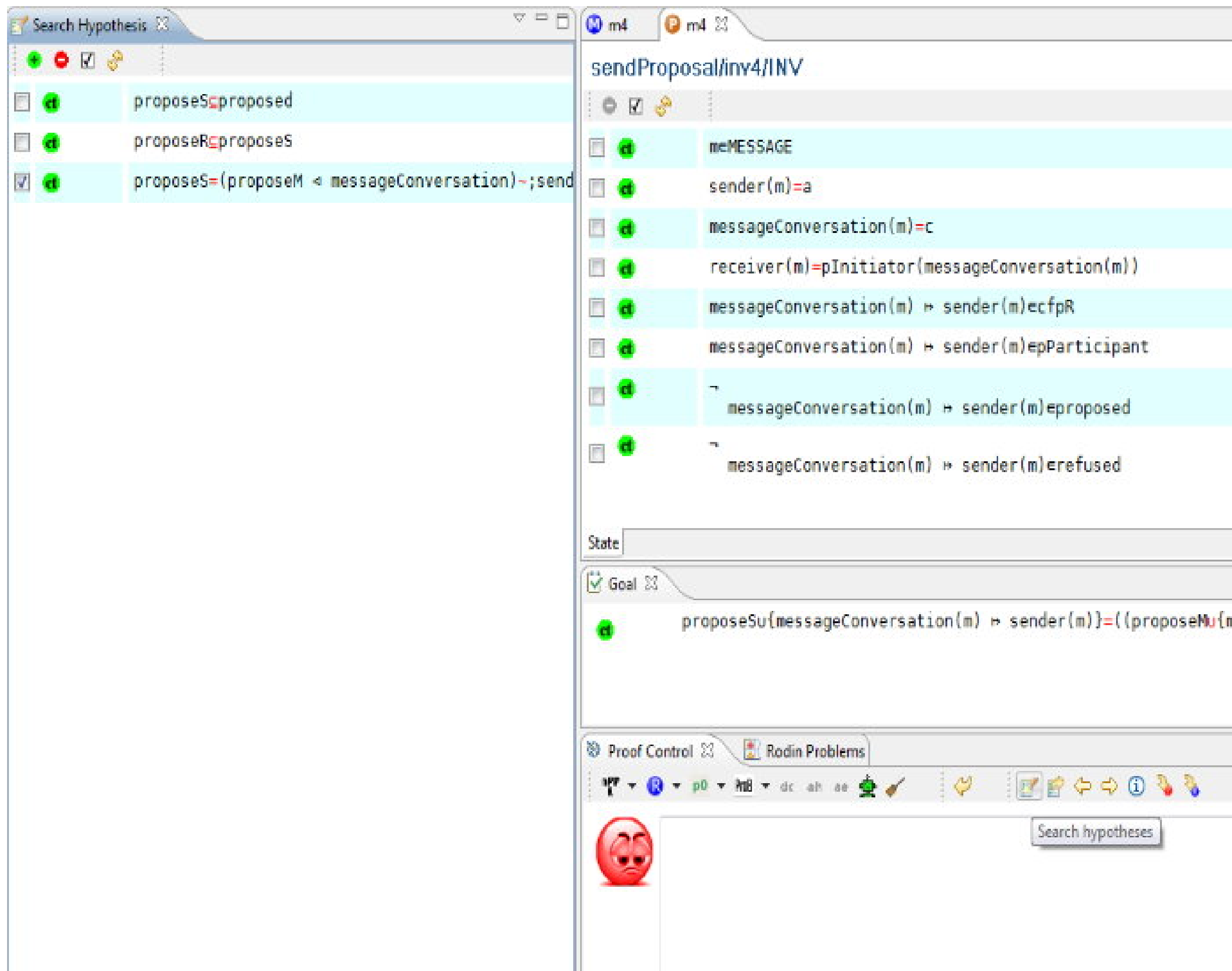

FIGURE B.3: Adding Hypotheses

hypotheses relating to the `proposeS` variable are then displayed in a window to the left of the main proof window. This is shown in Figure B.3. The required hypothesis can then be selected and added to the selected hypotheses window.

Two further hypotheses were needed for the provers to be able to discharge the proof obligation. The hypotheses $sender \in MESSAGE \rightarrow AGENT$ and $messageConversation \in MESSAGE \rightarrow CONVERSATION$ are added to tell the provers that for each message there is only one agent in the sender field and one conversation in the message field. Figure B.4 shows all of the hypotheses in the selected hypotheses window. Adding the hypotheses has automatically changed the form of the goal in the goal window. The `proposeS` variable has now been replaced with the right-hand side of the invariant $proposeS = (proposeM \lhd messageConversation)^{-1};\ sender$.

m4 *m4

sendProposal/inv4/INV

messageConversation(m)=c
receiver(m)=pInitiator(messageConversation(m))
messageConversation(m) ↦ sender(m)∈cfpR
messageConversation(m) ↦ sender(m)∈pParticipant
¬ messageConversation(m) ↦ sender(m)∈proposed
¬ messageConversation(m) ↦ sender(m)∈refused
proposeS=(proposeM ◁ messageConversation)∼;sender
messageConversation∈MESSAGE ⇸ CONVERSATION
sender∈MESSAGE ⇸ AGENT

State

Goal

((proposeM ◁ messageConversation)∼;sender)∪{messageConversation(m) ↦ sender(m)}=((proposeM∪{m})

Proof Control Rodin Problems

Figure B.4: Updated Goal

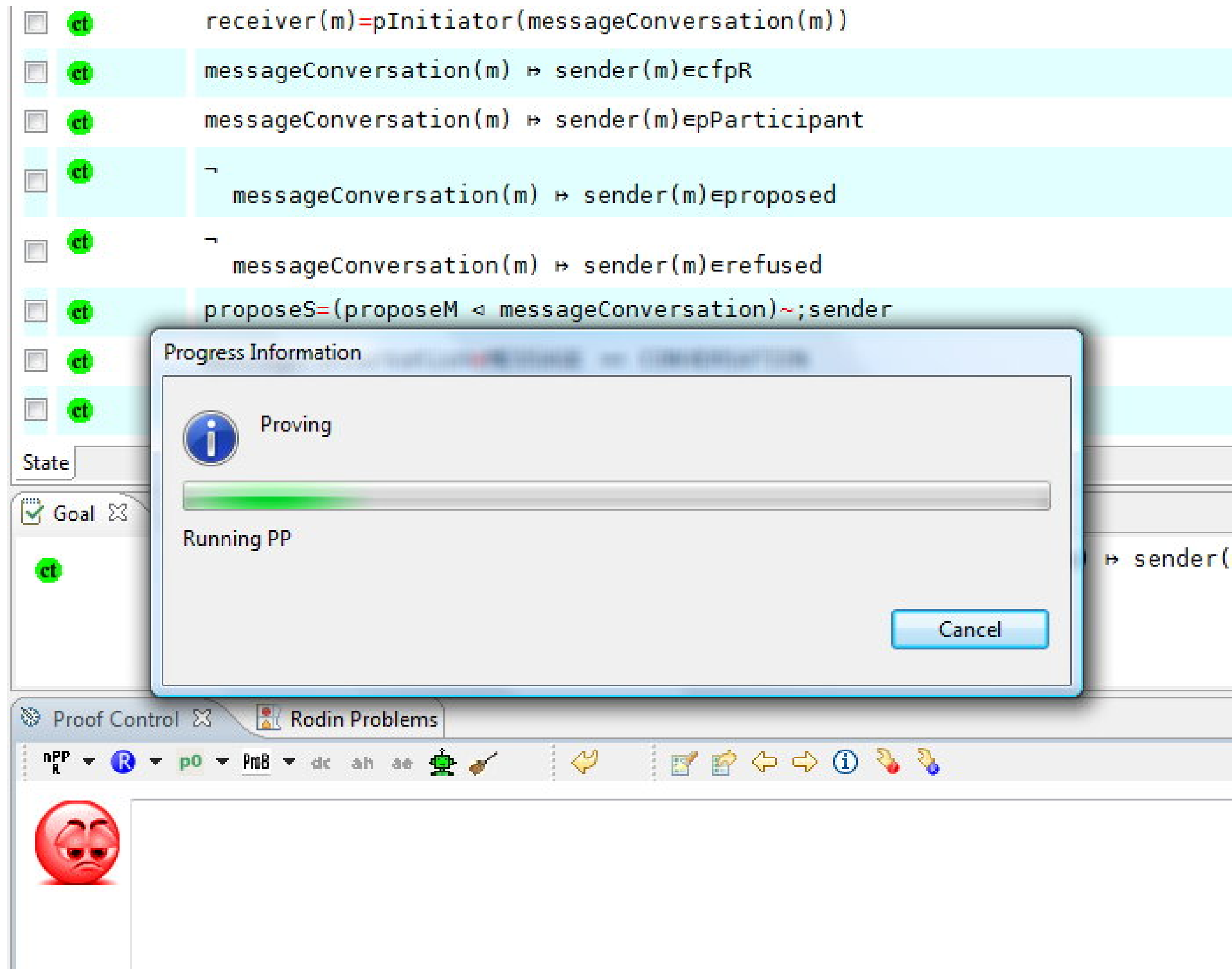

Figure B.5: Running Prover

The information required for a prover to discharge the proof obligation is now available in the selected hypotheses. Figure B.5 shows one of the available provers running in the interactive proving view.

Figure B.6 shows the discharged proof obligation with the proof tree window to the left of the main windows.

FIGURE B.6: Discharged Proof Obligation

TABLE B.1: RODIN Prover Statistics for the Contract Net Case Study

Model	**Total PO's**	**Proved Automatically**	**Proved Interactively**
m0	14	12	2
m1	62	46	12
m2	65	59	2
m3	113	44	69
m4	152	14	138

Table B.1 shows the statistics from the provers in the RODIN toolkit for the Contract Net case study. The table does not include the models involved in decomposition as these were verified using a different toolset. The table shows the total proof obligations generated for the models, the number of which were proved automatically by the RODIN toolkit and the number that needed some degree of interaction for them to be discharged. The first three models use the guidance to create the translation between the goal model and the Event-B models. These generated relatively few proof obligations. The majority of the proof obligations were discharged automatically by the provers. The last two models in the table introduce the messages and then refine the messaging medium. These models involve more data refinement than the earlier models and generate more proof obligations. A lot more of the proof obligations required interaction for them to be discharged. The refinements made to these models follow patterns that are repeated for the variables and events. Discharging the proof obligations generated by these changes using the interactive prover requires the same steps to be repeated for each proof obligation.

Appendix C

Fault-Tolerant Contract Net Case Study Event-B Models

C.1 Context

CONTEXT context

SETS

CONVERSATION

END

C.2 m0 - Abstract Machine

MACHINE m0

SEES context

VARIABLES

cfp
responded
selected
informed

failed
cancelled
recUnknown
recNotUnderstood

INVARIANTS

$inv1 : cfp \subseteq CONVERSATION$
$inv2 : responded \subseteq cfp$
$inv3 : selected \subseteq responded$
$inv4 : informed \subseteq selected$
$inv8 : failed \subseteq cfp$
$inv5 : cancelled \subseteq cfp$
$inv6 : recUnknown \subseteq cfp$
$inv7 : recNotUnderstood \subseteq cfp$

EVENTS

INITIALISATION

BEGIN
$act1 : cfp := \varnothing$
$act2 : responded := \varnothing$
$act3 : selected := \varnothing$
$act4 : informed := \varnothing$
$act5 : cancelled := \varnothing$
$act6 : recUnknown := \varnothing$
$act7 : recNotUnderstood := \varnothing$
$act8 : failed := \varnothing$
END

EVENT callForProposals
ANY
c
WHERE
$grd1 : c \in CONVERSATION$
$grd2 : c \notin cfp$
THEN
$act1 : cfp := cfp \cup \{c\}$
END

EVENT respond

ANY

c

WHERE

$grd1 : c \in cfp$

$grd2 : c \notin responded$

THEN

$act1 : responded := responded \cup \{c\}$

END

EVENT select

ANY

c

WHERE

$grd1 : c \in responded$

$grd2 : c \notin selected$

$grd3 : c \notin failed$

THEN

$act1 : selected := selected \cup \{c\}$

END

EVENT inform

ANY

c

WHERE

$grd1 : c \in selected$

$grd2 : c \notin informed$

$grd3 : c \notin failed$

THEN

$act1 : informed := informed \cup \{c\}$

END

EVENT failure

ANY

c

WHERE

$grd1 : c \in cfp$

$grd2 : c \notin failed$

$grd3 : c \notin informed$

THEN

$act1 : failed := failed \cup \{c\}$

END

EVENT cancel

ANY

c

WHERE

$grd1 : c \in cfp$

$grd2 : c \notin cancelled$

THEN

$act1 : cancelled := cancelled \cup \{c\}$

END

EVENT arbitraryComm

ANY

c

WHERE

$grd1 : c \in cfp$

THEN

$act1 : recUnknown := recUnknown \cup \{c\}$

END

EVENT receiveNotUnderstood

ANY

c

WHERE

$grd1 : c \in cfp$

THEN

$act1 : recNotUnderstood := recNotUnderstood \cup \{c\}$

END

END

C.3 Context2

CONTEXT context2

REFINES context

SETS

AGENT

END

C.4 m1 - First Refinement

MACHINE m1

REFINES m0

SEES context2

VARIABLES

cfpS
proposeS
acceptS
informS
rejectS
cfpR
proposeR
acceptR
rejectR
informR
responded
informed
beforeTimeout
afterTimeout
rejectSD
proposeRD
refuseS
refuseR
cancelS
cancelR
informCancelS
failCancelS

$informCancelR$
$failCancelR$
$informCancelled$
$failCancelled$
$failS$
$failR$
$failed1$
$failedCfp$
$failedCommit$
$unknownR$
$notUnderstoodS$
$notUnderstoodR$
$rejectRD$

INVARIANTS

$inv1 : cfpS \in CONVERSATION \leftrightarrow AGENT$
$inv2 : proposeS \in CONVERSATION \leftrightarrow AGENT$
$inv3 : acceptS \in CONVERSATION \leftrightarrow AGENT$
$inv4 : rejectS \in CONVERSATION \leftrightarrow AGENT$
$inv5 : informS \in CONVERSATION \leftrightarrow AGENT$
$inv6 : cfpR \subseteq cfpS$
$inv7 : proposeR \subseteq proposeS$
$inv19 : refuseS \subseteq cfpR$
$inv20 : refuseR \subseteq refuseS$
$inv8 : acceptR \subseteq acceptS$
$inv9 : rejectR \subseteq rejectS$
$inv10 : informR \subseteq informS$
$inv13 : proposeS \subseteq cfpR$
$inv44 : acceptS \subseteq proposeR$
$inv14 : informS \subseteq acceptR$
$inv21 : proposeS \cap refuseS = \varnothing$
$inv15 : beforeTimeout \subseteq dom(cfpS)$
$inv16 : afterTimeout \subseteq beforeTimeout$
$inv18 : proposeRD \subseteq proposeS$
$inv17 : rejectSD \subseteq proposeRD$
$inv42 : rejectRD \subseteq rejectSD$
$inv22 : cancelS \subseteq cfpS$
$inv23 : cancelR \subseteq cancelS$
$inv32 : failS \subseteq acceptR$
$inv24 : informCancelS \subseteq cancelR$

$inv25 : failCancelS \subseteq cancelR$
$inv27 : informCancelR \subseteq informCancelS$
$inv28 : failCancelR \subseteq failCancelS$
$inv26 : informCancelS \cap failCancelS = \varnothing$
$inv29 : informCancelled \subseteq dom(informCancelR)$
$inv30 : failCancelled \subseteq dom(failCancelR)$
$inv43 : informCancelled \cap failCancelled = \varnothing$
$inv37 : informed \cap failed1 = \varnothing$
$inv36 : informed \subseteq dom(informR)$
$inv38 : unknownR \subseteq cfpS$
$inv39 : notUnderstoodS \subseteq unknownR$
$inv41 : notUnderstoodR \subseteq notUnderstoodS$
$inv33 : failR \subseteq failS$
$inv34 : failed1 \subseteq dom(failR)$
$inv46 : failedCfp \subseteq afterTimeout$
$inv47 : failedCommit \subseteq dom(refuseR)$
$inv48 : failedCfp \cap dom(proposeR) = \varnothing$
$inv49 : failedCommit \cap dom(proposeR) = \varnothing$
$inv11 : cfp = dom(cfpS)$
$inv12 : selected = dom(acceptS)$
$inv31 : cancelled = informCancelled \cup failCancelled$
$inv40 : recUnknown = dom(notUnderstoodS)$
$inv45 : failed = failed1 \cup failedCfp \cup failedCommit$

EVENTS

INITIALISATION

BEGIN

$act1 : cfpS := \varnothing$
$act2 : proposeS := \varnothing$
$act3 : acceptS := \varnothing$
$act4 : informS := \varnothing$
$act5 : rejectS := \varnothing$
$act6 : cfpR := \varnothing$
$act7 : proposeR := \varnothing$
$act8 : acceptR := \varnothing$
$act9 : rejectR := \varnothing$
$act10 : informR := \varnothing$
$act11 : responded := \varnothing$
$act12 : informed := \varnothing$

$act13 : beforeTimeout := \varnothing$
$act14 : afterTimeout := \varnothing$
$act15 : rejectSD := \varnothing$
$act16 : proposeRD := \varnothing$
$act17 : refuseS := \varnothing$
$act18 : refuseR := \varnothing$
$act19 : cancelS := \varnothing$
$act20 : cancelR := \varnothing$
$act21 : informCancelS := \varnothing$
$act22 : failCancelS := \varnothing$
$act23 : informCancelR := \varnothing$
$act24 : failCancelR := \varnothing$
$act25 : informCancelled := \varnothing$
$act26 : failCancelled := \varnothing$
$act27 : failS := \varnothing$
$act28 : failR := \varnothing$
$act29 : failed1 := \varnothing$
$act30 : unknownR := \varnothing$
$act31 : notUnderstoodS := \varnothing$
$act32 : notUnderstoodR := \varnothing$
$act33 : rejectRD := \varnothing$
$act34 : failedCfp := \varnothing$
$act35 : failedCommit := \varnothing$
END

EVENT sendCfp
REFINES callForProposals
ANY
c
as
a
WHERE
$grd1 : c \in CONVERSATION$
$grd2 : c \notin dom(cfpS)$
$grd3 : as \in CONVERSATION \leftrightarrow AGENT$
$grd6 : a \in AGENT$
$grd4 : dom(as) = \{c\}$
$grd5 : ran(as) = AGENT \setminus \{a\}$
THEN
$act1 : cfpS := cfpS \cup as$
$act2 : beforeTimeout := beforeTimeout \cup \{c\}$

END

EVENT receiveCfp

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpS$

$grd2 : c \mapsto a \notin cfpR$

THEN

$act1 : cfpR := cfpR \cup \{c \mapsto a\}$

END

EVENT sendProposal

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin proposeS$

$grd3 : c \mapsto a \notin refuseS$

THEN

$act1 : proposeS := proposeS \cup \{c \mapsto a\}$

END

EVENT receiveProposal

ANY

c

a

WHERE

$grd1 : c \mapsto a \in proposeS$

$grd2 : c \mapsto a \notin proposeR$

$grd3 : c \notin afterTimeout$

$grd4 : c \notin failedCommit$

THEN

$act1 : proposeR := proposeR \cup \{c \mapsto a\}$

END

EVENT deadline

ANY

c

WHERE

$grd1 : c \in beforeTimeout$

$grd2 : c \notin afterTimeout$

THEN

$act1 : afterTimeout := afterTimeout \cup \{c\}$

END

EVENT receiveProposal2

ANY

c

a

WHERE

$grd1 : c \mapsto a \in proposeS$

$grd2 : c \mapsto a \notin proposeR$

$grd3 : c \mapsto a \notin proposeRD$

$grd4 : c \in afterTimeout$

THEN

$act1 : proposeRD := proposeRD \cup \{c \mapsto a\}$

END

EVENT failToPropose

REFINES failure

ANY

c

WHERE

$grd1 : c \notin dom(proposeR)$

$grd2 : c \in afterTimeout$

$grd3 : c \notin failedCfp$

THEN

$act1 : failedCfp := failedCfp \cup \{c\}$

END

EVENT sendRefusal

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpR$

$grd2 : c \mapsto a \notin refuseS$

$grd3 : c \mapsto a \notin proposeS$

THEN

$act1 : refuseS := refuseS \cup \{c \mapsto a\}$
END

EVENT receiveRefusal
ANY
c
a
WHERE
$grd1 : c \mapsto a \in refuseS$
$grd2 : c \mapsto a \notin refuseR$
THEN
$act1 : refuseR := refuseR \cup \{c \mapsto a\}$
END

EVENT sendReject
ANY
c
a
WHERE
$grd1 : c \mapsto a \in proposeRD$
$grd2 : c \mapsto a \notin rejectSD$
THEN
$act1 : rejectSD := rejectSD \cup \{c \mapsto a\}$
END

EVENT responded
REFINES respond
ANY
c
WHERE
$grd1 : c \in dom(proposeR)$
$grd2 : c \notin responded$
THEN
$act1 : responded := responded \cup \{c\}$
END

EVENT failToCommit
REFINES failure
ANY
c
WHERE

$grd1 : c \in dom(refuseR)$
$grd2 : c \notin dom(proposeR)$
$grd3 : c \notin failedCommit$
THEN
$act1 : failedCommit := failedCommit \cup \{c\}$
END

EVENT select
REFINES select
ANY
c
as
ar
WHERE
$grd1 : c \in dom(proposeR)$
$grd2 : c \notin dom(acceptS)$
$grd3 : c \notin dom(rejectS)$
$grd4 : as \subseteq \{c\} \lhd proposeR$
$grd5 : as \neq \varnothing$
$grd6 : ar = \{c\} \lhd proposeR \setminus as$
$grd7 : c \in responded$
THEN
$act1 : acceptS := acceptS \cup as$
$act2 : rejectS := rejectS \cup ar$
END

EVENT receiveAccept
ANY
c
a
WHERE
$grd1 : c \mapsto a \in acceptS$
$grd2 : c \mapsto a \notin acceptR$
THEN
$act1 : acceptR := acceptR \cup \{c \mapsto a\}$
END

EVENT receiveReject
ANY
c
a

WHERE

$grd1 : c \mapsto a \in rejectS$

$grd2 : c \mapsto a \notin rejectR$

THEN

$act1 : rejectR := rejectR \cup \{c \mapsto a\}$

END

EVENT receiveReject2

ANY

c

a

WHERE

$grd1 : c \mapsto a \in rejectSD$

$grd2 : c \mapsto a \notin rejectRD$

THEN

$act1 : rejectRD := rejectRD \cup \{c \mapsto a\}$

END

EVENT sendInform

ANY

c

a

WHERE

$grd1 : c \mapsto a \in acceptR$

$grd2 : c \mapsto a \notin informS$

$grd3 : c \mapsto a \notin failS$

THEN

$act1 : informS := informS \cup \{c \mapsto a\}$

END

EVENT receiveInform

ANY

c

a

WHERE

$grd1 : c \mapsto a \in informS$

$grd2 : c \mapsto a \notin informR$

THEN

$act1 : informR := informR \cup \{c \mapsto a\}$

END

EVENT informed
REFINES inform
ANY
c
WHERE
$grd1 : c \in dom(informR)$
$grd2 : c \notin informed$
$grd3 : c \notin failed1$
THEN
$act1 : informed := informed \cup \{c\}$
END

EVENT sendCancel
ANY
c
as
WHERE
$grd1 : c \in dom(cfpS)$
$grd2 : as = \{c\} \lhd cfpS$
$grd3 : c \notin dom(cancelS)$
THEN
$act1 : cancelS := cancelS \cup as$
END

EVENT receiveCancel
ANY
c
a
WHERE
$grd1 : c \mapsto a \in cancelS$
$grd2 : c \mapsto a \notin cancelR$
THEN
$act1 : cancelR := cancelR \cup \{c \mapsto a\}$
END

EVENT sendInformCancel
ANY
c
a
WHERE
$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin informCancelS$

$grd3 : c \mapsto a \notin failCancelS$

THEN

$act1 : informCancelS := informCancelS \cup \{c \mapsto a\}$

END

EVENT sendFailCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cancelR$

$grd2 : c \mapsto a \notin failCancelS$

$grd3 : c \mapsto a \notin informCancelS$

THEN

$act1 : failCancelS := failCancelS \cup \{c \mapsto a\}$

END

EVENT receiveInformCancel

ANY

c

a

WHERE

$grd1 : c \mapsto a \in informCancelS$

$grd2 : c \mapsto a \notin informCancelR$

THEN

$act1 : informCancelR := informCancelR \cup \{c \mapsto a\}$

END

EVENT informCancel

REFINES cancel

ANY

c

WHERE

$grd1 : c \in dom(informCancelR)$

$grd2 : c \notin informCancelled$

$grd3 : c \notin failCancelled$

THEN

$act1 : informCancelled := informCancelled \cup \{c\}$

END

EVENT failCancel
REFINES cancel
ANY
c
WHERE
$grd1 : c \in dom(failCancelR)$
$grd2 : c \notin failCancelled$
$grd3 : c \notin informCancelled$
THEN
$act1 : failCancelled := failCancelled \cup \{c\}$
END

EVENT sendFail
ANY
c
a
WHERE
$grd1 : c \mapsto a \in acceptR$
$grd2 : c \mapsto a \notin failS$
$grd3 : c \mapsto a \notin informS$
THEN
$act1 : failS := failS \cup \{c \mapsto a\}$
END

EVENT receiveFail
ANY
c
a
WHERE
$grd1 : c \mapsto a \in failS$
$grd2 : c \mapsto a \notin failR$
THEN
$act1 : failR := failR \cup \{c \mapsto a\}$
END

EVENT failed
REFINES failure
ANY
c
WHERE
$grd1 : c \in dom(failR)$

$grd2 : c \notin failed1$

$grd3 : c \notin informed$

THEN

$act1 : failed1 := failed1 \cup \{c\}$

END

EVENT receiveArbitraryComm

ANY

c

a

WHERE

$grd1 : c \mapsto a \in cfpS$

THEN

$act1 : unknownR := unknownR \cup \{c \mapsto a\}$

END

EVENT sendNotUnderstood

REFINES arbitraryComm

ANY

c

a

WHERE

$grd1 : c \mapsto a \in unknownR$

$grd2 : c \mapsto a \notin notUnderstoodS$

THEN

$act1 : notUnderstoodS := notUnderstoodS \cup \{c \mapsto a\}$

END

EVENT receiveNotUnderstood

REFINES receiveNotUnderstood

ANY

c

a

WHERE

$grd1 : c \mapsto a \in notUnderstoodS$

$grd2 : c \mapsto a \notin notUnderstoodR$

THEN

$act1 : notUnderstoodR := notUnderstoodR \cup \{c \mapsto a\}$

END

END

Bibliography

J-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, 8th International Conference on Formal Engineering Methods, ICFEM 2006, volume 4260/2006 of Lecture Notes in Computer Science, pages 588 – 605, Macao, China, 2006. Springer-Verlag.

J-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae, XXI, 77(1 - 2):1 – 28, 2006.

Jean-Raymond Abrial. The B-Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge, 1996.

Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In David Basin and Burkhart Wolff, editors, Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, volume 2758 of Lecture Notes in Computer Science, pages 1–24, Rome, Italy, 2003. Springer-Verlag.

Jean-Raymond Abrial and L. Mussat. Introducing dynamic constraints in B. In D Bert, editor, Second International B Conference B'98: Recent Advances in the Development and Use of the B Method, volume 1393 of Lecture Notes in Computer Science, pages 83 – 128, Montpellier, France, 1998. Springer.

C. Alexander, S. Ishikawa, and M. Silverstein. A pattern language: towns, buildings, construction. Oxford University Press, 1977.

T. Anderson and P.A. Lee. Fault Tolerance: Principles and Practice. Prentice-Hall International Inc., New Jersey, USA, 1981.

Y. Aridor and D.B. Lange. Agent design patterns: elements of agent application design. In Proceedings of the second international conference on Autonomous agents, pages 108–115. ACM Press New York, NY, USA, 1998.

R.J. Back. Incremental software construction with refinement diagrams. In Manfred Broy, Johannes Gruenbauer, David Harel, and Tony Hoare, editors, Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, pages 3 – 46, Marktoberdorf, Germany, 2005. Springer.

R.J. Back, A. Akademi, J. Von Wright, F.B. Schneider, and D. Gries. Refinement Calculus: A Systematic Introduction. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1998.

R.J. Back and K. Sere. Stepwise refinement of action systems. Structured Programming, 12(1):17–30, 1991.

Bernhard Bauer. UML class diagrams revisited in the context of agent-based systems. In Michael Wooldridge, Paolo Ciancarini, and Gerhard Weiβ, editors, Agent Oriented Software Engineering II: Second International Workshop, volume 2222 of Lecture Notes in Computer Science, pages 101–119. Springer-Verlag, 2001.

Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A formalism for specifying multiagent interaction. In Paolo Ciancarini and Michael Wooldridge, editors, Agent-Oriented Software Engineering, volume 1957 of Lecture Notes in Computer Science, pages 91–103. Springer-Verlag, Berlin, 2001.

Bernhard Bauer and James Odell. UML 2.0 and agents: How to build agent-based systems with the new UML standard. Journal of Engineering Applications of Artificial Intelligence, 18(2):141–157, 2005.

K. Beck and R.E. Johnson. Patterns generate architectures. In Mario Tokoro and Remo Pareschi, editors, ECOOP'94. Proceedings of the 8th European Conference on Object-Oriented Programming, volume 821 of Lecture Notes in Computer Science, pages 139–149, Bologna, Italy, 1994. Springer-Verlag London, UK.

Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A succesful application of B in a large project. In J.M. Wing, J. Woodcock, and J. Davis, editors, FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, volume 1708 of Lecture Notes in Computer Science, pages 369–387, Toulouse, France, 1999. Springer-Verlag.

JA Bergstra, A Ponse, and SA Smolka. Preface. In JA Bergstra, A Ponse, and SA Smolka, editors, Handbook of Process Algebra. Elsevier Science Inc. New York, NY, USA, 2001.

DM Berry and E Kamsties. Ambiguity in requirements specification. In Sampaio do Prado Leite, Doorn Julio Cesar, and Jorge Horacio, editors, Perspectives on Software Requirements, volume 753 of The Springer International Series in Engineering and Computer Science, pages 7 – 44. Kluwer Academic Publishers, 2004.

Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of specification patterns with the B Method. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, ZB 2003: Formal Specification and Development in Z and B, volume 2651 of Lecture Notes in Computer Science, pages 40 – 57, Turku, Finland, 2003. Springer-Verlag.

Grady Booch. Object Oriented Design with Applications. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, USA, 1991.

Rafael H. Bordini, Mehdi Dastani, and Michael Winikoff. Current issues in multi-agent systems development. In G. OHare, A. Ricci, M. OGrady, and O. Dikenelli, editors, Engineering Societies in the Agents World VII 7th International Workshop, ESAW 2006, volume 4457 of Lecture Notes in Computer Science, pages 38–61, Dublin, Ireland, 2006. Springer-Verlag.

F.M.T. Brazier, C.M. Jonker, and J.Treur. Principles of compositional multi-agent systems development. In J. Cuena, editor, Proceedings of the 15th IFIP World Computer Congress, WCC'98, Conference on Information Technology and Knowledge Systems, pages 347–360, 1998.

Frances M. T. Brazier, Barbara M. Dunin-Keplicz, Nicholas R. Jennings, and Jan Treur. Formal specification of multi-agent systems: a real-world case. In First International Conference on Multi-agent Systems, pages 25 – 32, San Fransisco, CA, USA, 1995. AAAI Press, CA, USA.

Frances M. T. Brazier, Barbara M. Dunin-Keplicz, Nicholas R. Jennings, and Jan Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. International Journal of Cooperative Information Systems, 6(1):67–94, 1997.

Paolo Bresciani and Paolo Donzelli. REF: A practical agent-based requirement engineering framework. In Conceptual Modelling for Novel Application Domains. ER 2003 Workshops, volume 2814 of Lecture Notes in Computer Science, pages 217–228, Chicago, IL, USA, 2003. Springer-Verlag.

Paolo Bresciani, Anna Perini, Paulo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems, 8:203–236, 2004.

FP Brooks. No silver bullet: Essence and accidents of software engineering. IEEE Computer, 20(4):10–19, 1987.

M Butler and C Ferreira. An operational semantics for stac, alanguage for modelling long-running business transactions. In Rocco De Nicola, Gianluigi Ferrari, and Greg Meredith, editors, Coordination Models and Languages: 6th International Conference, COORDINATION 2004, volume 2949 of Lecture Notes in Computer Science, pages 87–104, Pisa, Italy, 2004. Springer.

M Butler and M Leuschel. Combining csp and b for specification and property verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, FM 2005: Formal Methods: International Symposium of Formal Methods Europe, volume 3582, pages 221–236, Newcastle, UK, 2005. Springer.

M. Butler and D. Yadav. An incremental development of the Mondex System in Event-B. Formal Aspects of Computing, 20(1):61–77, 2007.

Michael Butler. RODIN deliverable 16. prototype plug-in tools. Available From: http://rodin.cs.ncl.ac.uk/deliverables/rodinD16.pdf. Technical report, University of Newcastle-upon-Tyne, UK, 2006.

Michael Butler, Michael Leuschel, and Colin Snook. Tools for system validation with B abstract machines. In Proceedings of ASM 2005: 12th International Workshop on Abstract State Machines, pages 57 – 69, Paris, 2005.

M.J. Butler. Stepwise refinement of communicating systems. Science of Computer Programming, 27(2):139–173, 1996.

Dominique Cansell and Dominique Méry. Time constraint patterns for Event B development. In Jacques Julliand and Olga Kouchnarenko, editors, B 2007: Formal Specification and Development in B, volume 4355 of Lecture Notes in Computer Science, pages 140 – 154, Besancon, France, 2007. Springer-Verlag.

Edward Chan, Ken Robinson, and Brett Welch. Patterns for B: Bridging formal and informal development. In Jacques Julliand and Olga Kouchnarenko, editors, B 2007: Formal Specification and Development in B, volume 4355 of Lecture Notes in Computer Science, pages 125 – 139, Besancon, France, 2007. Springer-Verlag.

P Cilliers. Complexity and Postmodernism: Understanding Complex Systems. Routledge, 1998.

Edmund M. Clarke Jr., Orna Grumberg, and Doran A Peled. Model Checking. The MIT Press, USA, 2000.

ClearSy. B4free home page. Available From: http://www.b4free.com/, 2005.

P. Coad. Object-oriented patterns. Communications of the ACM, 35(9):152–159, 1992.

G Coulouris, J Dollimore, and T Kindberg. Distributed systems: concepts and design. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

D. De Roure, M.A. Baker, N.R. Jennings, and N.R. Shadbolt. The evolution of the Grid. Grid Computing: Making the Global Infrastructure a Reality, 2003.

Dwight Deugo, Michael Weiss, and Elizabeth Kendall. Reusable patterns for agent coordination. In A. Omicini, F. Zambonelli, M. Klusch, and R Tolksdorf, editors, Coordination of Internet Agents: Models, Technologies and Applications, pages 347 – 368. Springer, 2001.

Antoni Diller. Z: An Introduction to Formal Methods, Second Edition. John Wiley & Sons, Chichester, UK, second edition, 1994.

M. d'Inverno and M. Luck. Development and application of a formal agent framework. In M. G. Hinchey and L. Shaoying, editors, ICFEM'97: First IEEE International Conference on Formal Engineering Methods, pages 222–231, Hiroshima, Japan, 1997. IEEE Computer Society.

Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In Rao Singh and Michael Wooldridge, editors, Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, volume 1365 of Lecture Notes in AI, pages 155–176. Springer-Verlag, 1998.

Paolo Donzelli. A goal-driven and agent-based requirements engineering framework. Requirements Engineering, 9(1):16–39, 2004.

Eclipse. Eclipse platform homepage. Available From: http://www.eclipse.org, 2007.

A. Edmunds and M. Butler. Linking event-b and concurrent object-oriented programs. In Refine 2008 - International Refinement Workshop, page In Press, Turku, Finland, 2008. Springer-Verlag.

N. Evans and M. Butler. A proposal for records in Event-B. In T. Nipkow and J. Misra, editors, Formal Methods 2006, volume 4085 of Lecture Notes in Computer Science, pages 221 – 235, McMaster, Canada, 2006. Springer-Verlag.

Hind Fadil and Jean-Luc Koning. A formal approach to model mulitagent interactions using the B formal method. In Felix F. Romas, Victor Lrios Rosillo, and Ungerm Herwig, editors, 5th International School and Symposium, ISSADS 2005, volume 3563 of Lecure Notes in Computer Science, pages 516 – 528, Guadalajara, Mexico, 2005. Springer-Verlag.

R Fagin, JY Halpern, MY Vardi, and Y Moses. Reasoning about knowledge. MIT Press, Cambridge, Massachusetts, USA, 2003.

A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2, pages 737–744. ACM Press New York, NY, USA, 2002.

Jacques Ferber. Multi-Agent Systems: Introduction to Distributed Artificial Intelligence. Addison Wesley, 1999.

FIPA. FIPA ACL message structure specification. Available From: http://www.fipa.org/specs/fipa00061/SC00061G.pdf. Technical report, FIPA, 2002a.

FIPA. FIPA brokering interaction protocol specification. Available From: http://www.fipa.org/specs/fipa00033/SC00033H.pdf. Technical report, FIPA, 2002b.

FIPA. FIPA contract net interaction protocol specification. Available From: http://www.fipa.org/specs/fipa00029/SC00029H.pdf. Technical report, FIPA, 2002c.

FIPA. FIPA query interaction protocol specification. Available From: http://www.fipa.org/specs/fipa00027/SC00027H.pdf. Technical report, FIPA, 2002d.

FIPA. FIPA request interaction protocol specification. Available From: http://www.fipa.org/specs/fipa00026/SC00026H.pdf. Technical report, FIPA, 2002e.

Michael Fisher. Concurrent METATEM - a language for modelling reactive systems. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, PARLE '93 Parallel Architectures and Languages Europe, volume 694 of Lecture Notes in Computer Science, pages 185–196, Munich, Germany, 1993. Springer-Verlag.

Michael Fisher. Temporal development methods for agent-based systems. Autonomous Agents and Multi-Agent Based Systems, 10:41–66, 2005.

Michael Fisher. Metatem: The story so far. In R.H. Bordini, M. Dastani, J. Dix, and A.E.F. Seghrouchni, editors, Programming Multi-Agent Systems. Third International Workshop, ProMAS 2005, volume 3862 of Lecture Notes in Artificial Intelligence, pages 3 – 22, Utrecht, The Netherlands, 2006. Springer-Verlag.

Michael Fisher, Rafael H. Bordini, Benjamin Hirsch, and Paolo Torroni. Computational logics and agents: A road map of current technologies and future trends. Computational Intelligence, 23(1):61 – 91, 2007.

Michael Fisher and Michael Wooldridge. Towards formal methods for agent-based systems. In D Duke and A S Evans, editors, Proceedings of the BCS-FACS Nothern Formal Methods Workshop, Electronic Workshops in Computing, Ilkley, UK, 1996. Springer.

Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In Jörg P. Müller, Michael J. Wooldridge, and Nicholas R. Jennings, editors, ECAI'96 Workshop. Intelligent Agents III. Agent Theories, Architectures, and Languages, volume 1193 of Lecture Notes in Computer Science, pages 21 – 35, Budapest, Hungary,, 1996. Springer-Verlag.

A. Fuxman, R. Kazhamiakin, M. Pistore, and M. Roveri. Formal Tropos: Language and semantics. Available From: http://dit.unitn.it/ ft/papers/ftsem03.pdf. Technical report, Trento, 2003.

A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in tropos. Requirements Engineering, 9(2):132–150, 2004.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

H Gao, Y He, Z Qin, L Shao, and X Heng. Application of event b to modelling multi-agent interactions. In Information and Communication Technology, 2007. ICICT'07. International Conference on, pages 197–200, Cairo, EGYPT, 2007. IEEE.

Mark Greaves, Victoria Stavridou-Coleman, and Robert Laddaga. Guest editors introduction: Dependable agent systems. IEEE Intelligent Systems, 19(5):20 – 23, 2004.

A. Hall. Seven myths of formal methods. Software, IEEE, 7(5):11–19, 1990.

A. Hall. Using formal methods to develop an ATC information system. Software, IEEE, 13(2):66–76, 1996.

Stefan Hallerstede. Justifications for the Event-B modelling notation. In J. Julliand and O. Kouchnarenko, editors, B2007, 7th International Conference of B Users, volume 4355 of Lecture Notes in Computer Science, pages 46 – 63, Besancon, France, 2007. Springer.

D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, Logics and Models of Concurrent Systems, volume F-13 of NATO ASI Series, pages 477 – 498, New York, USA, 1985. Springer-Verlag.

C. Heitmeyer. On the need for practical formal methods. In Anders P. Ravn and Hans Rischel, editors, FTRTFT'98, Formal Techniques in Real-Time and Fault-Tolerant Systems, volume 1486 of Lecture Notes in Computer Science, pages 18–26, Lyngby, Denmark, 1998. Springer.

M.G. Hinchey and J.P. Bowen. To formalize or not to formalize. IEEE Computer, 29 (4):18–19, 1996.

CAR Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, 1969.

CAR Hoare. Communicating Sequential Processes. Prentice Hall International, 1985.

I. Houstan and S. King. CICS project report: Experiences and results from the use of Z in IBM. VDM91. Formal software development methods, vol. 1: Conference contribution. In Soren Prehn and Hans Toetenel, editors, Lecture Notes in Computer Science, volume 552, pages 588–596, Noordwijkerhout, The Netherlands, 1991. Springer-Verlag.

M.P. Huget and J.L. Koning. Interaction protocol engineering. In Marc-Phillipe Huget, editor, Communication in Multiagent Systems: Agent Communication Languages and Conversation Policies, volume 2650 of Lecture Notes in Artificial Intelligence, page 209222. Springer, 2003.

M Huth and M Ryan. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, 2004.

Carlos A. Iglesias, Mercedes Garijo, José C. Gonzáles, and Juan R. Velasco. Analysis and design of multiagents systems using MAS-commonKADS. In Munindar P. Singh, Anand Rao, and Michael J. Wooldridge, editors, 4th International Workshop, ATAL'97: Intelligent Agents IV: Agent Theories, Architectures, and Languages, Lecture Notes in Artificial Intelligence, pages 313–327, Providence, Rhode Island, USA, 1997. Springer-Verlag.

Alexei Iliasov, Linas Laibinis, Alexander Romanovsky, and Elena Troubitsyna. Towards formal development of mobile location-based systems. In Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, Proceedings of the Workshop on Rigorous Engineering of Fault Tolerant Systems, Newcastle upon Tyne, 2005.

Alexei Iliasov, Linas Laibinis, Alexander Romanovsky, and Elena Troubitsyna. Rigorous development of fault-tolerant agent systems. Available From: http://www.tucs.fi/research/publications/insight.php?id=tIlLaRoTr06a&table=techreport. Technical Report 776, Turku Centre for Computer Science, 2006.

D. Jackson. Dependable software by design. Scientific American - American Edition, 294(6):68, 2006.

Nicholas R. Jennings. On agent-based software engineering. Artificial Intelligence, 117: 277–296, 2000.

N.R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. The Knowledge Engineering Review, 8(3):223250, 1993.

C. B. Jones. Systematic Software Development using VDM, Second Edition. Prentice Hall, Upper Saddle River, NJ, USA, 1990.

C. B. Jones. RODIN deliverable D9. preliminary report on methodology. Available From: http://rodin.cs.ncl.ac.uk/deliverables/rodinD9.pdf. Technical report, University of Newcastle-upon-Tyne, UK, 2005.

C. B. Jones. RODIN deliverable D19. intermediate report on methodology. Available From: http://rodin.cs.ncl.ac.uk/deliverables/D19.pdf. Technical report, University of Newcastle-upon-Tyne, UK., 2006.

C.B. Jones, D. Jackson, and J. Wing. Formal methods light. Computer, 29(4):20–22, 1996.

A. Krishna, A.K. Ghose, and S.A. Vilkomir. Co-evolution of complementary formal and informal requirements. In Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of, pages 159–164, 2004.

S. Kumar and P.R. Cohen. Towards a fault-tolerant multi-agent system architecture. In Proceedings of the Fourth International Conference on Autonomous Agents, pages 459–466. ACM Press New York, NY, USA, 2000.

L. Lamport. Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering, 3(2):125–143, 1977.

Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In RE'01 International Joint Conference on Requirements Engineering, pages 249–263, Toronto, 2001. IEEE.

G.T. Leavens, J-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, and S. Peyton-Jones. Roadmap for enhanced languages and methods to aid verification. In Proceedings of the 5th international conference on Generative programming and component engineering, pages 221–236, Portland, Oregan, USA, 2006. ACM Press New York, NY, USA.

E. Letier. Reasoning About Agents in Goal Oriented Requirements Engineering. PhD thesis, Universit Catholique de Louvain, 2001.

E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In Proceedings of the 24th International Conference on Software Engineering, pages 83–93. ACM Press New York, NY, USA, 2002.

M. Leuschel and M. Butler. Automatic refinement checking for B. In Kung-Kiu Lau and Richard Banach, editors, International Conference on Formal Engineering Methods (ICFEM 2005), volume 3785 of Lecture Notes in Computer Science, page 345 359, Manchester, UK, 2005. Springer-Verlag.

Michael Leuschel and Michael Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, FME 2003: Formal Methods, volume 2805 of Lecture Notes in Computer Science, pages 855–874. Springer Verlag, 2003.

M. Luck and M. d'Inverno. Formal methods and agent-based systems. In C. Rouff, M. Hinchey, J. Rash, W. Truszkowski, and D. Gordon-Spears, editors, Agent Technology from a Formal Perspective, NASA Monographs in Systems and Software Engineering. Springer, 2006.

Michael Luck and Mark d'Inverno. A formal framework for agency and autonomy. In Proceedings of the First International Conference on Multi-Agent Systems, pages 254–260. AAAI Press/ MIT Press, 1995a.

Michael Luck and Mark d'Inverno. Structuring a Z specification to provide a formal framework for autonomous agent systems. In J. Bowen and M. Hinchey, editors, ZIM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, volume 967 of Lecture Notes in Computer Science, pages 47–62. Springer-Verlag, 1995b.

Michael Luck and Mark d'Inverno. Development and application of a formal agent framework. In Proceedings of the First IEEE International Conference on Formal Engineering Methods, Hiroshima, Japan, 1997.

C. Méteyer, Jean-Raymond Abrial, and L. Voisin. Rodin deliverable 3.2. Event-B language. Available From: http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf. Technical report, University of Newcastle-upon-Tyne, UK., 2005.

B. Meyer. Object-oriented software construction. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. Software Engineering, IEEE Transactions on, 21(4):356–372, 1995.

N. Neagu, K. Dorer, D. Greenwood, and M. Calisti. LS/ATN: Reporting on a successful agent-based solution for transport logistics optimization. In IEEE 2006 Workshop on Distributed Intelligent Systems (WDIS'06), Prague, 2006.

G.A. Papadopoulos. Models and technologies for the coordination of internet agents: A survey. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, Coordination of Internet Agents: Models, Technologies, and Applications, page 2556. Springer, 2000.

H.V.D. Parunak and J. Odell. Representing social structures in UML. In Michael J. Wooldridge, Gerhard Weiβ, and Paolo Ciancarini, editors, Agent-Oriented Software Engineering II: Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001: Revised Papers and Invited Contributions, volume 2222 of Lecture Notes in Computer Science, pages 1 – 17. Springer, 2002.

S Paurobally, J Cunningham, and NR Jennings. A formal framework for agent interaction semantics. In Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, pages 91–98, Utrecht Netherlands, 2005. ACM Press New York, NY, USA.

A. Perini, M. Pistore, M. Roveri, and A. Susi. Agent-oriented modelling by interleaving formal and informal specification. In P. Giorgini, J.P. Müller, and J. Odell, editors, Agent Oriented Software Engineering 2003, volume 2935 of Lecture Notes in Computer Science, pages 36–52. Springer-Verlag, 2004.

Tolety Siva Perraju. Multi agent architectures for high assurance systems. In American Control Conference, pages 3154 – 3157, San Diego, California, USA, 1999.

M.R. Poppleton. Towards feature-oriented specification and development with Event-B. In Pete Sawyer, Barbara Paech, and Patrick Heymans, editors, 13th International Working Conference, REFSQ 2007, volume 4542 of Lecture Notes in Computer Science, pages 367 – 381, Trondheim, Norway, 2007. Springer.

R.S. Pressman. Software engineering: a practitioner's approach: European adaptation. McGraw-Hill, 2000.

J. Price. Christopher Alexander's pattern language. Professional Communication, IEEE Transactions on, 42(2):117–122, 1999.

Anand S. Rao and Michael P. Georgeff. Modelling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, San Mateo, California, 1991. Morgan Kaufmann Publishers.

D. Riehle and H. Zuellighoven. Understanding and using patterns in software development. Theory and Practice of Object Systems, 2(1):3–13, 1996.

J. Rushby. Formal Methods and Their Role in the Certification of Critical Systems. SRI International, Computer Science Laboratory, 1995.

D Sangiorgi and D Walker. The [pi]-calculus: A Theory of Mobile Processes. Cambridge University Press, 2001.

K. Schelfthout, T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns. Agent implementation patterns. In J. Debenham, B. Henderson-Sellers, N. Jennings, and J. Odell, editors, Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies, pages 119–130, 2002.

R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers, 29(12):1104–1113, 1980.

C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology (TOSEM), 15(1):92–122, 2006.

C Snook and M Butler. Uml-b and event-b: an integration of languages and tools. In The IASTED International Conference on Software Engineering - SE2008, page (In Press), Innsbruck, Austria, 2008.

Neil Storey. Safety-Critical Computer Systems. Pearson Education Limited, Bath, UK, 1996.

Jan Sudeikat, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Evaluation of agent-oriented software methodologies - examination of the gap between modeling and platform. In P. Giorgini, J.P. Müller, and J. Odell, editors, Agent-Oriented Software Engineering V, Fifth International Workshop, AOSE 2004, volume 3382, pages 126 – 141, New York, NY, USA, 2004. Springer-Verlag.

H Treharne and S Schneider. Using a process algebra to control boperations. In IFM'99 1st International Conference on Integrated Formal Methods, pages 437–457, York, UK, 1999. Springer-Verlag.

Lin Wang, F. Li Hon, Dhrubajyoti Goswami, and Zunce Wei. A fault-tolerant multi-agent development framework. In J. Cao, L.T. Yang, M. Guo, and F. Lau, editors, Parallel and Distributed Processing and Applications, volume 3358 of Lecture Notes in Computer Science, pages 126 – 135, Hong Kong, China, 2004. Springer.

M. Weiss. Pattern-driven design of agent systems: Approach and case study. In J. Eder and M. Missikoff, editors, Conference on Advanced Information Systems Engineering (CAiSE), volume 2681 of Lecture Notes in Computer Science, page 711 723, Klagenfurt/Velden, Austria, 2003. Springer.

Michael Wooldridge and Paul E. Dunne. The computational complexity of agent verification. In John-Jules Meyer and Milind Tambe, editors, The Eighth International Workshop on Agent Theories, Architectures and Languages (ATAL-2001): Intelligent Agents VIII, volume 2333 of Lecture Notes in Computer Science, pages 338 – 350, Seattle, WA, USA, 2001. Springer.

Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. The Knowledge Engineering Review, 10(2):115 – 152, 1995.

Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems, 3: 285–312, 2000.

E Yu. Agent-oriented modelling: Software versus the world. In Michael J. Wooldridge, Gerhard Weiβ, and Paolo Ciancarini, editors, Agent-Oriented Software Engineering II: Second International Workshop, volume 2222 of Lecture Notes in Computer Science, pages 206 – 225, Montreal, Canada,, 2002. Springer.

Eric S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In 3rd IEEE International Symposium on Requirements Engineering (RE'97), page 226. IEEE Computer Society, 1997.

E.S.K. Yu and J. Mylopoulos. Understanding why in software process modelling, analysis, and design. In Proceedings of the 16th international conference on Software engineering, pages 159–168, Sorrento, Italy, 1994. IEEE Computer Society Press, Los Alamitos, CA, USA.

Franco Zambonelli, Nicholas R. Jennings, Andrea Omicini, and Michael Wooldridge. Agent-oriented software engineering for internet applications. In Andrea Omicini, Franco Zambonelli, M Klusch, and R. Tolksdorf, editors, Coordionation of Internet Agents: Models, Technologies and Applications. Springer, 2000.

Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multi-agent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodology, 12(3):317–370, 2003.

Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. Autonomous Agents and Multi-Agent Systems, 9:253–283, 2004.