AN INTERACTION FRAMEWORK FOR MULTIAGENT SYSTEMS

by

MATTHEW JAMES MILLER

B.S., University of Nebraska at Kearney, 2003
M.S., Kansas State University, 2007

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2012

Abstract

A multiagent system is a system that is composed of multiple autonomous agents. Autonomous agents are given the right and the responsibility to make decisions based on their perceptions and goals. Agents are also constrained by their capabilities, the environment and the system with which they reside. An agent within the system may need to coordinate with another agent in the system. This coordination may allow the agent to give updates from sensor readings, communicate updated map information or allow the agent to work on a cooperative task such as lifting an object.

To coordinate agents must be able to communicate with one another. To communicate agents must have a communication medium. The medium is the conduit through which the information flows. Additionally there must be a set of rules to govern which agent talks at what time. This set of rules is called a communication protocol. To effectively and efficiently communicate all agents participating in the communication must be using compatible protocols.

Robotic agents can be placed in diverse environment and there are multiple avenues for communication failure. Current multiagent systems use fixed communication protocols to allow agents to interact with one another. Using fixed protocols in an error prone environment can lead to a high rate of system failure.

To address these issues, I propose that a formal framework for interaction be defined. The framework should allow agents to choose new interaction protocols when the current protocol they are using fails. A formal framework allows automated tools to reason over the possible choices of interaction protocols. The tools can enumerate the protocols that will allow the agent to achieve its desired goal.

AN INTERACTION FRAMEWORK FOR MULTIAGENT SYSTEMS

by

MATTHEW JAMES MILLER

B.S., University of Nebraska at Kearney, 2003
M.S., Kansas State University, 2007

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2012

Approved by:

Dr. Scott DeLoach

Department of Computing and
Information Sciences

Copyright

Abstract

A multiagent system is a system that is composed of multiple autonomous agents. Autonomous agents are given the right and the responsibility to make decisions based on their perceptions and goals. Agents are also constrained by their capabilities, the environment and the system with which they reside. An agent within the system may need to coordinate with another agent in the system. This coordination may allow the agent to give updates from sensor readings, communicate updated map information or allow the agent to work on a cooperative task such as lifting an object.

To coordinate agents must be able to communicate with one another. To communicate agents must have a communication medium. The medium is the conduit through which the information flows. Additionally there must be a set of rules to govern which agent talks at what time. This set of rules is called a communication protocol. To effectively and efficiently communicate all agents participating in the communication must be using compatible protocols.

Robotic agents can be placed in diverse environment and there are multiple avenues for communication failure. Current multiagent systems use fixed communication protocols to allow agents to interact with one another. Using fixed protocols in an error prone environment can lead to a high rate of system failure.

To address these issues, I propose that a formal framework for interaction be defined. The framework should allow agents to choose new interaction protocols when the current protocol they are using fails. A formal framework allows automated tools to reason over the possible choices of interaction protocols. The tools can enumerate the protocols that will allow the agent to achieve its desired goal.

Table of Contents

Table of Contents vi

List of Figures x

List of Tables xi

List of Restrictions xii

List of Definitions xiv

List of Relations xv

Dedication xvi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Thesis Statement . . . 5
- 1.3 Research Approach . . . 6
- 1.4 Scope . . . 7
- 1.5 Thesis Organization . . . 7

2 Background 9
- 2.1 Knowledge . . . 10
- 2.2 Interaction . . . 11
- 2.3 Computer Program Verification . . . 13
  - 2.3.1 Unit Testing . . . 14
  - 2.3.2 Model Checking . . . 15
- 2.4 Protocols . . . 15
  - 2.4.1 Verifying Protocols . . . 16
  - 2.4.2 Protocol Layering . . . 17
  - 2.4.3 Protocol Composition . . . 18
- 2.5 Models at Runtime . . . 18
- 2.6 Requirements Elicitation . . . 21
  - 2.6.1 KAOS . . . 21
  - 2.6.2 The i^* Framework . . . 22
- 2.7 Multiagent Systems . . . 22
  - 2.7.1 Multiagent Methodologies . . . 23
- 2.8 Linear Temporal Logic (LTL) . . . 25

2.9 Logic Programming . . . 26
2.10 SAT Solvers and Zchaff . . . 27
2.11 Background Summary . . . 27

3 Framework for Interacting Agents (FIA) Definition 28
3.1 Model Overview . . . 28
3.1.1 Agents . . . 30
3.1.2 Goals . . . 31
3.1.3 Actions . . . 32
3.1.4 Atomic Actions . . . 34
3.1.5 Interactions . . . 35
3.1.6 Interaction Role . . . 37
3.1.7 Protocols . . . 39
3.1.8 Protocol Role . . . 41
3.2 Bindings . . . 44
3.3 MultiAgent Interaction Model (MAIM) Assumptions . . . 47
3.4 LTL Logic Conversion . . . 48
3.4.1 Preconditions . . . 49
3.4.2 Postcondition . . . 50
3.5 Interaction Model Summary . . . 52

4 FIA Algorithms 53
4.1 Introduction . . . 53
4.2 AutoBinding System Overview . . . 55
4.2.1 Example System Details . . . 56
4.3 Automatic binding generation . . . 59
4.3.1 Binding Algorithm . . . 59
4.3.2 Binding Functions . . . 64
4.4 Provability vs. Viability . . . 67
4.4.1 Proof Mode . . . 67
4.4.2 Viable Mode . . . 67
4.4.3 Differences . . . 68
4.5 Binding Goals, Interactions and Protocols . . . 69
4.6 Substitutions and Proofs . . . 70
4.7 Conclusion . . . 74

5 FIA Demonstration 76
5.1 Methodology . . . 76
5.1.1 Methodology Requirements . . . 76
5.1.2 Physical Demonstration System . . . 77
5.1.3 Physical Demonstration Results . . . 79
5.2 Simulated Demonstration System . . . 83
5.2.1 Simulation Algorithm . . . 84

5.2.2 Simulated System . . . 85
5.2.3 Negotiation Failure . . . 87
5.2.4 Simulating Systems without the Interaction Framework . . . 89
5.2.5 Simulating Results using FIA . . . 91
5.3 Demonstration Conclusion . . . 92

6 Related Work **93**
6.1 Models at Runtime . . . 93
6.1.1 Static Models . . . 94
6.1.2 Self-Adaptive Models . . . 94
6.1.3 Goal Models at Runtime . . . 95
6.2 Protocol Reasoning Methods . . . 96
6.2.1 Commitments In Multiagent Systems . . . 96
6.2.2 Sharing State and Intent within Multiagent Systems . . . 98
6.3 Multiagent System Protocols from Specification . . . 100
6.4 Multiagent Systems Languages . . . 101
6.5 Verifying Agent Behavior . . . 102
6.6 Fault Tolerant Systems . . . 104
6.6.1 Fault-Tolerant Protocols . . . 105
6.6.2 Fault-Tolerance Framework . . . 106
6.6.3 Hermes . . . 106
6.7 ADL and Meta models . . . 107
6.7.1 Architecture Description Languages . . . 107
6.7.2 Meta-models . . . 108
6.8 Summary . . . 110

7 Conclusion **111**
7.1 Current State of Interactions in Multiagent Systems . . . 111
7.2 The Formal Interaction Framework . . . 112
7.3 Future work enabled by FIA . . . 115
7.4 FIA Limitations . . . 117

Bibliography **118**

A Additional Interaction Model Definitions **133**
A.1 Interaction Model and Definitions . . . 133
A.2 Agent Relations . . . 136
A.3 Interaction Role Relation & Restrictions . . . 137
A.4 Protocol Role Relations & Restrictions . . . 138
A.5 Variable Definitions . . . 140
A.6 Restrictions . . . 140
A.7 Consistency . . . 143

B Additional Functions **146**
B.1 LTL Example XSB Code . . . 146

C Binding Formal Definitions **148**
C.1 Variables and Expressions . . . 148
C.2 Variable Convenience Functions . . . 150
C.3 Binding formal definition . . . 152
C.3.1 Forming Bindings . . . 152
C.4 Action Bindings . . . 153
C.4.1 Action Input Bindings . . . 153
C.4.2 Goal Output Bindings . . . 156
C.4.3 Goal-Action Bindings . . . 157
C.5 Interaction Role Bindings . . . 158
C.5.1 InteractionRole Input Bindings . . . 158
C.5.2 Goal-Interaction Role Output Bindings . . . 159
C.5.3 Goal-Interaction Role Bindings . . . 160
C.6 Interaction Protocol Bindings . . . 161
C.6.1 Interaction Protocol Input Bindings . . . 161
C.6.2 Protocol Interaction Output Bindings . . . 161
C.7 Binding Proofs . . . 162
C.7.1 Binding shorthand notation . . . 166

D Results **167**
D.1 Auction Example Verification . . . 167

List of Figures

2.1 Operational Semantics of Goal Models Abstract Architecture 20

3.1 Interaction Model (MAIM) high level view 29
3.2 Worker Agent Specification 30
3.3 Example Agent and Goal Specification 32
3.4 Example Agent,Goal & Action Specification 34
3.5 Example Agent,Goal Action & Interaction Specification 36
3.6 Example Agent,Goal, Interaction & Interaction Role Specification 38
3.7 GPS Move Protocol & Protocol Role Specification 40
3.8 Full Example Specification 42
3.9 Required Action/Interaction Level Bindings 46
3.10 Required Role Level Bindings 46

4.1 Binding Types 55
4.2 Auction High-level Example 57
4.3 Auction Example Specification 60
4.4 Required Goal-Interaction Bindings 61
4.5 Required Interaction-Protocol Bindings 61

5.1 Demonstration Configuration 77
5.2 Android Screen Shot 79
5.3 Physical Demonstration Configuration Interaction Model 80
5.4 Simulation Setup 85
5.5 No Framework Simulation Results 89
5.6 Framework Simulation Results 91

6.1 Event Calculus Model[99] 96

A.1 Consistent Example 144

C.1 Move Specification with Types 149

[]

List of Tables

List of Figures ix

3.1 LTL operators . . . 49
3.2 LTL Precondition Derivation . . . 50
3.3 Postcondition LTL conversion Table . . . 50

4.1 Input Mapping . . . 62
4.2 Combination Example . . . 63
4.3 Valid Input Bindings . . . 63
4.4 Valid Output Bindings . . . 64
4.5 Seller Agent Variables . . . 64
4.6 Sell Item Goal Variables . . . 66
4.7 Goal-Knowledge-Interaction Input Function . . . 71
4.8 Interaction-Protocol Input Function . . . 71
4.9 Substitution Table . . . 73

5.1 Interaction Success without Capability Failure . . . 86
5.2 Valid Tuples For PC Agent when Tablet Camera fails . . . 87
5.3 Valid Tuples For Mobile Agent 1 when Tablet Camera fails . . . 87
5.4 Partial Success Situation . . . 90

C.1 Example Expression . . . 162
C.2 Example Binding . . . 162
C.3 Example Expression Input Binding Substitution . . . 165
C.4 Example Expression Input and Output Binding Substitution . . . 165

[]

List of Restrictions

1 Restriction (Protocol implies Protocol Roles) . . . 48
2 Restriction (Interaction precondition implies Interaction Role precondition) . 48
3 Restriction (Interaction postcondition implies Interaction Role postcondition) 48

4 Restriction (Type Uniqueness) . . . 134
5 Restriction (Variable Types) . . . 135
6 Restriction (One Agent Per Goal) . . . 137
7 Restriction (Role Played By Single Agent) . . . 137
8 Restriction (Non-empty Interaction Role Composition) . . . 137
9 Restriction (Interaction Role Range min $\leq$ max) . . . 138
10 Restriction (Interaction Role Composition Uniqueness) . . . 138
11 Restriction (Non-empty Protocol Role Composition) . . . 138
12 Restriction (Protocol Role Range min $\leq$ max) . . . 139
13 Restriction (Protocol Role Composition Uniqueness) . . . 139
14 Restriction (Action achieves Goal) . . . 141
15 Restriction (Interaction Role Composition Parameters) . . . 141
16 Restriction (Protocol Role Composition Parameters) . . . 142
17 Restriction (Protocol and Protocol Role Imply Interaction and Interaction Role) . . . 142
18 Restriction (Implements implies valid binding) . . . 142
19 Restriction (Protocol Role for each Interaction Role) . . . 143
20 Restriction (Interaction Role for each Protocol Role) . . . 143
21 Restriction (Binding Type) . . . 152
22 Restriction (Binding Partial Function) . . . 152
23 Restriction (A-K is Binding) . . . 153
24 Restriction (K-G is Binding) . . . 154
25 Restriction (A-K-G is a Binding) . . . 155
26 Restriction (A-K Composition) . . . 155
27 Restriction (G-K is Binding) . . . 156
28 Restriction (K-A is Binding) . . . 156
29 Restriction (G-A is Binding) . . . 157
30 Restriction (Goal Action Composition) . . . 157
31 Restriction (IR-K is Binding) . . . 158
32 Restriction (K-G is Binding) . . . 159
33 Restriction (IR-K-G is Binding) . . . 159
34 Restriction (IR-K-G Composition) . . . 159
35 Restriction (G-K is Binding) . . . 159

36 Restriction (K-IR is Binding) 160
37 Restriction (G-IR is Binding) 160
38 Restriction (G-IR Composition) 160
39 Restriction (P-I is Binding) 161
40 Restriction (I-P is Binding) 162
41 Restriction (Input Binding) 163
42 Restriction (Output Binding) 163

List of Definitions

1 Thesis . . . 5

1 Definition (Agent Attributes) . . . 30
2 Definition (Goal Attributes) . . . 31
3 Definition (Action Attributes) . . . 33
4 Definition (Interaction Role Attributes) . . . 37
5 Definition (Protocol Attributes) . . . 39
6 Definition (Protocol Role Attributes) . . . 43

7 Definition (Entity Definition) . . . 53
8 Definition (Interaction Tuple) . . . 54
9 Definition (Variable Pair) . . . 54
10 Definition (Binding Relation) . . . 54
11 Definition (Partial Interaction Tuple) . . . 62
12 Definition (Interaction-Protocol Tuple) . . . 70
13 Definition (Type System) . . . 133
14 Definition (Type Attribute) . . . 133
15 Definition (Variable) . . . 134
16 Definition (Variable Attributes) . . . 134
17 Definition (Specification Language) . . . 135
18 Definition (Specification Language Symbols) . . . 136
19 Definition (Expression) . . . 136
20 Definition (Variables Fuction) . . . 150
21 Definition (Fresh Variables) . . . 164
22 Definition (Input Substitution) . . . 164
23 Definition (Output Substitution) . . . 164
24 Definition (Directed Binding) . . . 166
25 Definition (Binding Set with Expression) . . . 166

List of Relations

1 Relation (Action Goal) . . . 34
2 Relation (Agent Action) . . . 34
3 Relation (Interaction Role Composition) . . . 37
4 Relation (Protocol implements Interaction) . . . 41
5 Relation (Protocol Role Composition) . . . 43

6 Relation (Type Variable) . . . 135
7 Relation (Agent Knowledge) . . . 136
8 Relation (Agent Goal) . . . 137
9 Relation (Interaction Role Range) . . . 138
10 Relation (Protocol Role Range) . . . 139
11 Relation (Protocol Role implements Interaction Role) . . . 140
12 Relation (Free Variable) . . . 150
13 Relation (Binding Definition) . . . 152
14 Relation (A-K Relation) . . . 153
15 Relation (K-G Relation) . . . 154
16 Relation (A-K-G Relation) . . . 155
17 Relation (G-K) . . . 156
18 Relation (K-A) . . . 156
19 Relation (G-A) . . . 157
20 Relation (Goal Action Agent Bindings Tuple) . . . 158
21 Relation (IR-K) . . . 158
22 Relation (K-G) . . . 159
23 Relation (IR-K-G) . . . 159
24 Relation (G-K) . . . 159
25 Relation (K-IR) . . . 160
26 Relation (G-IR) . . . 160
27 Relation (Goal Interaction Role Bindings) . . . 160
28 Relation (P-I) . . . 161
29 Relation (I-P) . . . 161

Dedication

I dedicate this thesis to my family and friends, especially my wife Beth. Thanks also to my children Addie and Keagan. They enjoyed my drafts for the scratch paper they produced. I would like to thank my mother and father for raising me right. They put up with my long quest to become a teacher. I would also like to thank John and Laura Homer. They have been the best of friends.

Chapter 1

Introduction

This chapter introduces the reader to the current state-of-the-art in multiagent systems and it shows that the lack of a formal model for interactions is a shortfall of current multiagent system methodologies. The chapter describes the creation of Framework for Interacting Agents (FIA) and the MultiAgent Interaction Model (MAIM) used in FIA. Multiagent systems designed using FIA will be more flexible and robust in the advent of failure. FIA allows agents to dynamically select interactions and protocols that satisfy the agent's goals.

1.1 Motivation

HUMANS have been interacting with one another for millennia, through wall paintings, smoke signals, oral communication and the written word. Only recently have humans been able to communicate across vast distances quickly by using telegraphs, telephones, and computer networks. Formal definition of communication protocols[100] and procedures have come into existence only in the last two centuries. Many of the advances in telecommunication and computer science have revolved around transferring and processing information within closed systems such as telephone or computer networks[104]. This focus on closed systems has led to information silos, where it is difficult for the information to migrate from one silo to another[103]. In the past decade computer system design has shifted from closed systems to more open and interconnected systems.

Multiagent systems are one approach to developing such interconnected systems. The

multiagent systems paradigm has evolved significantly in the past decade. Multiagent systems began as a distributed connected system, designed as a simple extension of a computer thread. The definition of multiagent systems has expanded to include the autonomy[121], mobility and physicality of agents[19]. Multiagent systems are being used in a variety of applications. Multiagent systems are being used for temperature control[59], remote surveillance[95] and cinematic animation[1] as well as many other applications. The first multiagent systems were developed using ad-hoc approaches with no framework support. Early designers focused on developing working systems and therefore maintenance and software reuse were not considered. As computer science and multiagent systems have evolved, software designers have moved from ad-hoc to formal design processes and the automated generation of implementations from formal designs. This evolution is evident in the proliferation of multiagent system frameworks such as KAOS[39], GAIA[124], O-MASE[52], Tropos[27] and Prometheus[86]. Designers elicit both formal and informal requirements for the system. These requirements are then used to create models and eventually an implementation of the system.

Most multiagent methodologies use the concepts of agents, goals, roles and capabilities. *Agents* have the ability to sense and affect their environment, and communicate with other agents within the system[123]. Agents can be autonomous or semi-autonomous and they can react to changes in their environment. *Goals* describe desired states of agents or desired states of the system[123]. Goals allow an agent to interact with its environment in a proactive manner. For example, an agent may have a goal to minimize battery use at night. Some models allow the multiagent system to define avoidance goals. Avoidance goals are states that the agents and the system attempt to avoid, for example ensuring that an atomic bomb does not leave a secured facility. Agents may have to perform a series of actions to ensure that an avoidance goal is always satisfied.

Roles define the rights and obligations of an agent playing the role[52]. Roles are an abstraction that allow system designers to compartmentalize aspects of a system. The role abstraction allows designers to reason about the system at a higher level, where the details

of the role are not relevant. Roles within multiagent systems are parallel to roles in the real world. For example, a person may play the role of father, student and coworker. The rights for each respective role include the right to raise their child as they see fit, the right to private academic records and the right to get paid for work performed. Responsibilities may include responsibility for a child's actions, doing one's own work, and acting in an ethical manner. Within a multiagent system it is possible for an agent to play multiple roles, such as a searcher role and a relay role. The agent playing the searcher role may be responsible for searching an area. The role may give the agent the right to be in the search area. The relay role may make the agent responsible for relaying messages that the agent receives.

Capabilities capture the real world abilities possessed by agents[121]. Capabilities are generally realized through sensors and actuators. Sensors allow an agent to observe aspects of the environment. Actuators allow an agent to modify the environment. Example capabilities include motors, pneumatics, radars, Radio Frequency transceivers, GPS's, accelerometers, gyroscopes and many more. Capabilities may also include processing capabilities such as a graphics processor or the ability to reason.

The multiagent paradigm allows and encourages the use of decentralized information[125]. Information may not be consistent across the system due to bandwidth, logistical and infrastructure restrictions. For example, agents may have a map of the environment and each agent may update the map with new information. If that map is not centrally managed, each agent can have a different version of the map, which is based on their own perceptions.

In addition to information fragmentation, many multiagent systems are designed to work with heterogeneous agents, where the capabilities of the agents differ. Some agents may be desktop computers connected to a wired network, while others may be battery powered, running a minimal operating system connected to a wireless network. Multiagent systems should incorporate the unique abilities of each of these agents in a robust and flexible manner. The introduction of mobile agents allows multiagent systems to run in a variety of environments. Due to these variable environmental conditions, multiagent systems may

be more prone to failure than other computer systems. For example, an agent running in a desert on a wireless network connected to a battery has multiple avenues of failure. Thus multiagent systems should be designed to operate in failure prone environments[126].

A failure-prone environment can make the process of designing communication protocols more difficult. Multiagent Systems typically define communication protocols in a formal manner. However, these systems often do not allow protocols to be chosen dynamically at runtime. Current state-of-the-art multiagent systems predefine agent interactions and protocols. For example, the Prometheus methodology defines interactions in terms of Interaction Diagrams (AUML Sequence Diagrams)[86]. Their sequence diagrams are designed for specific scenarios of the system. This type of design will fail when the system encounters an unexpected scenario. Likewise, the GAIA methodology defines protocols and an Interaction Model that leverages those protocols. Protocols are defined as an "institutionalized pattern of interaction" that can be reused within the system[124]. GAIA also defines a Service Model that leverages the protocols. These services are predefined for each agent in the system and they based on the roles that the agent is able to play.

The following quote from Marin *et al.* accurately states a major shortfall in current multiagent systems.

> *"A priori, the main motivation for multi-agent systems lies in the distributed nature of information, resources and action. It seems also intuitive that one of the fundamental issues of distributed computer systems is the possibility of host or network failures. However, it is to be noticed that most of the current distributed multi-agent platforms and applications do not yet address, in a systematic way, this possibility of failures[75]."*

In the Prometheus methodology the scenario diagrams handle failures. A designer has to define every failure scenario that an agent is likely to encounter. The GAIA methodology defines that the service model must include adaptive services. Neither of the above methodologies provides an explicit mechanism to deal with an interaction failure in a robust manner.

The lack of a mechanism in the methodology implies that the designers will (1) create a rigid interaction model using the methodology or (2) they will provide their own ad-hoc failure mechanism. Neither of these options provides a truly flexible mechanism for dealing with failure. Additionally, having predetermined protocols and interactions goes against a central tenet of distributed Multiagent systems: the ability to adapt to failure[63,66]. Multiagent systems need an explicit mechanism that provides the ability to adapt to failure[75]. This mechanism can be implemented by designing a formal framework and providing logic through a formal reasoning process. The formality allows researchers to prove the soundness of the results. The framework should define multiagent interactions by formally defining the relations between the agents, interactions, and protocols.

1.2 Thesis Statement

Thesis 1

> *Using a formal interaction framework to design and implement multiagent systems can result in systems that have the ability to adapt in the event of communication and capability failures.*

A formal interaction framework should provide the following: (I) Formal Model, (II) Design Tools such as Methods, Techniques and Analysis and (III) Runtime Support. Providing these three items should yield a framework that allows agents to adapt to failures.

A formal interaction framework must first and foremost provide a formal interaction model along with the semantics of the interaction model. Designers can model multiagent systems using the interaction model. The interaction model should have well understood semantics, such that a multiagent system designer can create multiagent systems.

A formal interaction framework should support design analysis. The analysis should yield results that accurately represent the behavior of the multiagent system. The analysis should also be automated to avoid translation errors. The designer should be able to use the results of the analysis to determine if the interactions within the multiagent system work

correctly.

The final requirement for a formal interaction framework is that it should be explicitly incorporated into the multiagent system at runtime. Runtime incorporation eliminates the need for a separate compilation step to convert the model into actual software. In addition runtime incorporation eliminates the need for a separate algorithm for adapting to failure. When a failure occurs, the agent simply leverages the same tools used in the design analysis to enumerate the interactions that satisfy the agent's requirements.

An interaction framework that provides formal models, design analysis, and runtime support should yield a multiagent system that can adapt to failure. Creating an interaction framework that meets these requirements would be a major benefit to the multiagent system community, as it would create more flexible and robust multiagent systems.

1.3 Research Approach

The following research tasks were executed to create a formal interaction framework that provides multiagent systems with the ability to adapt to failures.

I Formally define the components of MAIM

II Define how the components of MAIM interact with one another.

III Define reasoning for FIA that can predict system behavior.

IV Demonstrate that systems designed with FIA can adapt to communication and capability failures.

Task I requires that each entity within MAIM be intuitively and formally defined. MAIM includes the basic multiagent systems concepts of Agents and Goals as well as Interactions and Protocols.

Task II defines how the model entities interact with one another. Specifically the required relationships must be defined in a formal manner. The formal definition allows MAIM to be reasoned over by a human or an automated system.

Task III defines the algorithms that FIA uses to reason about a MAIM model. The algorithms should be usable at design time in order to aid designers in the creation of model. Feedback allows the designer to find design flaws earlier in the design process, which can help reduce the cost of those errors.

Finally, in Task IV, a real world application has been designed and implemented using FIA. This application shows that a system that uses FIA is more flexible and robust than a system designed without the framework.

1.4 Scope

Selecting the optimal protocol is orthogonal to this proposal. An optimal protocol selection model would give the agents the ability to leverage other properties of the protocols. These properties include bandwidth, power usage and latency. The optimal protocol model would allow the system to be optimal in one or more of these facets. These choices are similar to those in the saying " Fast, Good, Cheap; pick any two". FIA assumes that agents can use their autonomy to choose the optimal protocol when they have a choice.

This thesis also does not address tool support for FIA. Tool support helps designer when they are defining interactions and protocol specifications. agentTool is a tool that provides a graphical interface for designing multiagent systems. agentTool uses the O-MaSE Methodology and the GMoDS goal model. Integration with agentTool would aid system designers in creating interaction models. Tool support would be a future research endeavor.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 provides background information that will help the reader understand some of the specific terminology used in multiagent systems, protocols, interactions and model checking. It also provides an overview of some of the popular multiagent system frameworks that have been created. Chapter 3 defines FIA.

Many of the low level details of FIA are provided in Appendix A. Chapter 4 defines the FIA reasoning algorithms. These algorithms can be used to verify that a system can meet its specifications. Additionally, the actual agents can use the algorithms to find interactions that will achieve their goals. Chapter 5 integrates FIA into a multiagent system. This system is able to adapt to failures and it is more robust than a system designed without the framework. Chapter 6 provides the reader with a cross-section of work that is related to FIA and Interactions within multiagent systems. Chapter 7 gives a summary of this thesis and it highlights research endeavors enabled by this thesis.

Chapter 2

Background

This chapter provides the reader with a background on a broad range of topics related to this thesis. Section 2.1 surveys some of the current philosophical views that are held about knowledge. Knowledge, information and data can be transferred via interaction, which is explored in Section 2.2. Section 2.3 gives the reader an overview of a few software design techniques used to ensure that software works as specified. The advantages and disadvantages of each technique are given. Protocols are the conduit though which interaction takes place, and a brief overview of Protocols and Protocol design is provided in Section 2.4. A formal model that can be used at both design time and runtime is a powerful concept. Section 2.5 enumerates some formal multiagent system models that can be used during both processes. Section 2.6 shows how software designers capture requirements and how a system can be implemented from those requirements Section 2.7 provides background on current Multiagent System methodologies. A Framework designed for Multiagent Systems requires an understanding of the concepts of agents, goals, roles and capabilities. The last two sections give the reader an overview of some of the tools used in FIA. Section 2.9 gives background in to logic-based programming and the XSB programming language. Finally, Section 2.10 gives a brief background on SAT solvers.

2.1 Knowledge

A debate has been going on since the Greeks first introduced logical reasoning. Some of the major questions have been "What is knowledge?" and "How is knowledge different than information?" [15] The quote below exemplifies one view of their difference.

> *A commonly held view with sundry minor variants is that data is raw numbers and facts, information is processed data, and knowledge is authenticated information.* [15]

This view is intuitive in that knowledge arises from information which in turn arises from raw data. An opposing philosophical view is that humans innately have knowledge. This view also states that knowledge must exist before data and information can be created. Humans use their knowledge to hypothesize about information that they believe should exist. Humans try and collect data that either proves or disproves their hypothesis [110]. The collected data can then be reformulated into new information and new knowledge.

In addition to the ambiguity concerning knowledge, there is also ambiguity as to the definition of information and knowledge. Some researchers believe that there are two different types of knowledge, *explicit* and *tacit* [83]. *Explicit knowledge* is knowledge that can be stored in a regular form, using a defined structure that is bound by a set of rules. *Tacit knowledge* is knowledge that is specific to a person or group of people that cannot be disseminated easily; however, tacit knowledge is believed to be the key to innovative and competitive companies.

Contrary to the belief that explicit knowledge exists, there is Stenmark [103] who argues that knowledge can *only* exist inside the human mind. He believes that the only thing an information system can process is information. Stenmark does acknowledge that knowledge and information are intertwined, such that one cannot exist without the other.

FIA addresses the fact that agents can have knowledge, but it does not attempt to address these philosophical questions about how knowledge can be created or derived within an

agent. Instead those philosophical questions should be addressed within the scope of agent autonomy.

2.2 Interaction

Knowledge, information and data are shared by humans through a process called communication. Linguists define several key components that are required for communication to take place. There must be (1) a sender that sends a message, (2) a receiver that receives a message, (3) a message channel through which the message is sent, (4) a message format that defines the arrangement of the message and (5) a context in which the message appears. If these elements are present then the sender can perform a communicative act with the receiver[61]. A communicative act is an action that transfers information from the sender to the receiver.

Broadcasting is one of the oldest forms of communication. An example of a simple broadcast system is a radio station and receiver. The radio station broadcasts the signal to a wide area. The station sends out a signal to a general area, but the system is ignorant of where the radio receivers are located. A reactive system is a system where the receiver can perform an action to react to a message that was sent. However, the receiver does not directly communicate through a specific protocol back to the transmitter. Broadcast radio can be used to create a reactive system. This is the case for Weather Alerts that are broadcast by radio stations through the Emergency Alert System[47]. Radio stations can broadcast that a tornado is heading toward a specific community. The community does not directly respond back to the radio station, but relays the information to its citizens to take cover from the storm.

Interaction is a sequence of communicative acts where each party understands the meaning of each communicative act. A communicative act occurs within a communication medium. A *communication medium* is the "conduit" through which a message can be transmitted. A variety of communication mediums exist, including electromagnetic radia-

tion (radio), mechanical waves (audio) and electrical signals (coaxial cable, ethernet). Each medium has specific advantages and disadvantages. For example, most humans have the ability to create and detect sound waves, thus humans can communicate in this manner rather naturally. Sound does not travel easily across vast distances and thus radio communication is better for telecommunication.

In an interaction, the understanding and the meaning of communicative acts are based on the context of the communicative act. A pair of cars pulling up to a stop sign is an example of an interaction. Each driver assumes that the other driver knows the rules of the road. By making this assumption, each driver knows what each of them should do. For example, if both cars pull up to a stop sign at the same time, the driver to the right has the right of way. However if a situation arises where the rules of the road are ambiguous, then the drivers may have use hand signals to interact with one another. Each action (hand signal) carries an implied meaning to the participants. An interaction can allow different protocols to be used to achieve the same result. For example when a human is having a conversation with another human, the protocol for conversation can be Voice Over IP (VOIP), email, instant message, facebook[8] or twitter[13]. Each of these protocols uses the Internet as the communication medium. The interaction may be the same, but the protocol can vary. Each of the above protocols has different rules that determine how the protocol works. For example, in email the delay between messages may be days or weeks, but in vocal communication the delay is usually measured in seconds. It is considered rude to reply to a question weeks later when talking to a person face to face.

When communication theorists first started formulating the definition of interactions, there were several competing models. There are three classical models: Fayol, Weber and Taylor. Fayol's model is summarized by the following quote.

> "An effective organization is highly structured, and each individual knows where he or she fits. Clear structures facilitate the function of organization and clear rules deal with these structures" [81].

This theory describes the functions of an organization, but leaves out some of the human elements, such as interpersonal communication. This theory gives us the rigid structure required by most organizations and protocol systems.

Weber defined bureaucracy and the ideal theory of how a bureaucracy should work[118]. Bureaucracies provide a set of rules that govern the decisions of an organization, which is quite different from the arbitrary authoritarian style that preceded bureaucracies. This theory describes bureaucratic hierarchies that only allow information to flow through approved channels. The decisions the bureaucracy makes must be centralized at the top levels of the hierarchy for the bureaucracy to correctly operate.

Taylor's theory involves the scientific study of organizations[107]. Taylor's theory states that each job in an organization has a "one best way", which can be observed by scientific study of that job. It also states that each worker in an organization has a job that suits them best. The best workers should be put in their best suited job to increase the efficiency of the organization and the efficiency of the individual worker. The final element of the theory is that management and workers should have a strict division of labor. The organization's management should perform the scientific study and management operations while the workers focus on the hard labor.

These three models treat humans as machines and try to reason about them as such. These theories lead to the common practices of division of labor, strict protocols between different levels in an organization, and the practice of striving to make humans more efficient at their jobs. This thesis is based on these classical models.

2.3 Computer Program Verification

Interactions allow humans to communicate knowledge, information and data among one another. Humans have also developed computer systems to create, derive and store information and data. Computer science is the study of how to create computer systems in an efficient, fast and economical manner. Creating computer programs that are correct is a

difficult task (anyone running a computer can attest to this fact). Thus computer scientists and practitioners have developed methods to aid them in creating correct computer systems. Two such methods are unit testing and model checking.

2.3.1 Unit Testing

Unit testing is a method of verifying that a computer system works as specified[31]. Most programs have several functions. Unit tests verify that each component produces the correct output when given a specific input. For example if the component has a square function, then giving the function the input of 4 should return the result of 16. Unit tests can then be run each time the component is modified to ensure that the output of the component is still correct. If the test fails, then either the programmer has made an error in the test or in the modified component. Unit tests give component designers confidence that a component works correctly, but they do not prove that a component is correct. If the designer forgets about a specific case, then the unit test may pass, but the component may return an incorrect result. For example, a programmer may forget to write a test that checks for *null*. If the component is called with the value of *null*, the component may crash. There are several tools that programmers can use to create unit tests for their component. The first, and simplest, is ad-hoc unit testing. A programmer writes a program that tests each function and ensures that the function returns the correct result. The second method is to use a framework that is made for unit tests. There are many frameworks available, including JUnit, NUnit[3] and many others. JUnit[10] is a testing framework that allows designers to specify testing methods. Designers can also specify properties that must hold for methods that are being tested. For example, a programmer may use the *assertTrue()* method to specify that something in their code must be true when tested. When the programmer chooses to run the unit tests, the JUnit testing framework automatically verifies the validity of each test case.

2.3.2 Model Checking

As software has proliferated in the past decade, the need to verify that the software works correctly has become of greater importance. There are many examples where software did not work properly including the Therac-25[18], The Chinook Helicopter Disaster[6] , and Toyota Prius braking system[12]. Software designers are beginning to move from a testing paradigm to a verification paradigm. Model checking is a process that takes a model of a system and verifies that it satisfies a specification for that model. There are a wide variety of model checkers that have been developed including SPIN[22], Bogor[96], and Java Pathfinder[9] to name a few. If a model can be verified by a model checker, the results are more conclusive than if the model is only verified via unit testing. A unit test will check a program with a *small number* of inputs, whereas a model checker can check a model of the program with *all possible* inputs. When a model checker verifies a model the designer is assured that the model meets its specification. Unfortunately, the cost of running a program with all possible inputs can be quite high. Model checkers are limited by the size of the input and by the time allotted to verify the model. As the size of input increases, the time required to check the model increases exponentially[78]. This exponential time increase makes model checking infeasible for some models. Thus a programmer must decide whether model checking is required or if unit testing is sufficient.

2.4 Protocols

Communication requires that both the sender and the receiver adhere to a shared protocol. Protocols have been around since Claude Chappe[28] created the optical telegraph in 1792. Over time protocols have been applied to radio broadcast, television broadcast, telephone networks and computer based network systems. In each of these systems, the protocol starts as a fixed set of rules that tend to evolve over time. For example in the telegraph the "stop bit"[5] came to be used to end a sentence[2], since the "stop bit" was cheaper to send than a period. Telegraph readers learned to interpret the sentences correctly. Telephone systems

started as a system to send voice over the telephone network. Modems[106] made use of the telephone network and provided network connectivity for computer systems. In these two cases, humans have adapted the original protocol in order to meet different needs. This adaptation shows that while protocols are designed to be rigid, they can be adapted to meet the needs of the system.

Humans have an innate ability to learn and adjust to changes in their environment. However, this task is more difficult for computers to perform. There are several cases where using protocols in the wrong context led to undesired outcomes. One of the most famous examples was the Enigma machine used by the Nazi's in World War II. The encryption protocol was designed to be secure if the keys were not reused. However, the operators of the Enigma machine became lazy and reused the keys, thus decreasing the security of the encryption protocol[87]. If a protocol is modified, then the properties of the redesigned protocol must be verified for correctness.

2.4.1 Verifying Protocols

Protocol design has evolved into a process that involves at least three steps. The first step is to specify what the protocol should do. The second step is to model the protocol at a high level, abstracting some of the details. The specifications given in the first step are used to create formal properties that must hold for the model. The third step is to implement the protocol given the specification and the model.

The protocol model can involve many states in many different processes. The full complexity of a typical protocol model is quite high, and thus the protocol model must be verified through a proof technique. The preferred verification technique is model checking. A model checker has the ability to run a protocol through every possible state to verify that the protocol operates as desired. The first, and most widely used, protocol model checker was SPIN[22]. SPIN was developed at Bell Labs for verifying the correctness of telecommunications protocols. SPIN has been used to check flood control systems[62], call processing

systems[58] and satellite software[57]. Models and properties are specified in the PROMELA[102] modeling language and then properties of the model are verified by SPIN.

2.4.2 Protocol Layering

The most widely used protocol in the world is the Transmission Control Protocol/Internet Protocol (TCP/IP)[90]. The TCP/IP stack is used to send data reliably from one computer to another computer. The Internet Protocol (IP) is used to route data across a network, regardless of the underling physical hardware. The Transmission Protocol (TCP) divides the data into segments that are delivered by the IP, which in turn transmits the data across the network. TCP handles transmission errors and retransmits data that is lost, dropped or corrupted. TCP/IP is the underlying protocol on which the Internet is based, and services such as HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP) and Secure Shell (SSH) all run on top of TCP/IP. The TCP/IP protocol is hierarchically decomposed to allow for modular design. The standardized Open Systems Interconnection (OSI) model[94] defines a seven layer protocol stack. Each layer has the responsibility of providing services to the layer above it by using the services of the layer below it. This decomposition allows the designers at each layer to assume that the underlying layers work correctly. For example, the Layer 2 provides point-to-point reliable communication. Examples of layer 2 protocols include IEEE 802.11 Wireless and IEEE 802.3 Ethernet. The Layer 2 relies upon Layer 1 (physical) to convert electrical signals into bits. The Layer 2 is used by Layer 3 (TCP/IP) to provide communication between different networks. The layered approach has proven to be a good approach to designing network protocols.

The Horus model[112] provides a layering mechanism for protocol designers. The designers create a set of micro-protocols that can be arranged like Lego blocks in order to create a macro-protocol that is tailored to the requirements of the application. These requirements may include encryption, acknowledgment, or reliable transport. The protocols designed by such systems can be more efficient than monolithically-designed protocols.

2.4.3 Protocol Composition

Protocol layering and protocol composition are slightly different concepts. In protocol layering, protocols are arranged in a stack, whereas in protocol composition, protocols are connected in a graph like structure. The OSI model allows a protocol to make use of a single lower level protocol. Protocol composition allows protocols to make use of many other protocols[29]. For example, an encryption protocol may require a random number to be sent in a message. If a random number protocol exists, the encryption protocol could use it to acquire a random number. The random number protocol may be initiated with a completely different entity, and thus the protocols are not arranged in layers as prescribed by OSI.

Protocol composition has a major benefit in that by composing protocols, the designers can leverage the properties of the protocols used in the composition. Properties might include Quality of Service (QOS), reliable delivery and shared session keys[26]. By leveraging these properties the designer can create meta-protocols tailor made for a specific application. There are several frameworks that provide programmers with the ability to create these meta-protocols including Appia[80], Horus[112] and Maestro[26]. Protocol composition is a key component of designing large systems. Each of these protocol composition frameworks provides different benefits and drawbacks and application designers must choose the framework that best meets their needs. For example, a highly flexible system may require more configuration and it may have a performance penalty. An application that requires an efficient protocol may not be able to use a flexible framework like Appia, but could use a simpler framework like Horus.

2.5 Models at Runtime

As software has become more complex, there has been a push to develop systems that leverage formal models at both design time and runtime. Using models at runtime is such a prevalent topic that there are conferences dedicated to it[11]. In addition to being prevalent

in research, many software processes and architectures are based on this principle, including Rational Unified Process (RUP)[65], Domain Specific Modeling (DSM)[50], and UML's Model Driven Architecture (MDA)[79]. Systems using these designs typically take a model of the system and generate an implementation of the system directly from those high level models. This provides a direct correlation between the model and the implementation.

The above systems generate software that adheres to the model semantics. A major question that arises is "What happens when the requirements of the system change at runtime?" For a system designed with one of the above architectures, the system must be recompiled and redeployed. Runtime models excel when the requirements of the system change at runtime. The systems behavior can be modified by changing the model of the system. These changes can be done without having to recompile the entire system. One example of such a system is the Goal Model for Dynamic Systems (GMoDS). GMoDS is a dynamic goal model for computer systems[44]. The system is specified by a goal specification tree. There is a goal at the top, which is the overall goal of the system. The top level specification goal is decomposed into smaller child goals. To achieve a parent goal one of two things must happen, either (I) **all** the children must be achieved or (II) **one** of the children must be achieved. The former is an AND goal and the latter is an OR goal. Thus the goal tree is known as an AND/OR goal specification tree. GMoDS also provides a tree at runtime called an instance tree. This tree is created based on the specification tree. A GMoDS based system dynamically adds and removes instances of the specification goals to the instance tree at runtime. An example that illustrates the dynamic nature of the goal model is a search and rescue system. The search and rescue divides a particular area into a number of subareas that must be searched. As victims are located, new instance goals are created for the rescue of each victim. In addition to dynamic goal creation, the goal model defines a partial ordering on how agents can achieve goals. This ordering allows designers to specify that particular goals must be accomplished before other goals can be attempted. In the search and rescue example, the agents may wait until all the victims are located before

attempting to rescue any of the victims.

Winikoff *et al.*[113] abstractly defines what multiagent system goals are and how they can be used. Morandini *et al.*[82] extends the abstract goal model architecture to define semantics for an AND/OR decomposed goal tree. This model requires that leaf goals have a set of steps that an agent can directly follow (a plan). These plans can be executed by the agents to achieve those leaf goals.. A leaf goal is a goal that is not decomposed, and thus it has no children. The leaf goals are one of three types. *Perform goals* are goals that do not specify a particular state. *Achieve goals* are goals that do specify an achievement state. *Maintenance goals* attempt to maintain a particular state of the environment. Their abstract architecture defines the semantics of the high level parent goals (shown in Figure 2.1) as well as the semantics of the leaf goals. Their architecture defines that non-leaf goals are in one of the following states {*suspended (S), active-deliberate (AD), active-undefined (AU), active-success (AS), active-failure (AF)*}. The semantics of their abstract architecture provides a mechanism for goals to change state. These formal semantics allow the system to dynamically adapt to changes in the environment.

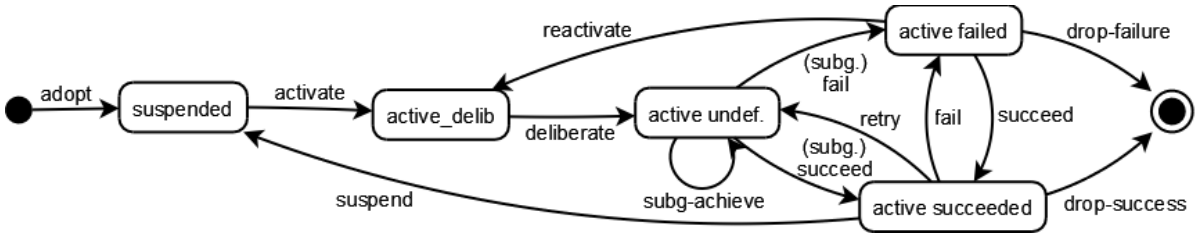


Figure 2.1: *Operational Semantics of Goal Models Abstract Architecture*

The Organization Model for Adaptive Computational Systems (OMACS) is a model designed to provide organizational support for multiagent systems[42]. This model defines that an Organization consists of Agents, Capabilities, Roles and Goals. In addition to these entities it also defines relations among the entities. For example the *possesses* relation defines

which capabilities an agent possesses. The Organizational model provides an Organizational Achievement Function (*oaf*) that quantifies how well an Organization achieves its goals. The OMACS model can be used at runtime to allow a system to adapt to failures in an agent. An example of an agent failure occurs when an agent loses a capability. If that agent was assigned to achieve a goal that is required by the system then the system cannot achieve its goal. An OMACS based system must then try and find a new set of assignments of agents to goals that will allow the system to achieve the goal of the system.

Models that are used at runtime are a new trend in computing that provide a direct relationship between the specification of a system and the implementation of that system. Any new framework should provide such a formal model and an automated mechanism to either generate a system or provide a runtime model.

2.6 Requirements Elicitation

The first step in designing a system is gathering or eliciting requirements from the organization requesting a new system. There are several tools, namely KOAS and i^*, that specialize in eliciting requirements for multiagent systems.

2.6.1 KAOS

The KAOS methodology is a goal-driven requirements elicitation environment[39]. KAOS provides a goal model that is AND/OR decomposed and it was one of the first methodologies to incorporate the AND/OR decomposition of a goal model. This decomposition allows a developer to iteratively refine the goal model. KAOS allows the specification of goals to include temporal and first order logic specifications. KAOS also included a graphical user interface (named GRAIL) to aid designers. The major pitfall of KAOS is the lack of a relationship between the specification of the system and its implementation. As seen in Section 2.5 integrating the specification into implementation has been considered a key feature in large systems. The GRAILS environment provides the ability to create specifications that

can be read by external tools. These external tools (they used a tool named DOORS[39]) allow the designer to reference the specification when implementing the system. The designer must take care to ensure that changes to the KAOS model are synchronized with the implementation of the system, otherwise the implementation will not accurately reflect the specification of the system.

2.6.2 The i^* Framework

The i^* modeling framework is intended for the early requirements phase of design[127]. Agents within the i^* framework have the ability to rely on other agents for services. This reliance is modeled by the Strategic Dependency model. The dependency model allows the system designer to elaborate Resource, Task and Goal dependencies that an agent may have with other agents in the system. A notable contribution to multiagent systems was the introduction of a softgoal. Softgoals are goals for which there is no specific task that can accomplish that goal. For example, a soft goal can be quality, efficiency or redundancy. Softgoals are most often qualitative and thus are hard to directly implement. In addition to the Strategic Dependency Model, i^* includes a Strategic Rational Model. This model allows a system designer to specify and view the rationale that an agent would use when making decisions on how to achieve a goal. The i^* framework intentionally allows the requirements to be specified in an informal manner and thus the properties cannot be systematically verified.

2.7 Multiagent Systems

The current state of computer programming is focused on allowing programmers to quickly create correct, maintainable and flexible programs. This shift is evident in the rapid adoption of programming languages that include runtime compilers and automatic memory management (Java, C#, Python, etc). The shift in the focus of programming has led computer scientists to create new methods, models and tools for creating computer systems. A programming paradigm is a collection of these methods and models to solve a particular type

of problem. A modern paradigm that has emerged is that of multiagent systems[119]. Multiagent systems are a relatively recent concept in the computing world (early 90's). The first multiagent systems were purely software based and started out as a collection of desktop machines connected to a wired network. Current multiagent systems have expanded to include agents that are battery powered and communicate via wireless protocols. Additionally, they can include a wide variety of sensors and actuators[27,97].

The multiagent system paradigm defines what an agent is and it provides a method to connect agents. Robotic multiagent systems tend to be deployed in environments where the likelihood of failure is higher than in normal computing environments[114]. This increased likelihood of failure has lead to multiagent system frameworks being developed that allow agents to adapt to failure and self organize[42].

2.7.1 Multiagent Methodologies

There are a variety of multiagent methodologies that are available for use in system design. Some of these methodologies include GAIA[124], Tropos[27], Prometheus[86], and O-MaSE[41]. Each one of these methodologies provides a design time model that allows the designer to specify desired behavior and/or restrict undesired behavior.

The GAIA Methodology

The Gaia methodology is *"intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed that it can be implemented directly"*[124]. Thus Gaia requires an additional tool or framework that can take the design documents and create the software that implements those design documents. Gaia produces a static organization structure that does not work well for large numbers of agents. Gaia also has no systematic process for detecting or dealing with agent failure. At the top level Gaia, decomposes the system in terms of a Role Model. Agents in the system play roles and those roles may require interacting with other agents. The Interaction Model for Gaia defines how the agents interact with one another. The Interaction Model is a high level design artifact.

Any changes to the Interaction Model should lead to changes in the models at the lower levels of the design. The integration of these changes may not be a trivial task, as other tools are required to create an implementation of the system.

Tropos

Tropos is multiagent system methodology that is formed around the *Belief,Desire and Intention* (BDI) software model[27]. The BDI psychological model attempts to emulate the human mental model[115]. The BDI software architecture adapts this mental model for designing agent based systems[93]. Tropos uses the i^* modeling framework for the requirement elicitation phase. Tropos has a formal meta-model and uses UML for diagram creation. Tropos includes a metamodel that includes Actors, Goals, Plans, Resources, Dependencies, Capabilities and Beliefs. Tropos leverages the concept of AND/OR decomposition from KOAS and the notion of softgoals from i^*. Tropos uses plan diagrams at execution and AUML (Agent UML) sequence diagrams to model communication. Tropos typically uses the JACK[30] platform for development, but the model allows any suitable platform to be used to create a system implementation.

Prometheus

The Prometheus Methodology is designed to be a practical methodology. The methodology allows designers to include a variety of materials such as UML sequence diagrams, interaction diagrams and use case diagrams. Interactions are *"derived from use case scenarios using a fairly mechanical process"*[86]. The major benefit of Prometheus is ease of use and the ability to reuse design documents. However, its ease of use and design reuse are also its biggest detriment. Prometheus lacks standardization and formalization, thus creating a cohesive system model can be daunting.

O-MaSE

The Organization-based MultiAgent System Engineering (O-MaSE)[41] methodology was created to design multiagent systems that are organization-based. O-MaSE reinvents the original MaSE methodology[43] for organization-based multiagent systems. O-MaSE incorporates an organizational model (specifically OMACS) that allows a multiagent system to adapt to failures by reorganizing at runtime. agentTool III[43] is a tool that can be used to design multiagent systems using the O-MaSE methodology. agentTool III allows a designer to specify the entities within the system (Agents, Roles, Goals, Capabilities) and the relationships among those entities (which agent has what goals) by using a graphical interface. O-MaSE can produce an OMACS compliant system, which allows it to adapt at runtime to changes in the capabilities of agents.

Multiagent Methodology Summary

These methodologies provide two major improvements over the ad-hoc design. The first major benefit is that these methodologies allow designers to think at an abstract level. This abstraction allows designers to reason about a particular component without being overwhelmed by the low level details. Decomposition also allows design teams to work on separate parts of the system simultaneously. The second major benefit is the ability to reuse components. Each methodology allows designers to reuse the design of an agent, goal or protocol. Software reuse can be a critical tool in a designers arsenal.

2.8 Linear Temporal Logic (LTL)

Linear Temporal Logic is a temporal based logic that allows a designer to specify properties about the current state or a future state[60]. LTL's temporal operators specify at what points in time the properties should hold. Some of the operators include `Global` ($\Box$), Eventually ($\Diamond$), Release ($\mathcal{R}$) and Until ($\mathcal{U}$). An example LTL property for a printer can be specified by the following: $\Box(busy(Printer) \vee free(Printer))$. This property specifies that

the printer is either busy or free. Another property could be that the printer is eventually free $\Diamond(free(Printer)$. The `Until` operator can be used to specify that the printer is busy until it is free *busy(Printer)* $\mathcal{U}$ *free(Printer)*.

In 1977 Amir Pnueli[89] proposed that LTL could be used to verify properties of programs. Since then, LTL has been used in the specification of properties of for a variety of modeling systems. Some examples include SPIN[22], PROMELA[102], Maude[46], and WSAT[51]. System designers can use LTL to specify invariants, safety properties or liveness properties. Invariants are properties that always hold. An example invariant is the global property $\Box($*busy(Printer)* $\vee\ free(Printer))$. Safety properties are used to specify that something bad never happens. Ensuring that deadlock never occurs is an example of a safety property. Deadlock occurs when the system reaches a state where it cannot proceed, and it typically occurs when processes are waiting for shared resources. An example deadlock would be $(\neg busy(Printer) \wedge \neg free(Printer))$. The deadlock avoidance property would be $\Box(\neg(\neg busy(Printer) \wedge \neg free(Printer)))$. Liveness properties are properties that state that something good eventually happens. An example liveness property is $\Diamond(free(Printer))$. This states that the printer is eventually free. LTL's expressive power has lead to it's widespread use in system modeling.

2.9 Logic Programming

Logic programming started in the 1970's. One of the first logic programming languages created was Prolog[34]. In a logic programming language, the designer defines a group of relations. A user can then query a logical program about the relations that have been defined. Logic programming has been used in artificial intelligence[16] as well as a variety of other fields including security verification[14], natural language processing[55] and theorem proving[105]. Prolog programs consist of terms. Some of the terms are called facts, which are assumed to be true. Additionally, there are rules that allow Prolog to make inferences based on rules and facts. For example a prolog program may have the following facts:

1. mammal(dolphin), 2. fish(nemo), and 3. mammal(clifford) . A programmer can query Prolog to find the mammals by issuing *mammal(X).* As a result, Prolog would list the values for X, which would yield X=$\{$*dolphin, clifford*$\}$. A rule for the example could be *animal(X) :- fish(X)*, which says that all fish are animals. Thus a user issuing the following query *animal(X)* would receive the result $X = \{nemo\}$.

The programming language XSB[4] is a version of Prolog that includes tabled resolution. *Tabled resolution* allows a program to compute an answer to a query and then store the query answer pair in a table. Then, when that query is issued again, the program can look up the value instead of recomputing that value. Tabled resolution increases the speed of an XSB program. XSB is open source and it was originally created at Stony Brook University. XSB is used in Section 3.4 for the derivation of facts from LTL logic.

2.10 SAT Solvers and Zchaff

A SAT solver is a program that can check whether or not a Boolean formula has a set of assignments to the Boolean variables such that the formula is equivalent to True. For example solving the Boolean formula for $A \vee B$ a value of True for A or B would yield a satisfying solution. However, the formula $A \wedge \neg A$ does not have any solution. SAT solving in general is a very difficult problem to solve (NP-Complete in the worst case). There are, however, SAT solvers that can solve certain problems in a reasonable amount of time. One such solver is the Zchaff[91] SAT solver.

2.11 Background Summary

This chapter has provided a brief overview of model checking, computer protocols, multiagent systems, logic programming, interactions and the study of knowledge. Each of these concepts are used in the definition of FIA, MAIM and the FIA reasoning algorithms defined in the next two chapters.

Chapter 3

FIA Definition

This chapter gives a formal definition for the MultiAgent Interaction Model (MAIM) and the Framework for Interacting Agents (FIA). MAIM defines each entity involved in FIA and the relations between those entities. MAIM also defines restrictions upon those entities and those restrictions define which configurations of the system are valid. The valid configurations can be reasoned over by FIA such that system behavior can be predicted.

3.1 Model Overview

FIA consists of three intertwined structures.

I Interaction Model (MAIM)

II Type System

III Binding System

Figure 3.1 gives a high level overview of MAIM. The central element of MAIM is the Agent. An Agent *has* a set of Goals that it is trying to *achieve*. Each Agent also has a set of Actions that it can *perform* in order to achieve those Goals. An Action can be one of two types, it can either be an Atomic Action or an Interaction. Interactions are composed of Interaction Roles. An Interaction Role is a set of responsibilities given to an agent while participating in an Interaction. Interactions are *implemented* by Protocols. A Protocol is composed of

Protocol Roles. Protocol Roles *implement* the responsibilities required by the Interaction Roles.

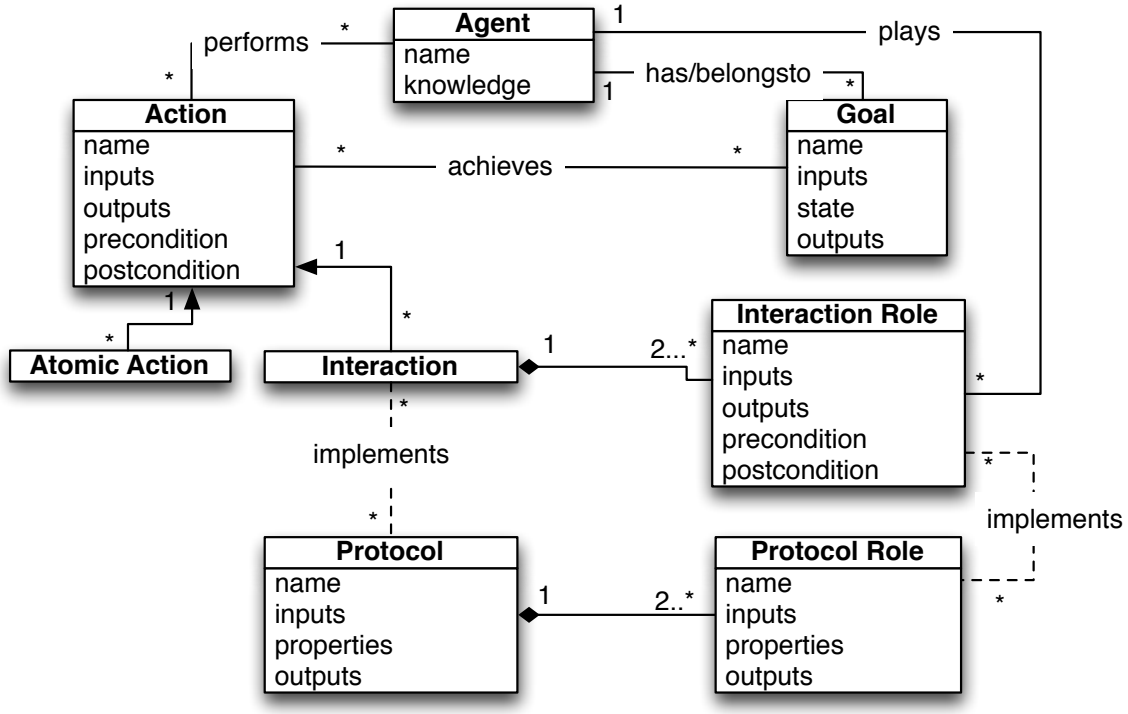


Figure 3.1: *Interaction Model (MAIM) high level view*

FIA also defines a Type System that consists of Types, Variables, a Specification Language and Expressions. The Type System provides a formal base for proving behavioral properties. The entities are specified using expressions that contain Variables. Each entity uses variables and expressions in a different manner. A Goal utilizes variables to state properties about the desired state that an Agent should strive to achieve. In an Action, the variables (inputs) are used to modify the behavior. The precondition specifies the Action's requirements and postcondition describe the effects of the Action. The formal definition of these concepts is located in Appendix A.

The binding system provides a mechanism that describes how information flows through MAIM. The binding system utilizes the Type System to define a formal channel through

which this information can flow. A Specification Language has been developed for specifying formal properties within FIA. This language is fully defined in Appendix A.

The next few sections define each of the various entities and how they are related. Definitions of the formal relations and the attributes of each entity will also be specified. Example entities are given to clarify the definitions and formalizations.

3.1.1 Agents

Agents are the central component in a multiagent system. "An *agent* is a computer system that is capable of independent action on behalf of its user or owner." [121] This description of an Agent provides a basic notion of an agent. MAIM provides the relations (as shown in Figure 3.1) between the agent and other entities within the model. Definition 1 states that an Agent has a *name* and some *knowledge*. The name allows agents within the system to identify each other. An Agent's Knowledge allows it to make decisions on current and future actions. The Knowledge also allows an Agent to select appropriate Interactions.

Definition 1 (*Agent Attributes*)

name	*String*
knowledge	*Set(Variable)*

An example Agent is shown in Figure 3.2. The Agent's name is "WorkerAgent". WorkerAgent has two pieces of knowledge. The first piece of knowledge is the Agent's location. The second piece is the current time.

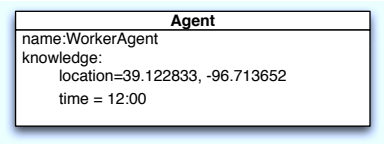


Figure 3.2: *Worker Agent Specification*

The Agent's Knowledge is a dynamic set of information that changes over time, depending on the Agent's reasoning model. Thus the Agent is allowed to save, update, or

purge information at will. In the WorkerAgent example the current time would be a piece of Knowledge that would be updated periodically (once a minute or every second). The details of the knowledge relation are not key to understanding an agent, and the formal definition of *knowledge* is located in Appendix A.2.

3.1.2 Goals

A *Goal* is the desired state of an agent within the environment. Agents start with a set of beliefs about their environment. Agents can use sensors and percepts to monitor and modify their environment. An Agent's state is the set of internal beliefs that an Agent has about their environment, but these beliefs may not correctly represent the environment. A Goal is a state (of the environment or the agent itself) that the Agent wishes to realize or achieve. The Agent may need to perform an Action or participate in an Interaction in order to achieve its Goals. Definition 2 enumerates the attributes that belong to a Goal. A Goal's *state* is described by a Specification Language expression (Definition 19). A Goal can have *inputs* that parameterize the Goal state. A Goal's inputs will affect how the Actions, Roles and Protocols are played. The inputs are a set of Variables that default to being read only. The *outputs* are a set of Variables whose value may be overwritten by the outcome of an action. If a variable is both an input and an output then the variable is read-write. Once a goal is achieved, any variables in that goal can be added to the agents knowledge.

Definition 2 (*Goal Attributes*)

name	*String*
inputs	*Set(Variable)*
outputs	*Set(Variable)*
state	*Expression*

Figure 3.3 extends Figure 3.2 by adding a goal to WorkerAgent. This goal's name is "move" and it has three parameters. The first input is the *location*, the second input is the *duration*, and the third input is the *start time*. The Goal specifies that the agent is to be at the location at a specific time (starttime + duration). The Goal has two outputs, the first

is the *finish time* and the second is the *finish location*.

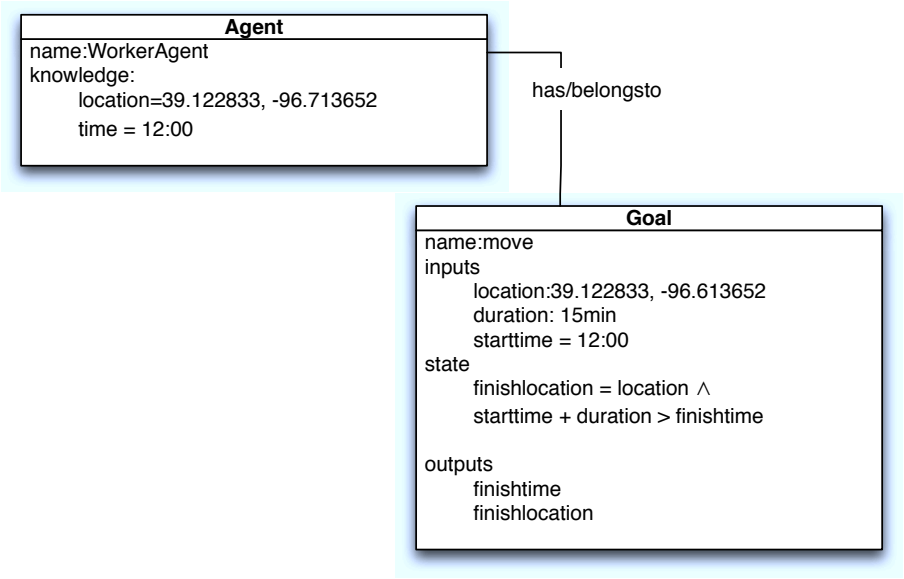


Figure 3.3: *Example Agent and Goal Specification*

Figure 3.1 shows the *has* relation between Agents and Goals. The inverse of this relation is a function, such that each Goal *belongsto* a single Agent. The *has* relation is defined in Appendix A.2 and is used in the Restrictions (Section A.6). In the example shown in Figure 3.3 the *has* relation is as follows.

$has = \{(WorkerAgent, Move)\}$

Thus, the *belongsto* function is define for one value, Move, such that $belongsTo(Move) = WorkerAgent$

Function 1 ($Goal \rightarrow Agent$)

$belongsto : Goal \rightarrow Agent$

3.1.3 Actions

An *Action* is performed by an agent in order to achieve a goal. Actions allow an agent to sense or affect their environment, which is a key tenet in multiagent systems. An Action is either something an Agent performs by itself or in the context of a group of Agents (an

interaction).

The attributes for an action are listed in Definition 3. An Action's precondition and postcondition formally specify what the Action requires and what it does. The *precondition* is an expression that must be true for the Action to function properly. The *postcondition* is an expression that must hold immediately after the Action has finished, if the precondition was initially true. The *inputs* allow the Action to be parameterized. The values acquired by the *outputs* depend on the result of the Action performed by an Agent. The *postcondition* specifies additional properties that must hold for the outputs. The example in Figure 3.4 shows the specification of an Action.

Definition 3 (*Action Attributes*)

name	*String*
inputs	*Set(Variable)*
outputs	*Set(Variable)*
precondition	*Expression*
postcondition	*Expression*

Figure 3.4 extends Figure 3.3 by adding an Action. The action's name is "Atomic Move". The Atomic Move action has six inputs and two outputs. The precondition of Atomic Move ensures that the current location is the start location and that the current location is not the finish location. The values of the outputs *time* and *finishlocation* are restricted by the properties given in the postcondition. The postcondition states that the current location is the same as the finish location and that the start time plus the allocated duration is more than the current time.

Figure 3.1 shows a relationship between Actions and Goals and between Agents and Actions. The Action-Goal relation is the *achieves* relation (Relation 1). The achieves relation defines which goals an action is capable of achieving. The *performs* relation (Relation 2) defines the allowable Actions that an Agent can perform. The allowable Actions are based on either the physical capabilities of the Agent or a set of organizational restrictions. MAIM does not define how this relation is created, only that the relation exists.

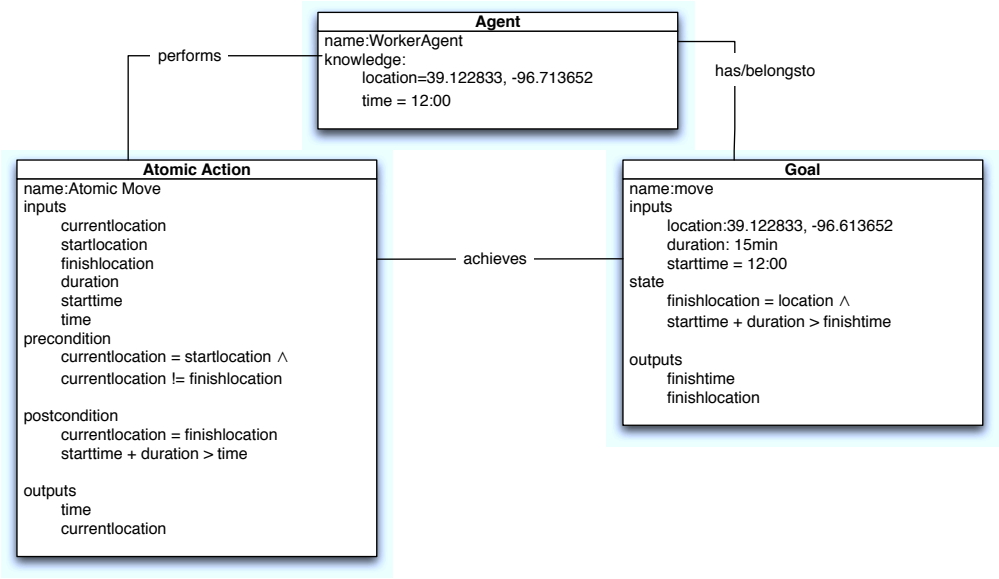


Figure 3.4: *Example Agent,Goal & Action Specification*

The *achieves* relation is a derived relation. The derivation uses formal proof techniques to prove that an Action *will* or *may* achieve a Goal. Chapter 4 defines the proof technique used to create this relation. The proof technique has two modes: proof mode and viable mode. In the proof mode the derivation proves that the Action *will* achieve the Goal. In the viable mode the Action *may* achieve the Goal. These modes are described in detail in Section 4.4.

Relation 1 (*Action Goal*)

achieves $\subseteq Goal \times Action$

Relation 2 (*Agent Action*)

performs $\subseteq Agent \times Action$

3.1.4 Atomic Actions

An *Atomic Action* is an action performed by a single agent while achieving a goal. Atomic Actions are typically the low level actions used by the agents to sense and manipulate their environment. Atomic Actions do not require agent coordination. Example capabilities

include GPS, camera, radio, locomotion and a gripper. For each of these capabilities there could be an Atomic Action. The GPS Atomic Action could define a precondition (must be able to see the sky) for using the GPS and it could define a postcondition (Location returned with a 3 meter accuracy). Figure 3.4 gives a concrete example of an Atomic Action. The Atomic Move action allows an Agent to move through its environment by using a GPS and its motion capabilities. The Atomic Move shows that an Atomic Action can merge multiple low level capabilities into a single Atomic Action. For this example, the *performs* relation now includes the Atomic Move action for the Worker Agent and the *achieves* relation includes the move goal and the Atomic Move action.

$\text{performs} = \{(\text{WorkerAgent}, \text{Atomic Move})\}$

$\text{achieves} = \{(\text{move}, \text{Atomic Move})\}$

3.1.5 Interactions

Clearly the definition of an Atomic Action is necessary but it is not a comprehensive definition. Multiagent systems require coordination and communication for a variety of reasons. For example, coordination allows a group of agents to work on tasks that are too large for one agent. Additionally, there are times where an agent does not have all of the required capabilities to achieve a specific task. Multiagent systems are inherently prone to failure and agents need to communicate and redistribute work when a failure occurs.

Interactions are one of the key entities within MAIM and FIA. The Oxford Dictionary defines an interaction as a "*reciprocal action or influence*" [77]. The meaning for an Interaction in MAIM emerges from this definition. An *Interaction* is an action that requires more than one participant and where each action has an effect on multiple entities within the Interaction. This definition broadly states that an Interaction requires more than one agent. An Interaction also requires that each action within the Interaction affects the current and future behavior of multiple agents.

Interactions are instances of Actions and, like Atomic Actions, they inherit all the at-

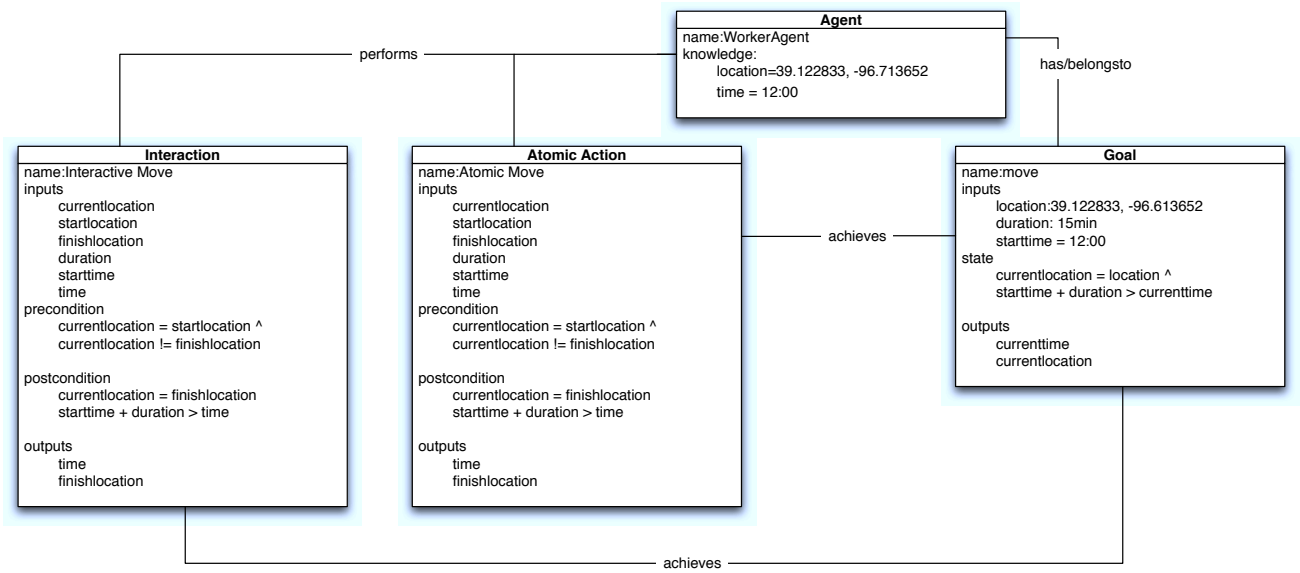


Figure 3.5: *Example Agent,Goal Action & Interaction Specification*

tributes of Actions. The major difference is that Interactions require that conditions hold for *all* agents in that interaction, whereas in Atomic Actions these conditions are required to internally hold for *each* agent. The internal states of agents within the system may differ due to the different beliefs that each agent holds. Figure 3.5 adds an Interaction to the example system. The Interactive Move has the same specification as the Atomic Move, but the Interactive Move requires multiple Agents and a Protocol to implement the Interaction.

An agent participating in an Interaction must be able to achieve at least one Goal. The precondition and postcondition of the Interaction must provide the supporting evidence that the achievement of the Goal is possible. The precondition and postcondition are used in the creation of a proof that is used to prove that the Goal achievement is possible. The updated relations for the example in Figure 3.5 are shown below.

$$\text{performs} = \{(\text{WorkerAgent}, \text{Atomic Move}), (\text{WorkerAgent}, \text{Interactive Move})\}$$

$$\text{achieves} = \{(\text{move}, \text{Atomic Move}), (\text{move}, \text{Interactive Move})\}$$

3.1.6 Interaction Role

"A *role* is a set of expectations that govern how people holding a given position should behave[109]." The definition for an Interaction Role is based on this interpersonal definition. An *Interaction Role* is the set of rights and obligations used in an Interaction. Interaction Roles are an abstract role within an Interaction that an Agent will perform. They are realized by Protocol Roles (Section 3.1.8), which define behaviors that fulfill the required obligations of the Interaction Role. Interaction Roles abstract some of the details present in the Protocol Roles which allows Interaction Roles to be more flexible and less dependent on the Protocol or the Agent that implements them. The Interaction Role Attributes are shown in Definition 4. The Interaction Role has read-only *inputs*. Given the inputs, if the *precondition* of the Interaction Role is met, then the *postcondition* holds when the role is finished. The *outputs* are a set of variables whose values are produced by the Interaction Role.

Definition 4 (*Interaction Role Attributes*)

name	*String*
inputs	*Set(Variable)*
precondition	*Expression*
postcondition	*Expression*
outputs	*Set(Variable)*

Figure 3.1 shows a compositional relationship between the Interaction and the Interaction Roles. This relationship is formalized by the *InteractionRoleComposition* relation shown in Relation 3. The composition relation restricts the *implement* relation between Interactions and Protocols, thus it restricts which Protocols can be used to realize an Interaction.

Relation 3 (*Interaction Role Composition*)

InteractionRoleComposition $\subseteq$ *Interaction* $\times$ *Interaction Role*

Figure 3.6 adds two Interaction Roles to the Interactive Move Interaction. The Interactive Mover Role is the Interaction Role played by the agent that is moving. The Interaction Location Role is the role that is played by an Agent that is able to determine the location of

another Agent. A topographical survey crew is a real world example of a system with two similar such roles[35]. In a surveying crew, one person stands at a known location and uses a theodolite to measure the location of the level staff. The level staff is held by a person moving within the environment. In the example shown in Figure 3.6 an Agent can be a mover agent if they can communicate with the Agent playing the Interactive Location Role. The Agent playing the Interaction Mover Role can move to a location without having the ability to determine its location using its built in sensors.

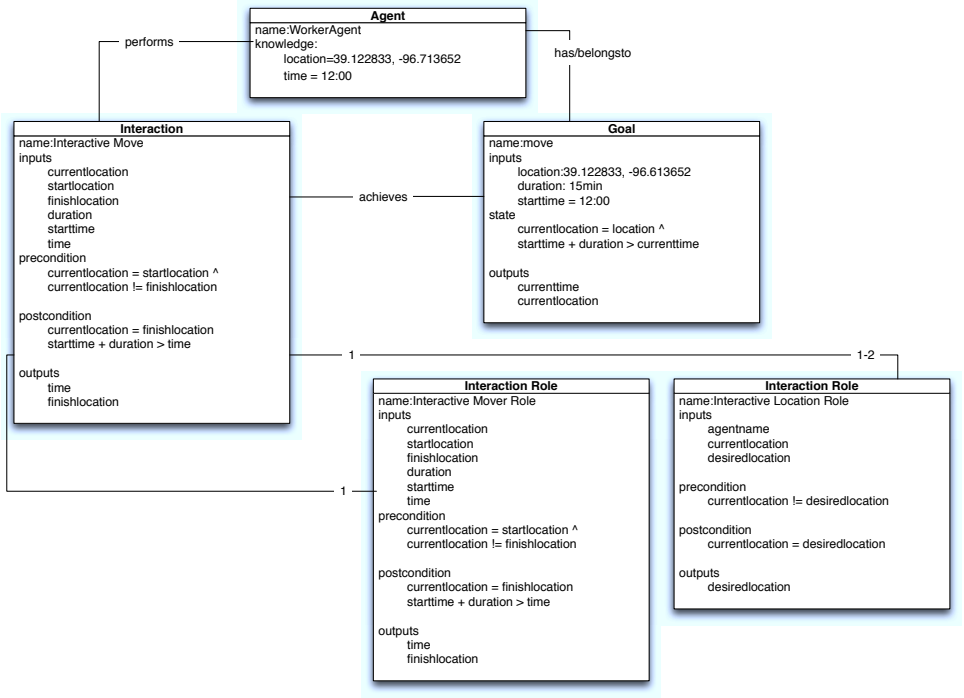


Figure 3.6: *Example Agent,Goal, Interaction & Interaction Role Specification*

There are several additional relations and restrictions that are placed on Interaction Roles, as listed in Appendix A.3. These restrictions include the cardinality rules for Interaction Role composition and a uniqueness property for the composition. The Role composition rules ensure that each Interaction is composed of two or more Interaction Roles, and that each Interaction Role is part of one Interaction. The cardinality of the composition allows the designer to specify the minimum and the maximum number of occurrences of each role.

The example in 3.6 shows that there can be 1-2 instances of the Interactive Location Role in the Interactive Move Interaction: 2 instances of the Interactive Location Role allow the agent playing the Interactive Mover Role to gather data from two different sources.

3.1.7 Protocols

Protocols have long been used in computer science to transfer information from one point to another. A *protocol* is a set of rules governing the exchange or transmission of data electronically between devices[77]. Protocols allow information to flow between the agents in the system and they define the rules that govern information flow. The Protocol definition is similar to the definition of an Interaction, in that two or more Protocol Roles are composed to define a Protocol. Given the *inputs*, a protocol guarantees that the properties will hold during the protocol. These protocol properties are specified in Linear Temporal Logic (LTL) so their truth value depends on the point in time which they are checked. Some properties hold when the protocol starts, some hold during the execution of the protocol, and some hold when the protocol is finished. The *outputs* are the set of Variables whose values are a product of the completion of the protocol. As with Interactions, the outputs can overwrite the values of the inputs. Figure 3.7 shows the specification of an example movement protocol. This protocol allows two agents to coordinate such that an agent that does not have a GPS can move to a location specified by GPS coordinates. The properties of the protocol specify that the agent either arrives at the location in the allocated time or the protocol has failed.

Definition 5 (*Protocol Attributes*)

name	*String*
inputparameters	*Set(Variable)*
properties	*Set(Expression)*
outputs	*Set(Variable)*

The definition of a Protocol in MAIM allows the model to leverage the current state of the art techniques in Protocol design. The model can take any protocol specified in LTL and reason about the protocol within the context of the model. The ability to reason

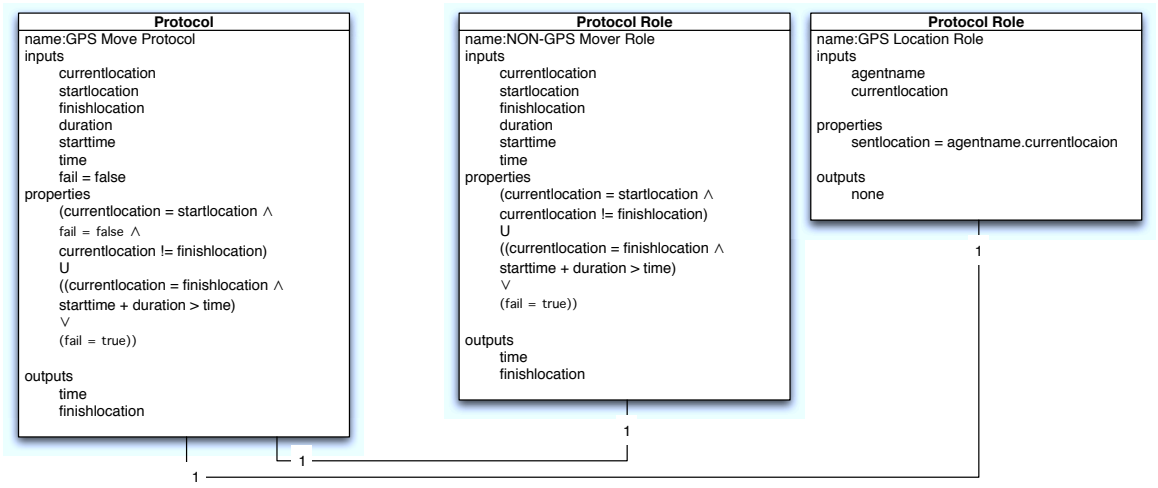


Figure 3.7: *GPS Move Protocol & Protocol Role Specification*

about protocols allows the agents to use predefined protocols in FIA and MAIM. Designers must use a tools such as Spin[22] to verify that a model ensures the properties that it states. Designers could also use JPF[17] to prove that the actual code meets the specification given.

Figure 3.1 shows a relation between Interactions and Protocols. The Relation 4 defines that a Protocol *implements* an Interaction. The implements relation shows which protocols have the ability to do the work required by an Interaction. There are two properties that are required for a Protocol to implement an Interaction.

I. Properties of Protocol hold initially given only the inputs specified in the Interaction precondition.

II. The Interaction postcondition can be proven to be true given the properties that are true at the completion of the Protocol.

The above two properties ensure that the Interaction can be implemented by the Protocol. The *implements* relation is formally defined in Relation 4. If both Property I and Property II hold then a protocol may implement an Interaction. There are additional restrictions that apply to Interaction Roles and Protocol Roles that also hold. These additional properties are described in the next section.

Relation 4 (*Protocol implements Interaction*)

$$\textbf{\textit{implements}} \subseteq \mathit{Interaction} \times \mathit{Protocol}$$

Figure 3.8 shows that GPS Move Protocol implements the Interactive Move Interaction. The implements relation for the example is as follows.

$$implements = \{(GPSMoveProtocol, InteractiveMove)\}$$

Property I verifies that the precondition is valid. In the example system Property I requires that we show that the following property holds.

Example 1

$$currentlocation = startlocation \wedge currentlocation \neq finishlocation$$

implies

$$currentlocation = startlocation \wedge fail = false \wedge currentlocation \neq finishlocation$$

The above property only holds if $fail = false$ is *true* when the Protocol starts. Figure 3.8 shows that the initial value for fail is false, and thus the property holds. In the Example system, Property II is as follows.

Example 2

$$((currentlocation = finishlocation \wedge starttime + duration > time) \vee (fail = true))$$

implies

$$((currentlocation = finishlocation \wedge starttime + duration > time))$$

It should be clear that the above property is not true. The condition $fail = true$ allows the implication to not hold. Chapter 4 defines a formal model that provides a more flexible proof technique which shows that there is a situation where the property can hold. Figure 3.1 shows the implements relation between a Protocol and an Interaction.

3.1.8 Protocol Role

Protocol Roles are a key element of the standard protocol definition in Computer Science. A *Protocol Role* is a set of behaviors, rights and obligations involved in participating in a

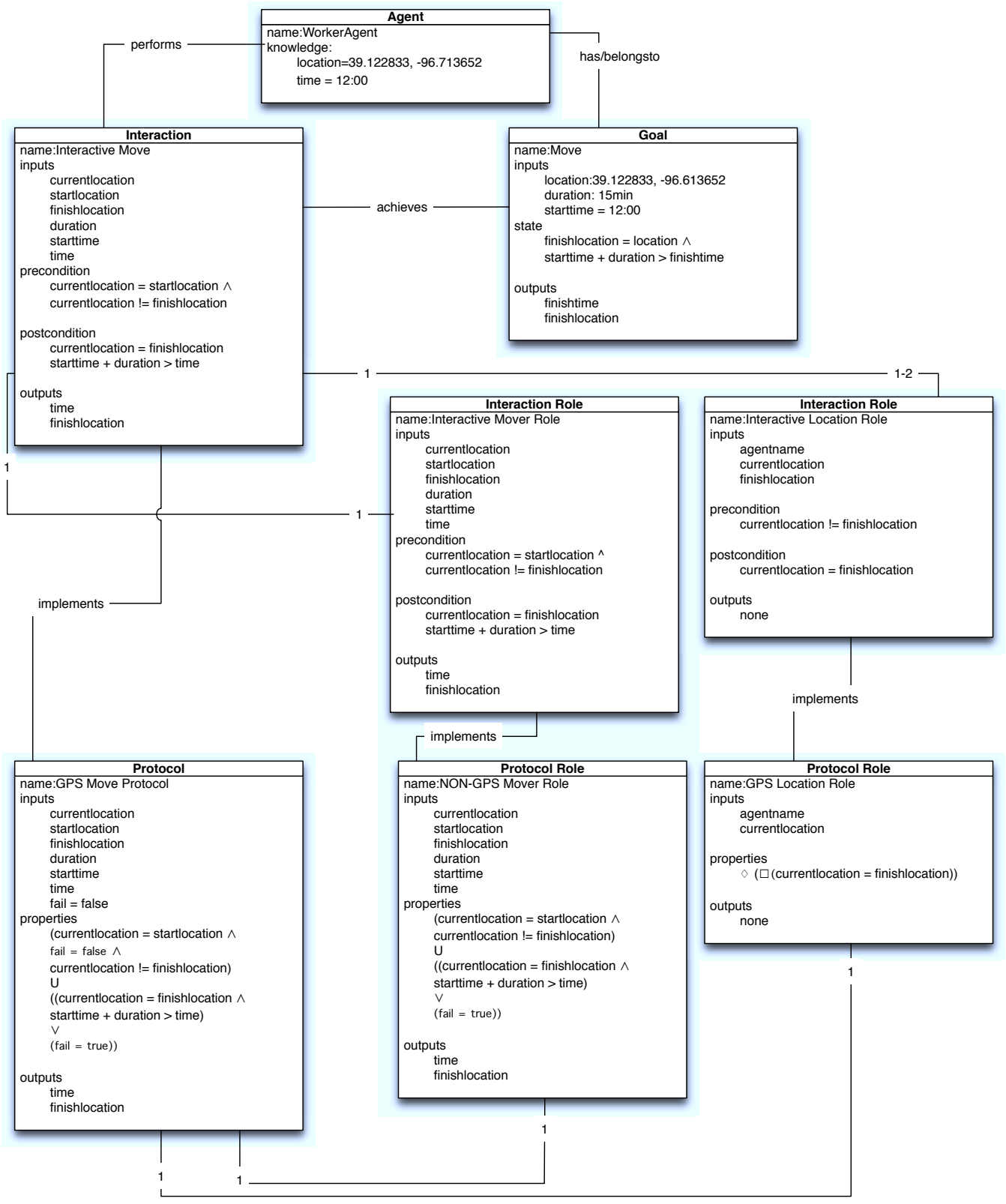


Figure 3.8: *Full Example Specification*

protocol. This Protocol Role is also based on the interpersonal definition[109] (just as the Interaction Role) . A Protocol Role defines the actions and responses for a particular agent within a Protocol. The Protocol Role attributes mirror the attributes that are defined for the Protocol. However, the properties are defined for the Protocol Role may be more restrictive than the Protocol properties. This allows the Protocol Role to be more specific than the Protocol. More restrictive roles for end points are common in actual protocol definitions. For example the TCP/IP protocol specifies that clients make a connection to a ip address and port. The server has the additional constraint that no other ip address can have a connection on that same port. This property is so common that the Promela language[102] and Spin model checker[22] have primitives for defining properties for both a protocol level property and a protocol role (process) level property.

Definition 6 (*Protocol Role Attributes*)

name	*String*
inputs	*Set(Variable)*
properties	*Set(Expression)*
outputs	*Set(Variable)*

A Protocol is composed of two or more Protocol Roles. Figure 3.1 shows the relation between each Protocol and the Protocol Roles, which is defined in Relation 5. This relation must be defined by the Protocol designer and the properties should be verified by a model checker.

Relation 5 (*Protocol Role Composition*)

ProtocolRoleComposition $\subseteq$ *Protocol* $\times$ *Protocol Role*

Figure 3.8 shows two Protocol Roles for the GPS Move Protocol. The first role is the NON-GPS Mover role. This role specifies the properties for the agent that is doing the actual moving. The GPS Location Role defines the protocol for the agent that is calculating and sending the location of the moving agent. It should be clear that the properties for the GPS Location Role are different than the properties for the protocol as a whole.

Protocol Roles *implement* Interaction Roles. This relation is defined in Relation 11 in Appendix A.4. The implements relation for the example in Figure 5 is as follows.

implements = { (NON-GPS Mover Role, Interactive Mover Role),(GPL Location Role,Interactive Location Role)}

The Protocol/Interaction implements relation requires two properties to hold. The Protocol Role/Interaction Role implements relation also requires two properties to hold.

I Properties of Protocol Role hold initially given only the inputs specified in the Interaction Role precondition.

II The Interaction Role postcondition can be proven to be true given the properties of the Protocol Role which are true at the completion of the Protocol Role.

The Protocol/Interaction implements relation gains additional constraints with the addition of Protocol Roles. If a Protocol implements an Interaction, then there must be a mapping of each Protocol Role to an Interaction Role and for each Interaction Role there must an implementing Protocol Role. This ensures that there is a match between roles defined in the Protocol and the roles defined in the Interaction.

In Figure 3.8 each of the Interaction Roles has a implementing Protocol Role. The Interactive Mover Role is implemented by the NON-GPS Mover Role. The Interactive Location Role is implemented by the GPS Location Role.

Appendix A.4 includes the above restrictions on the implements relation. The Appendix also includes the cardinality rules for Protocol Role composition and a uniqueness property for the composition.

3.2 Bindings

MAIM defines the entities required for FIA. The model is defined such that the information resides within the Agent and the Goal. If the Agent chooses to perform an Action then the Agent must transfer information into the Action. The information resides in Variables

located in the Agents Knowledge and the Agents Goals. In the example in Figure 3.8 the Worker Agent has a Variable named *location* which has the same semantics as the Move goal's *currentlocation* Variable.

Up to this point many multiagent systems have provided models that are similar to MAIM. System designers have two options in regards to moving information from the Agent's Knowledge to the inputs of an Action. A designer can either hard code how the information flows or there can exist a mechanism, within the model, that can automatically move the information correctly.

FIA uses the latter of the two options. FIA defines a mechanism that allows information to flow from the Agent to the Interactions and Protocols. This mechanism, called a *binding*, provides a way to relate the Variables within the Agent's Knowledge and Goals to the variables within the Actions, Interactions and Protocols. This section provides the intuitive definition of bindings. The formal definition of bindings is located in Appendix C. FIA defines three general types of bindings.

I. Action Level

II. Interaction Level

III. Role Level

The Action Level binding (Type I) is a binding between the Agent and an Action (Left hand side of Figure 3.9). This binding is useful only for Atomic Actions. The Interaction Level binding (Type II) is a binding between a Goal, an Agent, an Interaction and a Protocol (Figure 3.9). The Role Level binding (Type III) is the Role level binding (Figure 3.10). The role level binding is a binding between a Goal, an Agent, an Interaction Role and a Protocol Role. This type of binding determines which roles an agent is allowed to play within the Interaction and the Protocol.

An Agent that has the Action Level bindings can execute an Atomic Action. If the Agent has both the Interaction Level and the Role Level bindings it can participate in an

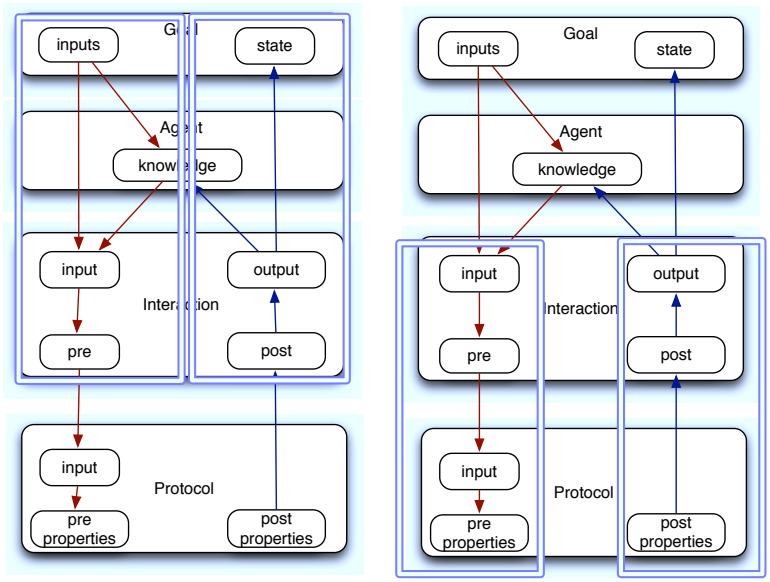


Figure 3.9: *Required Action/Interaction Level Bindings*

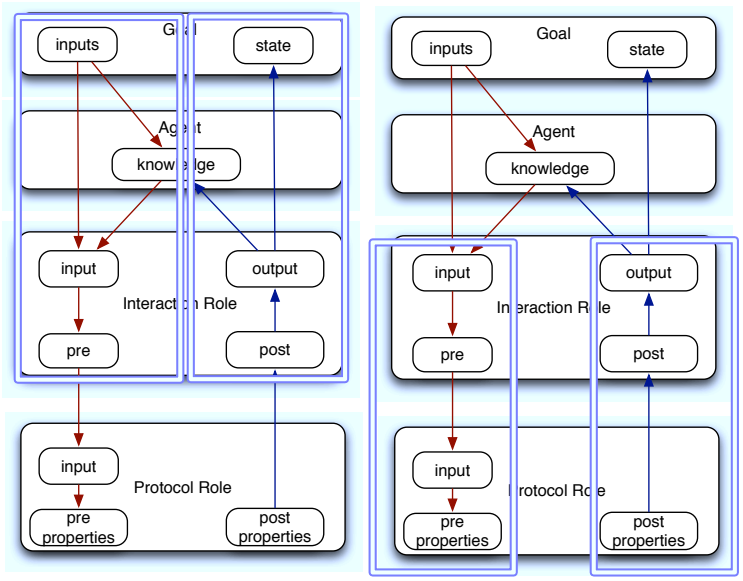


Figure 3.10: *Required Role Level Bindings*

Interaction using a Protocol. An Agent participating in an Interaction must play at least one of the Interaction Roles that the Interaction is composed of. For each of the Interaction Roles that the agent is playing, the agent must play the corresponding Protocol Roles that implement those Interaction Roles.

An example input binding may help clarify how bindings work. The Interaction Interactive Move in the Move Example (Figure 3.8) includes several inputs. The value for the input *currentlocation* may come from the WorkerAgent's Knowledge of *location* and the input *finishLocation* may get its value from the Move Goal's input *location*. A binding creates an inter-entity mapping from inputs to values. This mapping provides the Action with the required inputs.

An example output binding for the Goal Move may give the output *finishtime* the value that is stored in the Interactive Move's *time* output. Additionally the Move's *finisihLocation* may receive its value from the Interactive Move's *finishLocation* output. This binding provides the Agent with the required information to check that the Goal has been achieved.

3.3 MAIM Assumptions

MAIM requires that the designer has done three things (I) modeled the Protocol, (II) specified Protocol and Protocol Role properties, and (III) verified that the model possesses those properties. This verification provides the MAIM with a ground truth on which other properties can be derived. The model defines Restriction 1 which requires that the protocol properties imply the conjunction of all the protocol roles that implement that protocol. This restriction ensures that a proof can be made with the Protocols properties. If the properties of the Protocol do not imply the properties of the Protocol Roles, then it is be possible to construct two proofs that are inconsistent with one another. The ability to construct inconsistent proofs leads to undesirable behavior and thus it is disallowed.

Restriction 1 (*Protocol implies Protocol Roles*)

$\forall p\ Protocol\ |(p.properties \implies (\forall pr : Protocol\ Role\ |(p, pr) \in implements \bigwedge_{pr.properties}))$

In addition to the Protocol restriction, there are two restrictions defined for Interactions and Interaction Roles. Restriction 2 ensures that the Interaction precondition implies the conjunction of the Interaction Role preconditions. Restriction 3 ensures the Interaction postcondition implies the conjunction of all the Interaction Role postconditions.

Restriction 2 (*Interaction precondition implies Interaction Role precondition*)

$\forall i\ Interaction\ |(i.precondition \implies \forall ir : Interaction\ Role\ |(i, ir) \in implements \bigwedge_{ir.precondition})$

Restriction 3 (*Interaction postcondition implies Interaction Role postcondition*)

$\forall i\ Interaction\ |(i.postcondition \implies \forall ir : Interaction\ Role\ |(i, ir) \in implements \bigwedge_{ir.postcondition})$

The above restrictions ensure that inconsistent proofs cannot be made using the Protocols and Interactions. The next section defines how the protocol properties can be converted into pre and post conditions.

3.4 LTL Logic Conversion

Linear Temporal Logic (LTL) is logic that allows a designer to specify properties that are based on both relative and absolute time constraints. The logic supports properties that either always hold or hold for a period in time. Protocols are defined using LTL logical operators. The protocol properties can be verified for a model using a model checker like Spin[22]. FIA's algorithms reason about Protocols and Protocol Roles using non-LTL properties. For a precondition check, the LTL properties are converted to properties that hold initially. As these properties hold initially they are analogous to a precondition in an Interaction or Interaction Role. When creating a Goal satisfaction proof, the Protocol properties are converted to properties that must hold when the protocol completes. This is analogous to the postcondition in the Interaction or Interaction Role.

There are five basic LTL operators that are defined in the Specification Language. These

operators are listed in Table 3.1. Given these canonical definitions, FIA must show that the

Global	Gϕ	$\Box\phi$	Property ϕ always holds.
Next	N ϕ	$\circ\phi$	Property ϕ holds in the next state.
Finally	F ϕ	$\Diamond\phi$	Property ϕ holds at some point
Until	ψ U ϕ	$\psi\mathcal{U}\phi$	Property ψ holds up to the point where ϕ holds
Release	ψ R ϕ	$\psi\mathcal{R}\phi$	Property ϕ holds up to the point where ψ becomes true

Table 3.1: *LTL operators*

preconditions of the Interaction do not conflict with the initial state of the Protocol. The definition of the LTL rules assumes that extraneous LTL operators have been removed or simplified. This makes the LTL rules simpler to describe. The actual implementation does not assume simplification and thus it simplifies the expressions as it is processing them.

3.4.1 Preconditions

The conversion algorithm derives, from the input properties, a set of properties that hold initially. The implementation leverages the logical foundations of the Prolog programming language. Prolog can derive new rules based on previous facts and derived facts. Table 3.2 lists the conversion rules that are defined in Prolog. Line 1 shows the Global operator. A property that is globally true for the Protocol must be true when the Protocol starts. Line 2 defines the Next operator. This operator states that property ϕ will be true in the *next* state and not the current state and thus we do not add any additional properties. Line 3 defines Finally operator. This operator defines that eventually property ϕ will be true. This does apply to the current state and thus no additional properties are added. Line 4 is the Until operator. This operator specifies that ψ has to hold at least until ϕ holds. By the definition ψ must hold to start, and that ϕ may eventually be true. The algorithm derives that ψ holds initially. The algorithm excludes ϕ as it adds no additional properties. The final operator, Release is on Line 5. This operator specifies that ϕ must hold until and up to the point at which ψ holds. By definition ϕ must hold initially and thus the algorithm adds ϕ to the result. After the algorithm processes the above rules, the result is a list of

1	Gϕ	$\Longrightarrow$	ϕ
2	Nϕ	$\Longrightarrow$	true
3	Fϕ	$\Longrightarrow$	true
4	ψUϕ	$\Longrightarrow$	ψ
5	ψRϕ	$\Longrightarrow$	ϕ

Table 3.2: *LTL Precondition Derivation*

properties that initially hold for the Protocol. The Interaction precondition is combined with these properties to create a single conjunctive condition. This single condition is run through a SAT solver to ensure that there are no conflicts.

3.4.2 Postcondition

The postcondition conversion algorithm converts the LTL properties into properties that must hold when the Protocol has finished. The Protocol properties are used to create the proof that the Interactions postcondition holds when the Protocol finishes. The Proto-

1	Gϕ	$\Longrightarrow$	ϕ
2	Nϕ	$\Longrightarrow$	true
3	Fϕ	$\Longrightarrow$	true
4	ψUϕ	$\Longrightarrow$	true
5	ψRϕ	$\Longrightarrow$	true
6	$(\beta = G\gamma)$, Fβ	$\Longrightarrow$	γ
7	$(\beta = G\gamma)$, $\beta R\alpha$	$\Longrightarrow$	$\gamma \vee \alpha$
8	$(\beta = G\gamma)$, $\alpha U\beta$	$\Longrightarrow$	$\gamma \vee \alpha$
9	$\neg F\alpha$, $\alpha R\beta$	$\Longrightarrow$	β
10	$\neg F\beta$, $\alpha U\beta$	$\Longrightarrow$	α
11	$(\beta = G\gamma)$,$F\neg\alpha$, $\alpha U\beta$	$\Longrightarrow$	γ
12	$(\alpha = G\gamma)$,$F\neg\beta$, $\alpha R\beta$	$\Longrightarrow$	γ

Table 3.3: *Postcondition LTL conversion Table*

col postcondition derivation includes rules that require many facts. Prolog and XSB are specifically tailored for this kind of logical reasoning, which is why they were chosen as the language for implementation. The right most column of the table defines the new facts that are derived. If the value in the rightmost column is true, then no additional facts are added.

Line 1 defines the rule for the global operator. If something is globally true, it is added to the current set of postcondition facts.

The next operator (Line 2) does not supply any additional information, thus true is added to the postcondition facts.

Line 3 defines the rule for the finally operator. At first glance, it appears that ϕ should be included in the postcondition, but the semantics of the finally operator states that ϕ will eventually become true. This implies that it could become false at some point. Additional information is required in order to derive a postcondition property out of the finally operator. Line 4 and 5 define the rules for the until operator and the release operator. Each of these operators works like the finally operator in that no additional facts can be derived without additional information.

Line 6 defines the first multi-fact based rule. This rule combines a finally and a global. If we know that eventually β is true, and β is a global property $G\gamma$, then γ must hold in the final state.

Line 7 defines a compound rule that leverages the release property. Given a release $\beta R\alpha$ and it is the fact that β is a global $G\gamma$ then either α or γ is true in the postcondition. Thus either α is always true or β becomes true and remains true for the rest of the protocol. This rule adds a pair of disjunctive clauses, which may not be helpful most of the time as it weakens the premise and thus weakens the conclusions that can be derived. There are circumstances that can leverage this additional information, such as $\neg\alpha$. A stronger postcondition may be derived, thus the disjunction is included.

Line 8 defines a compound rule for the until operator. Given the property $\alpha U\beta$ and the property β is a global property ($G\gamma$), then either α or γ is true in the postcondition. This rule is similar to rule 7.

Line 9 defines another compound release rule. Given a release $\alpha R\beta$ property and the fact that α will never be true ($\neg F\alpha$) then the algorithm can derive that β is be true in the postcondition. In this case the definition of release states that if α is never true, then β

must remain true forever.

Line 10 defines a similar rule to the above rule. This rule applies the negation to the until operator instead of the release operator. Given the until property $\alpha U \beta$ and given the fact thatβ never becomes true then α will hold forever and when the Protocol completes.

Line 11 defines another compound rule for the until operator. Given the until property $\alpha U \beta$. If α is eventually false then β will eventually be true. If β is in the form of a global property ($G\gamma$), then γ must be true in the postcondition.

Line 12 defines a similar property to the property on Line 11 but it utilizes the release operator. Given the property $\alpha R \beta$ and the property that eventually β is not true and α is in the form of a global property $G\gamma$, then it is the case that γ will be true in the postcondition.

After the Protocol postcondition is derived (using the rules above) the Protocol postcondition and the Interaction postcondition are run through a SAT solver.

The XSB prolog code that implements these properties is listed in Appendix B.1. The code listed is for both the precondition and the postcondition derivations.

3.5 Interaction Model Summary

This chapter has laid the groundwork for FIA. The entities that comprise MAIM as well as the relationships between these entities have been defined. Agents, Goals, Atomic Actions, Interactions, Interaction Roles, Protocols and Protocol Roles are the formally defined entities. Additionally, the inter-entity relationships, such as the relationship between Agents and Interactions, are defined in the Model. This formal model for Interactions allows reasoning tools and algorithms to reason about the behavior of such a system. The LTL conversion rules define how the FIA algorithms can convert the LTL protocol properties into preconditions and postconditions. The next chapter provides such reasoning algorithms.

Chapter 4

FIA Algorithms

This chapter defines the algorithms used by FIA to reason about a system designed with MAIM. Section 4.2 defines bindings and how they are formed. Section 4.3 defines the algorithms that reason over MAIM. Section 4.4 defines the two different modes that the algorithms can use and Section 4.6 defines how the algorithms create and check the proofs that use the bindings defined in FIA.

4.1 Introduction

As presented in Chapter 3, MAIM defines the entities within a multiagent system. These entities are Agents, Goals, Atomic Actions, Interactions, Interaction Roles, Protocols and Protocol Roles. In order to define a set of algorithms to reason over MAIM, a few formal definitions are given. Definition 7 formally defines that all of the main elements of MAIM are entities that can be reasoned over.

Definition 7 (*Entity Definition*)

$$Entity = set(Agent) \cup set(Goal) \cup set(Atomic\ Action) \cup set(Interaction) \cup set(Interaction\ Role) \cup set(Protocol) \cup set(Protocol\ Role)$$

Interactive multiagent systems require communication between agents. In addition to requiring agent communication, a system constructed using FIA requires that information flows between Entities. Definition 8 defines a tuple that includes each of the model entities.

This tuple is defined to facilitate the understanding of algorithms and illustrate how the algorithms in the MAIM can reason about the system behavior.

Definition 8 (*Interaction Tuple*)

<Goal,Agent,Interaction,Interaction Role,Protocol,Protocol Role>

MAIM (Section 3.2) defined a binding as a formal relation between Variables within entities. Each entity in MAIM has a formal specification that contains Variables used for input and output. Thus, by definition, every Variable has an implicit context within an Entity. The implicit context is formalized in the tuple definition defined below.

Definition 9 (*Variable Pair*)

<Entity e, Variable v> | $v \in$ *variables(e)*

The VariablePair relation has a validity restriction that ensures that context of the Variable is correct. In valid VariablePair, the Entity e must have a Variable v. Appendix C.2 defines the *variables* function used below.

Two variables that are *bound* make up a tuple of VariablePair's(<VariablePair,VariablePair>). A binding, as shown in Definition 10, is formally defined as a set of *bound* Variables.

Definition 10 (*Binding Relation*)

bound = <VariablePair a,VariablePair b>

binding = set(bound)

The specification for each entity is different and thus where a variable is defined will vary. For example, in the Interaction Role Variables exist in the precondition and the postcondition, but in the Goal the Variables exist in the state. The location of the variables is defined in Appendix C.2. A high level overview of the binding model is shown in Figure 4.1. Each major entity in MAIM is shown and the binding functions are represented by the arrows. The bindings that are used for input are arrows pointing down (red), and the output bindings are shown by the arrows pointing up (blue). The direction of the arrows shows the information flow. The information can flow from the Goal and the Agents' Knowledge into the Interaction and Protocol. The output of the Protocol flows into the Interaction, which

then flows back to the Agent's Knowledge and the Agent's goal.

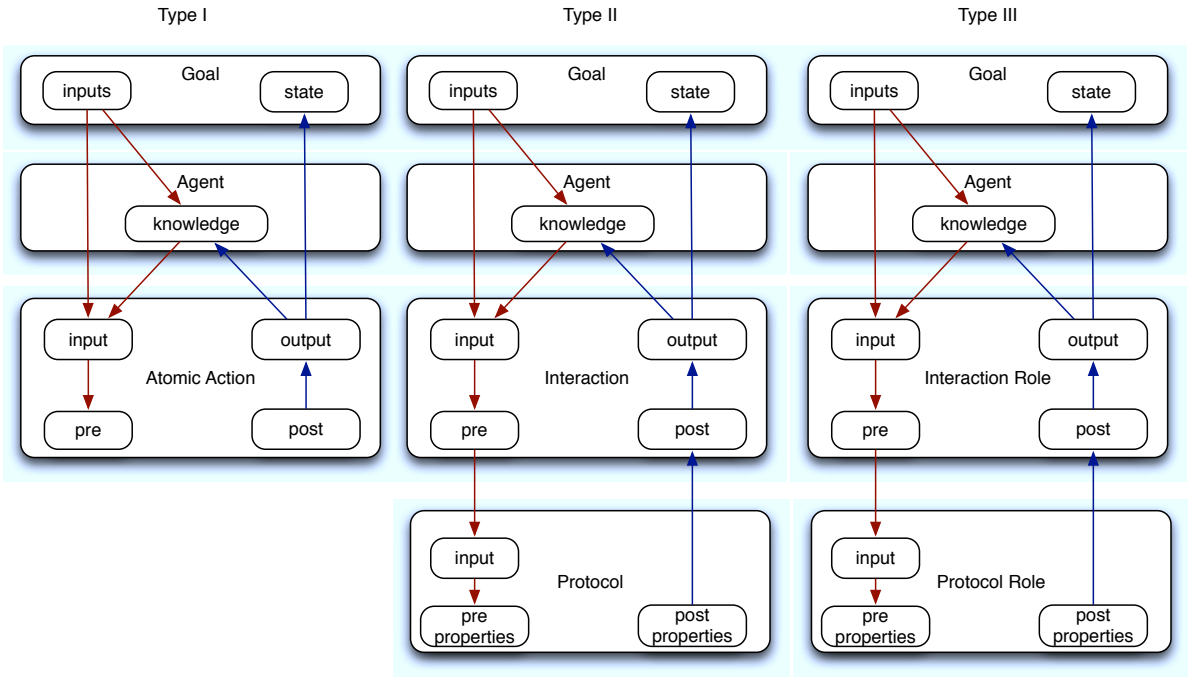


Figure 4.1: *Binding Types*

4.2 AutoBinding System Overview

The Autobinding System provides two distinct modes (Section 4.4). The first mode is proof mode. In proof mode the system can accept an Interaction Tuple as input and return true **iff** the goal **will** be satisfied. The second mode is the viable mode. In this mode the system will accept an Interaction Tuple as input and return true if the goal **may** be satisfied.

An example system has been designed to show how the binding system works. The example system is a multiagent system where there are Agents that need to buy and sell items. The example system uses an auction for the exchange of items. There are two popular auction types that are used to auction items. The first type (and most familiar) is the English Auction. In an English Auction, the seller starts with a low price and as Agents bid, the price increases until there is only one bidder left. The advantage of an English Auction, is that the item being sold should sell for the maximum price. The disadvantage

to the English Auction is that there is no limit on the number of bids that may be placed. The second type is the Dutch Auction. Dutch auctions are so named, as they are used in the Netherlands to auction tulips. In a Dutch auction the price starts at the maximum price the seller desires and the price keeps being lowered until someone bids, or the minimum is reached. The advantage of a Dutch auction is that the time for the auction can be determined beforehand, as the maximum number of rounds can be calculated. The disadvantage of the Dutch auction is that a maximum price is predetermined.

4.2.1 Example System Details

A high level overview of the example is shown in Figure 4.2. There are two agents, one Agent (Seller Agent) that has a Goal to sell an item (Sell Item Goal), and a second Agent (Buyer Agent) has a goal to buy an item (Buy Item Goal). For the sake of readability the item details have been omitted (such as name, type etc). This will make the model more readable and that does not affect the outcome of the algorithm. There is one Interaction (Sell Interaction) that should satisfy both of the agent's goals. The Sell Interaction has two Interaction Roles (Buyer Interaction Role and Seller Interaction Role) that implement the Interaction. There are also two protocols (Dutch and English Auction) that implement the Sell Item Interaction. The Dutch Auction has two roles, the seller role (Dutch Auction Seller Role), and the buyer role (Dutch Auction Buyer Role). The English Auction also has two similar roles, a seller role (English Auction Seller Role) and a buyer role (English Auction Buyer Role).

The specification for the entire example system is shown in Figure 4.3. Each entity's specification is shown in detail. The details of each specification are described below.

The Seller Agent has one goal, the Sell Item Goal. The Sell Item Goal has one input named min. An input is a parameter to a goal that specializes the goal in order to accommodate the needs of the agent. The min is the minimum price the Seller Agent is willing to accept. The goal states that the price is greater than or equal to the minimum and there is

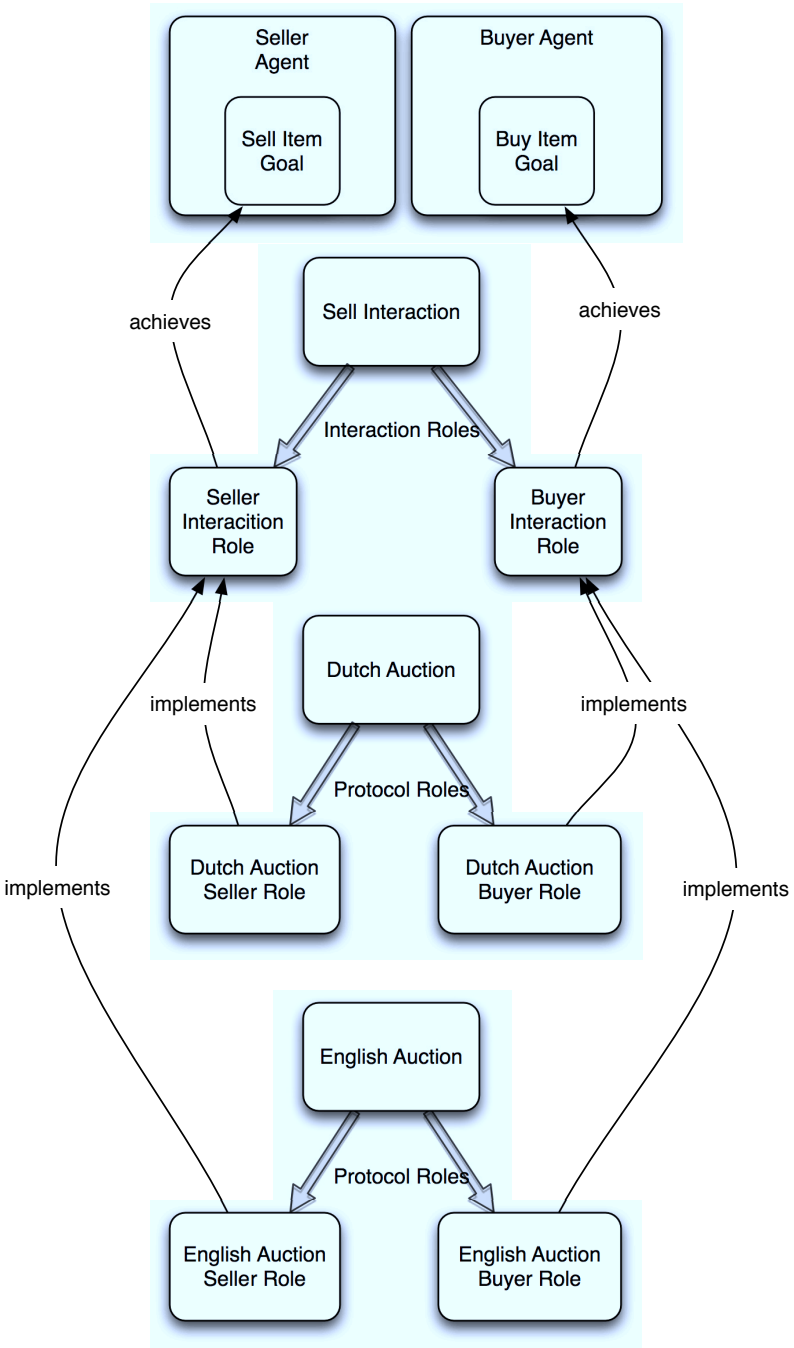


Figure 4.2: *Auction High-level Example*

a winner.

The Buyer Agent also has one goal, which is the Buy Item Goal. The Buy Item Goal has one input named max. The max is the maximum price that the Buyer Agent is willing to pay for the item. The desired state of the Buy Item Goal is that the price is less than or equal to the max and this agent is the winner.

The Sell Interaction has two Interaction Roles, the Seller Interaction Role and the Buyer Interaction Role. The Sell Interaction has two inputs, min and max. These inputs are the minimum and maximum that an item can be sold for. The precondition for the Interaction states that min is less than or equal to max. If the precondition is true, then the postcondition states that at the price is less than or equal to the max, the price is greater than or equal to the min, the Winner is the current agent, and the Winner is not null. The Sell Interaction has two outputs: price and Winner. The postcondition describes the possible values for price and Winner, thus the possible values are restricted. The Sell Interaction is a aggregation of the roles it provides, and thus the information in the Sell Interaction applies to both the seller and the buyer. So, when the interaction states that the Winner is the current agent, that is applicable for the Buyer Agent, but it provides no helpful information to the Seller Agent.

The Seller Interaction Role requires one input, min. The min is the minimum that the Agent playing that role is willing to sell the item for. There is no precondition for this Role. The postcondition states that the price is greater than or equal to the min and there is a winner. The outputs for this Role are the price and the Winner. The min is not listed as an output as the min should not change during the course of playing the role.

The Buyer Interaction Role also requires one input, max. The max is the maximum that the buyer is willing to buy an item for. There is no precondition for this interaction role. The postcondition states that the price is less than or equal to max and the winner is this participant. The outputs for this Role are the price and the Winner.

The Dutch Auction Protocol has two inputs, min and max. Again the min is the min-

imum price, and the max is the maximum price. There are three properties of the Dutch Auction Protocol. The first property states that eventually the price is less than or equal to max and this is the winner, or someone is the winner or the item is not sold. The second property states that eventually either the price is greater than min and there is a winner, or the item is not sold. The last property states that there is at most one bid.

The Dutch Auction Seller Role has one input, min. There are two properties that this role contains. The first property states that eventually either the price is less than or equal to the min and there is a winner, or the item was not sold. The second property states that there is at most one bid.

The Dutch Auction Buyer Role also has one input, max. This role also has two properties. The first property states that eventually either the price is less than or equal to the max and this agent is the winner, or there is some other winner, or the item was not sold. The second property states that there is at most one bid.

The English Auction has a very similar form to the Dutch Auction, except that the property of *There is at most one bid* becomes *There can be many bids.*

4.3 Automatic binding generation

By definition the Variables in MAIM have types. The types allow the algorithm to match Variables in one context to Variables in another context. The definition of VariablePairs encapsulates the implicit context of each Variable. The Type System definition ensures consistency across the domain of MAIM. In the Auction example System, the *Sell Item Goal* has a **PRICE** type and *Sell Item Interaction* has a **PRICE** type. The semantics of those two types are guaranteed to be the same by MAIM.

4.3.1 Binding Algorithm

Figure 4.1 shows the three types of bindings that can be created (Type I, Type II and Type III). The full definition of each of these types is described in full detail in Appendix C.

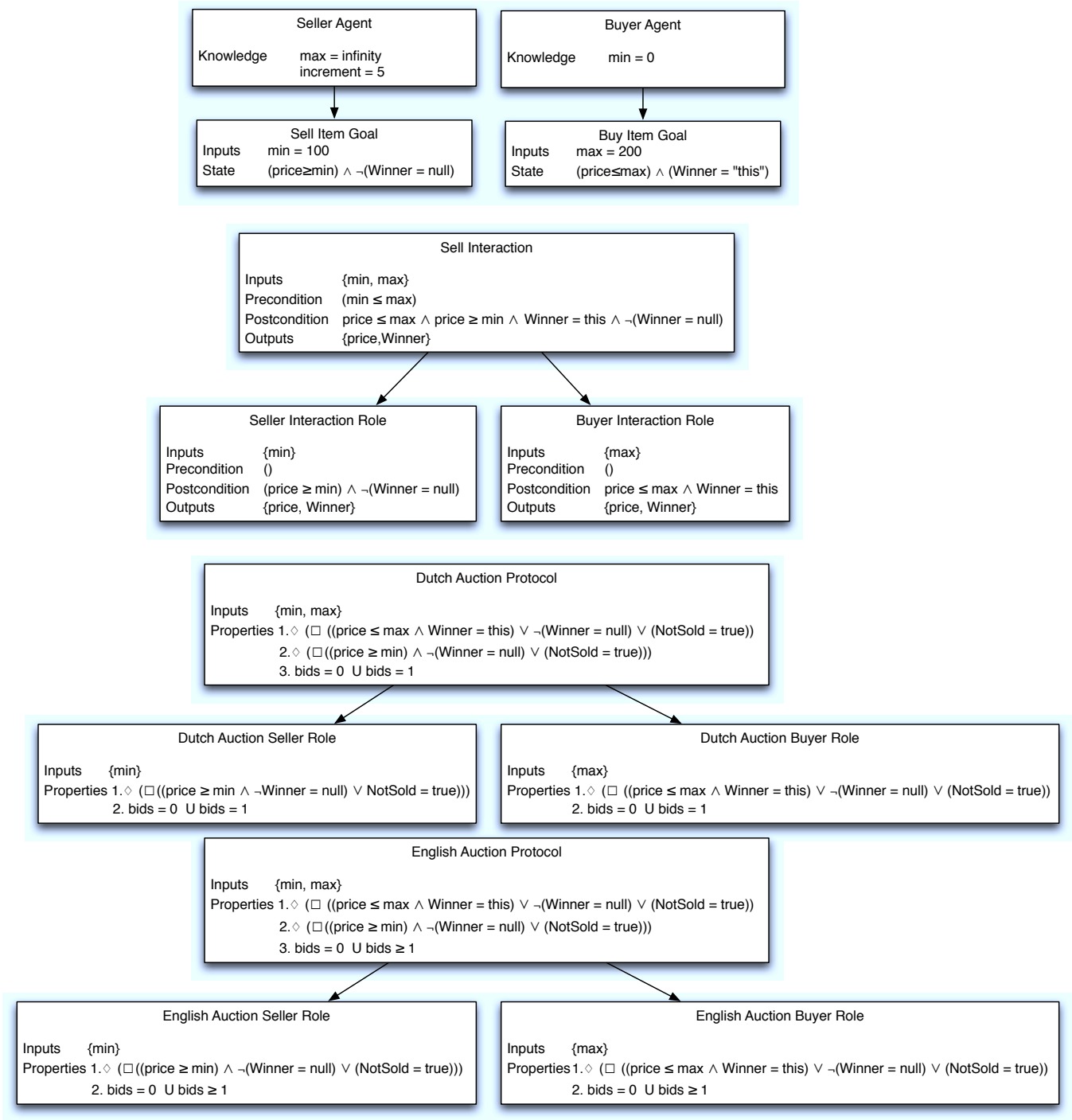


Figure 4.3: *Auction Example Specification*

Each of these binding can be created in 2 phases. In the first phase the input bindings are created. The input bindings are indicated by the downward red arrows. The second phase creates the output bindings. These bindings are indicated by the upward blue arrows. The Type II and Type III bindings can be further broken down into 2 stages. Figures 4.4 and 4.5 show those two additional stages. These stages separate the computation of the Agent-Goal-Interaction binding from the computation of the Interaction-Protocol binding. The Agent-Goal-Interaction bindings are computed using Algorithm 1. The Interaction-Protocol bindings are computed using Algorithm 8.

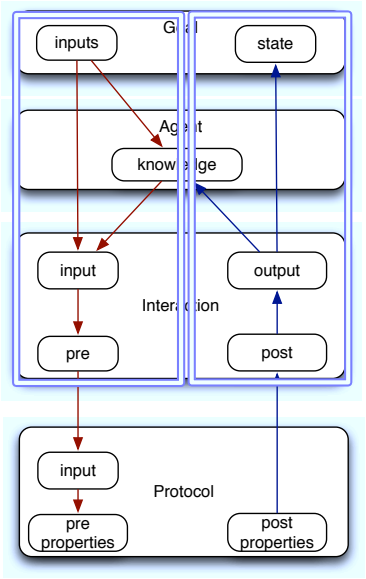


Figure 4.4: *Required Goal-Interaction Bindings*

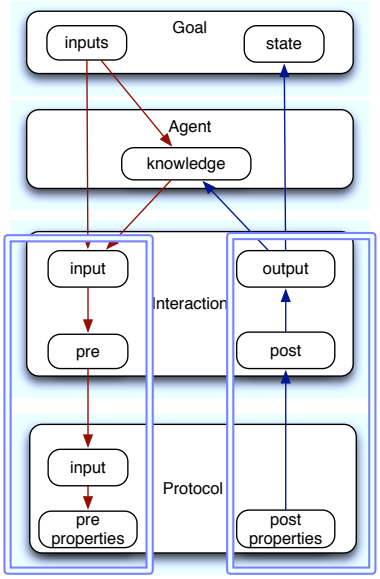


Figure 4.5: *Required Interaction-Protocol Bindings*

The first stage defined above does not include Protocols and Protocol Roles, thus the Partial Interaction Tuple excludes the Protocol and Protocol Role entities. Each Partial Interaction Tuple describes a group of entities such that if the Agent can participate in the Interaction by using the Interaction Role, then the Agent may or will (see Section 4.4) achieve its Goal. The Agent-Goal-Interaction Algorithm (Algorithm 1) takes an Agent as an argument and returns the set of possible *Partial Interaction Tuples* (Definition 11).

Table 4.1: *Input Mapping*

SellInteraction.min → {Seller Agent.max,Seller Item Goal.min}
SellInteraction.max → {Seller Agent.max,Seller Item Goal.min}

Definition 11 (*Partial Interaction Tuple*)

<goal,agent,interaction,interaction role>

Algorithm 1 takes an Agent a as an argument. The Seller Agent is used for example purposes. For each of the goals the algorithm computes the possible *Partial Interaction Tuples*. On Line 3 the algorithm iterates through the set of goals that the Agent possesses. The Seller Agent has a single goal, Sell Item Goal. Line 4 declares the set of valid input bindings. On Line 5 the algorithm iterates through all of the Interactions that the Agent possesses. The Seller Agent only has one interaction, the Sell Interaction, thus this is the Interaction used for the rest of the example. Line 6 calls the input binding function. This function is listed in Algorithm 2 and it returns a mapping of inputs to possible values (a set of VariablePairs). These values can originate either in the Agent's knowledge or in the Agent's Goal. The map contains an entry for each of the inputs in the Interaction. The Sell Interaction has two inputs, thus the map would have those two inputs as keys in the map (min and max). The values in the map are sets of VariablePairs. Each of these VariablePairs is from either the Agent's knowledge or the Agent's Goal. In the Auction example the variable *min* in the Sell Item Goal and the variable *max* in the agent have the same type (Integer) as the Sell Item Interactions' required inputs (min and max). Thus the map for this example would contain the entries listed in Table 4.1.

The Algorithm iterates through all the possible combinations of input bindings (Line 7) that can be generated from the Input Mapping. The set of combinations that are generated for the above example input mapping can be seen in Table 4.2. Each combination is a binding that must be checked by the algorithm.

Line 8 verifies that the Interaction precondition is valid. If the precondition for the Interaction is valid then the algorithm iterates through the Interaction Roles for that In-

Table 4.2: *Combination Example*

1.{<SellInteraction.min, Seller Agent.max>, <SellInteraction.max, Seller Agent.max >}
2.{<SellInteraction.min, Seller Agent.max>, <SellInteraction.max, Seller Item Goal.min>}
3.{<SellInteraction.min, Seller Agent.min>, <SellInteraction.max, Seller Agent.max >}
4.{<SellInteraction.min, Seller Agent.min>, <SellInteraction.max, Seller Item Goal.min>}

Table 4.3: *Valid Input Bindings*

1.{(<SellInteraction.min, Seller Agent.max>, <SellInteraction.max, Seller Agent.max >),	Sell Interaction,	Seller Interaction Role}
2.{(<SellInteraction.min, Seller Agent.max>, <SellInteraction.max, Seller Item Goal.min>),	Sell Interaction,	Seller Interaction Role}
3.{(<SellInteraction.min, Seller Agent.min>, <SellInteraction.max, Seller Agent.max >),	Sell Interaction,	Seller Interaction Role}
4.{(<SellInteraction.min, Seller Agent.min>, <SellInteraction.max, Seller Item Goal.min>),	Sell Interaction,	Seller Interaction Role}
5.{(<SellInteraction.min, Seller Agent.max>, <SellInteraction.max, Seller Agent.max >),	Sell Interaction,	Buyer Interaction Role}
6.{(<SellInteraction.min, Seller Agent.max>, <SellInteraction.max, Seller Item Goal.min>),	Sell Interaction,	Buyer Interaction Role}
7.{(<SellInteraction.min, Seller Agent.min>, <SellInteraction.max, Seller Agent.max >),	Sell Interaction,	Buyer Interaction Role}
8.{(<SellInteraction.min, Seller Agent.min>, <SellInteraction.max, Seller Item Goal.min>),	Sell Interaction,	Buyer Interaction Role}

teraction. Line 10 verifies the precondition of the interaction role given the input binding. If the precondition is valid then the triple (InputBinding, Interaction, InteractionRole) is added to the current set of valid binding triples. The *validBindings* variable stores the set of triples where the inputBinding is valid for both the Interaction and the Interaction Role. In the auction example, the precondition for the Interaction (Sell Item Interaction) is valid and the precondition for each of the Interaction Roles (Seller Interaction Role and Buyer Interaction Role) is valid. The algorithm iterates through each combination of bindings producing the eight tuples shown in Table 4.3.

The second half of the algorithm generates the output bindings for the Interaction-Goal bindings, shown on the right hand side of Figure 4.4. On Line 18 the algorithm iterates through the valid input bindings triples (Defined in Table 4.3). On Line 22 the output binding function is called. This function is defined in Algorithm 3. The result of this function call is the list of bindings shown in the Table 4.4. There are two variables in the Goal that are not listed as inputs: Price and Winner. Variables that are not listed as inputs default to being available for binding in the output. Each of these variables is mapped by the *getOutputBindings* function. The map returned is similar in structure to the input bindings (Line 6). The type for the Price in the Sell Item Goal only has one possible mapping to an output of the Sell Interaction (Price). This is the same case for the Winner Variable in the Sell Item Goal.

Table 4.4: *Valid Output Bindings*

1. Sell Item Goal.price ↦ {SellInteraction.price }
2. Sell Item Goal.Winner ↦ { SellInteraction.Winner}

Line 25 verifies that the *Interaction* postcondition implies the Goal state. Line 26 verifies that the *Interaction Role* postcondition implies the Goal state. If both conditions are true, then the tuple is added to the set of valid interaction tuples (Line 27). In the example the algorithm iterates through all eight lines in Table 4.3 and the algorithm checks each combination (only one is valid) of the possible valid output Bindings listed in Table 4.4.

4.3.2 Binding Functions

Understanding the algorithm that generates the binding map is essential to understanding how the Binding Algorithm works. The Input Binding function (Algorithm 2) takes an Agent, a Goal and an Interaction and returns a mapping from input variables to possible values. Line 3 starts by iterating through the set of inputs for the Interaction. For the Sell Interaction there the two inputs *min* and *max*. The algorithm looks up the type of each input (Line 5). The types for both min and max are floating point. The algorithm takes that type and looks up the set of bindable variables within the Agent knowledge (Line 6) and Goals inputs (Line 7). For the auction the bindable variables in the Seller Agent are listed in Table 4.5 and the bindable inputs for the Sell Item Goal are shown in Table 4.6. Line 8 checks to see if both sets are empty (no bindable variables). If so then the algorithm exits with a failure. If the sets are not empty then the algorithm maps each input to its possible values. When the algorithm has iterated through all the inputs, it then returns the mappings it has generated. The resulting map was shown in Table 4.1.

Table 4.5: *Seller Agent Variables*

1. typeOf(max) floating point
2. typeOf(increment) integer

The algorithm for the output pair binding is similar to the algorithm for the input binding

Algorithm 1 Getting Interaction Tuples

```
1: Set(PartialInteractionTuple) getTuples(Agent a)
2: Set(PartialInteractionTuple) result
3: for all Goal g ∈ a.getGoals() do
4:    List(Set(Bindings),Interaction, InteractionRole)) validBindings
5:    for all Interaction i ∈ a.getInteractions() do
6:       Map(VariablePair,Set(VariablePair)) inputBindings = getInputBindings(a,g,i)
7:       for all Set(Binding) inputBinding ∈ combinations(inputBindings) do
8:          if validPrecondition(a,g,i,inputBinding) then
9:             for all InteractionRole ir ∈ i.getInteractionRoles() do
10:               if validPrecondtion(a,g,ir,inputBinding) then
11:                  vaildBindings.add(inputBinding, i , ir)
12:               end if
13:            end for
14:         end if
15:      end for
16:   end for
17:   {Now we have valid input bindings, proceed to outputbindings}
18:   for all inputTuple ∈ validBindings do
19:      Interaction i = inputTuple.interaciton
20:      InteractionRole ir = inputTuple.interactionrole
21:      InputBinding ib = inputTuple.inputBinding
22:      Map(VariablePair,Set(VariablePair)) outputBindings = getOutputBindings(a,g,i)
23:      {Iterate through the output bindings}
24:      for all Set(Binding) ob ∈ combinations(outputBindings)  do
25:         if validPostcondition(a,g,i,ib,ob) then
26:            if validPostcondition(a,g,ir,ib,ob) then
27:               result.add(g,a,i,ir)
28:            end if
29:         end if
30:      end for
31:   end for
32: end for
33: return result
```

Algorithm 2 Getting the Input Binding Pairs for an Interaction

```
1: Map(VariablePair,Set(VariablePair)) getInputBindings(Agent a, Goal g, Interaction i)
2: Map(VariablePair,Set(VariablePair)) result
3: for all Variable input ∈ i.inputs do
4:    Set(Bindings) inputBindings
5:    Type inputType = input.getType();
6:    Set(Variable) AgentVariables = a.getKnowledgeVar(inputType)
7:    Set(Variable) GoalVariables = g.getInputVars(inputType)
8:    if AgentVariables.isEmpty() and GoalVariables.isEmpty() then
9:       exit(FAILURE)
10:   else
11:      result.put(input, AgentVariables ∪ GoalVariables)
12:   end if
13: end for
14: return result
```

generation. The only major difference is that on Line 3 (Algorithm 3) the algorithm iterates through the free variables and outputs in the Goal and not the inputs to the Interaction. The algorithm gets the value for the variable from the Agent's Knowledge (Line 6) and the output variables of the Interaction (Line 7).

Algorithm 3 Getting the Output Binding Pairs for an Interaction

```
1: Map(VariablePair,Set(VariablePair)) getOutputBindings(Agent a, Goal g, Interaction
   i)
2: Map(VariablePair,Set(VariablePair)) result
3: for all Variable output ∈ g.freeVariables do
4:    Set(Bindings) outputBindings
5:    Type outputType = output.getType();
6:    Set(Variable) AgentVariables = a.getKnowledgeVar(outputType)
7:    Set(Variable) InteractionVariables = i.getOutputVars(outputType)
8:    result.put(output, AgentVariables ∪ InteractionVariables)
9: end for
10: return result
```

These algorithms are defined to generate all possible type correct bindings. These pos-

Table 4.6: *Sell Item Goal Variables*

1.	typeOf(min)	floating point

sible bindings must be run through the algorithms in the next section to determine if the Interactions can achieve the Agents Goals.

4.4 Provability vs. Viability

Section 4.2 stated that there were two distinct proof modes: *Provable* and *Viable*. This section provides the intuition and the formalization of these concepts.

4.4.1 Proof Mode

In *Provable* mode, the algorithm *guarantees* that the Goal's state will be achieved if the Interaction is performed by the Agent. The only assumption that must be made is the assumption that the protocol operates without error. This algorithm is sound but the effect of using it may lead to one of two design approaches. In the first type of design the designer will create the goals such that they allow for the possibility of failure in the actions that may be performed by the Agents. This may lead to correct systems, but the overhead of designing such goals can be burdensome and the goals will be specific to the Interactions and Protocols defined for that system. The second type of design is one where there are no specified failure conditions within the Protocols. Unfortunately this is not realistic in a multiagent system, where Agents have a reasonable chance of failure. While the Provable mode does provide a satisfaction guarantee, it may not be the best choice in all situations. Clearly a more flexible algorithm is needed for realistic multiagent systems.

4.4.2 Viable Mode

The *Viable* mode provides the agent with the *possibility* that some execution path *may* lead to the achievement of the Agent's Goal. This means that there may be failure, and thus the Agent may need to repeat an Interaction or select some other Interaction, but the achievement is possible. This type of algorithm allows designers to create very specific Goals that are tailor made for an Agent. The designer can also have realistic, proven and

usable Protocols that work in theory and in practice. The Interactions provide a layer of abstraction between the Goal and the Protocol. Thus Interactions are more specific than the goals that they achieve, but they are less specific than the protocols that implement them.

4.4.3 Differences

The difference between the Provable mode and the Viable mode lies in the postcondition verification. For example, suppose that Protocol X has two properties as shown in Example 1.

Example 1 (*Protocol X properties*)

1. $(a \vee b)$
2. $(c \vee d)$

Now suppose that there is an Interaction Y with the postcondition as shown in Example 2.

Example 2 (*Interaction Y postcondition*)

$(a \wedge c)$

A proof to show that protocol X guarantees Interaction Y would yield the following proof obligation.

Example 3 (*Proof*)

$(a \vee b) \wedge (c \vee d) \implies (a \wedge c)$

Running the above equation through a SAT solver returns the result of satisfiable. By the definition of implication a result of satisfiable means that the implication does **not** hold. Thus the Provable mode states that by using Protocol X, the Interaction Y is not always satisfied.

The Viable mode algorithm takes the Protocol properties and converts them into disjunctive normal form (DNF). This conversion has the possibility of increasing the number of terms exponentially, but there are ways to combat some of those problems (by caching the results and pruning). A conversion of Protocol X's properties into disjunctive normal

form yields the condition listed in Example 4. The Viable algorithm checks to see if **any** of the clauses can used to prove Interaction Y's postcondition.

Example 4 (*Protocol X DNF*)

$(a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d)$

There are four clauses in the Disjunctive Normal form of the protocol's postcondition and the first clause $(a \wedge c)$ implies interaction Y's postcondition. This existence of a case where the postcondition implies the goal's state gives the algorithm proof that the goal's state *may* be achieved. Agents can then use the Interactions that may achieve their goals.

The Binding System does not require agents to use one mode or another. The agents can use the Provable mode for picking Interactions and Interaction Roles, and then use the Viable mode for selecting Protocols that implement those Interactions and Interaction Roles. The Viable mode was created to provide a more flexible system that can be used at both design time and at runtime.

4.5 Binding Goals, Interactions and Protocols

Section 4.3 defined an algorithm that enumerates the set of all possible type correct bindings from the Goal to the Interaction. This section defines an algorithm (Algorithm 4) that creates bindings from the Interaction to the Protocol (Figure 4.5). There are two major differences between the Interaction-Protocol binding and the Agent-Goal-Interaction binding. The first difference is that when the algorithm checks the precondition or the post condition, it must convert the protocol's set of LTL properties into a set of properties that either hold at the start of the protocol (precondition check) or at the end of the protocol (postcondition check). The set of rules used for this conversion are given in Section 3.4. The second major difference is that both Interactions and Protocols have compositional roles. To check for valid bindings, the algorithm has to iterate through all the possible combinations of Interaction Roles (Line 7) and Protocol Roles (Line 8). Other than the conversion of properties, this algorithm is similar to the algorithm for generating the Goal-Agent-Interaction bindings.

Definition 12 (*Interaction-Protocol Tuple*)

<Interaction, Protocol, Interaction Role, Protocol Role>

Algorithm 4 Getting Interaction-Protocol Tuples

```
1: Set(InteractionProtocolTuples) getProtocolInteractionTuples(Interaction i, Protocol p)

2: Set(InteractionProtocolTuples) result
3: Set(Bindings) inputBindings = getInputBindings(i, p)
4: Set(Set(Bindings), Interaction Role, ProtocolRole)) validBindings
5: for all Set(Bindings) inputBinding ∈ combinations(inputBindings) do
6:    if validPrecondition(inputBinding,i,p) then
7:       for all InteractionRole ir ∈ i.getInteractionRoles() do
8:          for all ProtocolRole pr ∈ p.getProtocolRoles() do
9:             if validPrecondition(inputbinding, ir,pr) then
10:               vaildBindings.add(inputBinding, ir, pr)
11:            end if
12:         end for
13:      end for
14:   end if
15: end for
16: for all inputTuple ∈ validBindings do
17:   InteractionRole ir = inputTuple.InteractionRole
18:   ProtocolRole pr = inputTuple.ProtocolRole
19:   Bindings inputBinding = inputTuple.inputBinding
20:   for all Set(Bindings) outputBindings ∈ combinations( getOutputBindings(i,p)) do
21:      if validPostcondition(i,p,inputBinding,outputBinding) then
22:         if validPostcondition(ir,pr,inputBinding,outputBinding) then
23:            result.add(i,p,ir,pr)
24:         end if
25:      end if
26:   end for
27: end for
28: return result
```

4.6 Substitutions and Proofs

The Viable mode splits the postconditions into multiple proofs that the algorithm has to check. This section defines the proof algorithm that is utilized by both the Provable mode and the Viable mode.

Bindings *are* functions that map a Variable in one entity to a Variable in another entity. The bindings are designed such that the output of one function can be used as an input to another binding. This property allows the AutoBinding System to compose the bindings. The AutoBinding system defines a binding that maps values from the Agent's Knowledge and the Goal to the Interaction inputs. This is a binding from the Interaction to Agent's Knowledge or a Goal's inputs. The values flow in the opposite direction of the bindings, from the Knowledge and the Goal into the Interaction. An example function is shown in Table 4.7. There is another binding that maps inputs from the Interaction to inputs of the Protocol (Table 4.8). The two bindings are composed to get the required inputs for the Interaction and the Protocol.

Table 4.7: *Goal-Knowledge-Interaction Input Function*

Sell Item Goal.min	$\rightarrow$	*Sell Interaction.min*
Seller Agent.max	$\rightarrow$	*Sell Interaction.max*

Table 4.8: *Interaction-Protocol Input Function*

Sell Interaction.min	$\rightarrow$	*Dutch Auction.min*
Sell Interaction.max	$\rightarrow$	*Dutch Auction.max*

The Auction example system is used to show how the proof model can be used to check bindings. The goal used in this proof demonstration is the *Sell Item Goal* (Equation 4.1) and the Interaction is *Sell Interaction* (Equation 4.2). Below are the formal specifications defined for the Goal and the Interaction. These specifications are also listed in Figure 4.3, but are repeated for the readers convenience.

$$(\text{price} \geq \text{min}) \wedge \neg(\text{winner} = \text{null}) \tag{4.1}$$

$$(\text{price} \geq \text{min}) \wedge \neg(\text{winner} = \text{null}) \wedge (\text{price} \leq \text{max}) \wedge (\text{winner} = \text{this}) \tag{4.2}$$

The algorithm must prove that if the *Sell Interaction* is performed, then its postcondition guarantees that the state for the *Sell Item Goal* will be achieved. This is done by testing

to see if the Interaction's postcondition implies the Goal's state. The algorithm must prove that the implication holds. This implication, or proof obligation is shown in Equation 4.3.

$$\begin{array}{ll} 1. & (\text{price} \geq \text{min}) \wedge \neg(\text{winner} = \text{null}) \wedge(\text{price} \leq \text{max}) \wedge (\text{winner} = \text{this}) \\ 2. & \implies \\ 3. & (\text{price} \geq \text{min}) \wedge \neg(\text{winner} = \text{null}) \end{array} \tag{4.3}$$

The algorithm takes both the proof obligation and the set of bindings. The bindings are used to relate or bind variables in one part of the proof to variables in another part of the proof. In Equation 4.3 the variable *price* on Line 1 refers to the price in the Sell Interaction, whereas the variable *price* on Line 3 refers to the price in the Sell Item Goal. Equation 4.4 defines the binding used in this example that identifies which variables are bound together. The algorithm takes the bindings and substitutes bound variables with fresh Variables. The fresh Variable for each binding is listed in the last column. A fresh Variable is a variable that does not exist in any of the specifications of any of the entities in the system. Fresh Variables are used to avoid variable capture and variable name collisions.

$$\begin{array}{llll} \textit{Sell Interaction.price} & \rightarrow & \textit{Sell Item Goal.price} & \text{0x1} \\ \textit{Sell Interaction.winner} & \rightarrow & \textit{Sell Item Goal.winner} & \text{0x2} \end{array} \tag{4.4}$$

The algorithm then substitutes each bound variable with its associated fresh variable as shown in Equation 4.5.

$$\begin{array}{l} (\text{0x1} \geq \text{min}) \wedge \neg(\text{0x2} = \text{null}) \wedge(\text{0x1} \leq \text{max}) \wedge (\text{0x2} = \text{this}) \\ \implies \\ (\text{0x1} \geq \text{min}) \wedge \neg(\text{0x2} = \text{null}) \end{array} \tag{4.5}$$

After the bound variables are substituted with their fresh variables, the algorithm substitutes the inputs from the input binding. In the example, *min* has a value of 100 and *max* has a value of ∞. The result of this substitution is shown in Equation 4.6.

$$\begin{array}{l} (\text{0x1} \geq 100) \wedge \neg(\text{0x2} = \text{null}) \wedge(\text{0x1} \leq \infty) \wedge (\text{0x2} = \text{this}) \\ \implies \\ (\text{0x1} \geq 100) \wedge \neg(\text{0x2} = \text{null}) \end{array} \tag{4.6}$$

The AutoBinding Algorithm generates several possible preconditions. The number of combinations is based on the number of variables that have the same type. In this example

the variables *min* and *max* have the same type. The algorithm checks each of these possible bindings. There is only one binding that proves to be correct. This binding was previously shown in Table 4.7. The algorithm takes the input values from the knowledge and goal and propagates them using the input binding function. The algorithm substitutes all input bindings that have a value and are not listed as outputs.

To solve our proof obligation, the AutoBinding algorithm runs it through a SAT solver to verify that the proof is either satisfiable or it is not satisfiable. The algorithm converts each Boolean expression into a Boolean variable. In this example characters are used to represent each Boolean expression. These substitutions are listed in Table 4.9 and the substituted proof obligation is shown in Equation 4.7.

Table 4.9: *Substitution Table*

0x1 $\geq$ 100	$\mapsto$	a
0x2 = null	$\mapsto$	b
0x1 $\leq$ inifinity	$\mapsto$	c
0x2 = this	$\mapsto$	d

$$(a) \wedge \neg(b) \wedge (c) \wedge (d) \implies (a) \wedge \neg(b) \tag{4.7}$$

After the algorithm has converted each of the Boolean expressions to a Boolean variable, it then runs the proof through the SAT solver. The algorithm proves that the state is true through the postcondition ($postcondition \implies state$) by showing the satisfiability of $postcondition \wedge \neg state$.

The conversion of the implication from Equation 4.7 is shown in Equation 4.8. The algorithm then converts the Boolean formula to Conjunctive Normal Form (CNF), which is the input format for a SAT solver. To prove that the implication holds, the algorithm checks to see if the formula is unsatisfiable (per the definition given in[98]). If the formula is satisfiable, then the implication does not hold and the Interaction will not achieve the Goal for the Agent.

$$(a) \wedge \neg(b) \wedge (c) \wedge (d) \wedge \neg(b) \wedge \neg((a) \wedge \neg(b)) \tag{4.8}$$

Given Equation 4.8 as input, the SAT solver returns an *UNSAT* result. Thus the *Seller Agent* can achieve the *Sell Item Goal* using the *Sell Interaction*, given the set of bindings in Table 4.7.

The results of running the entire auction example can be seen in Appendix D.1. These results are consistent with the intuition given in the design. There are four results that are verified at the finish of the algorithm. These results are listed below.

The Seller Agent can achieve the Sell Item Goal using the Sell Item Interaction playing the Sell Item Interaction Role. The Sell Item Interaction can be implemented by the Dutch Auction Protocol and the Sell Item Interaction Role is implemented by the Dutch Action Seller Protocol Role. The Interaction and Interaction Role can also be implemented by the English Auction Protocol using the English Auction Seller Protocol Role.

The Buyer Agent can achieve the Buy Item Goal using the Buy Item Interaction playing the Buy Item Interaction Role. The Buy Item Interaction can be implemented by the Dutch Auction Protocol and the Buy Item Interaction Role is implemented by the Dutch Action Buyer Protocol Role. The Interaction and Interaction Role can also be implemented by the English Auction Protocol using the English Auction Buyer Protocol Role.

4.7 Conclusion

This section has defined the FIA reasoning algorithms. These algorithms can be used to creating bindings between entities within MAIM. These algorithms generate type correct bindings based on a formal specifications defined in MAIM. The algorithms allow the designer to verify properties at design time. Finding problems at design time is much cheaper than finding them after the implementation has been created. The verification also provides agents with a flexible system that can meet the needs of the agent. Agents can adapt to changes in their capabilities and the addition new interactions and protocols. Agents can

learn new protocols and incorporate those protocols without having to be reprogrammed.

This section also defined the logic used for creating proofs, substituting bound variables and a process for verifying that a proof holds. All of this logic has been completely automated for use by system designers and agents within the system, as shown in Appendix D. The automated system is used in the demonstration given in the next section.

Chapter 5

FIA Demonstration

This chapter presents a physical system that demonstrates the use of FIA at runtime. The system was given both simulated and actual failures to show that FIA is both useful and robust. This chapter defines the methodology used to create this system, the configuration used to run the experiments, and presents the results that show how the system adapted to capability failures. Additionally, a pair of simulations were designed to show that FIA is more robust than typical multiagent systems.

5.1 Methodology

The process of integrating FIA into a multiagent system consists of three parts.

(I) Modeling the multiagent system using MAIM by defining Agents, Roles, Goals, Interactions and Protocols.

(II) Using the FIA algorithms for generating and verifying bindings.

(III) Demonstrating that the resulting multiagent system is more robust than typical multiagent systems, thus showing the usefulness of FIA.

5.1.1 Methodology Requirements

The configuration of the demonstration has to meet several requirements to show that the system is both useful and robust. First there must be at least two agents for any multiagent

system. Secondly, the system must be able to adapt to failures. This requires that the configuration has at least two Protocols. The final requirement is that the system must be able to fail. The existence of failure shows that the system can adapt to these failures.

5.1.2 Physical Demonstration System

There are two agents in this system. The first agent is on a laptop and the second agent is on a tablet running the Android operating system. Figure 5.1 depicts the configuration of the system. The tablet possesses many capabilities that the laptop does not have, such as a GPS, an accelerometer, a gyroscope and an ambient light sensor. There are three separate protocols that use two different communication mediums. The first medium is wireless communication (WiFi). The second communication medium is an optical system that uses the camera on each device to capture images. The images are then run through Zxing[7] (an open source library) to extract the encoded information. The information is encoded in images using the Quick Response (QR)[101] format. This format creates barcodes that are both compact and robust. QR codes store information in two dimensions (space efficient) and it uses Reed-Solomon codes for error correction (robustness).

Figure 5.1: *Demonstration Configuration*

This physical demonstration system provides three protocols that use the two different communication mediums. The first protocol uses wireless for sending and receiving on both the laptop and the tablet. The laptop wirelessly sends a request for the GPS location. The tablet receives the request via its wireless receiver. The tablet then sends back the GPS

location of the tablet via wireless to the laptop.

The second protocol uses only optical capabilities for sending and receiving for both devices. First, the laptop encodes the request for a GPS location into a QR code and displays it on the monitor (Line 9 Listing 5.1). The tablet uses the camera to take a picture as shown on Line 12 Listing 5.2. The tablet then processes the image looking for the request. Once it decodes a request, it then encodes the GPS location of the tablet into an image (Line 16 Listing 5.2). That image is displayed onto the tablet screen. The laptop uses its built-in camera to take pictures. The laptop tries to decode the the images from the camera looking for the encoded GPS location (Line 10 and 11 in Listing 5.1).

Listing 5.1: *Physical Demonstration QR Protocol Laptop Excerpt*

```
8 QR+WRITE
9 gps:0    #Send GPS Goal Request
10 QR+READ
11 Latitude = 39.12 Longitude = -96.71 #Recieve GPS Location
12 QR+ACK
13 QR+WAIT
14 QR+ACKTIMEOUT #FINISH
```

Listing 5.2: *Physical Demonstration QR Protocol Tablet Excerpt*

```
10 I/PHDCameraLog(10809): 1320966421536    READ
11 I/PHDCameraLog(10809): 1320966423910    WAIT
12 I/PHDCameraLog(10809): 1320966427913    READDONE
13 I/PHDCameraLog(10809): 1320966427933    gps:0 #Recieve Request for GPS
14 I/PHDCameraLog(10809): 1320966427933    PARSE!gps:0
15 I/PHDCameraLog(10809): 1320966427934    WRITE
16 I/PHDCameraLog(10809): 1320966427934    TEXT=Latitude = 39.12 Longitude = -96.71 #WRITE GPS Location
17 I/PHDCameraLog(10809): 1320966442923    WAIT
18 I/PHDCameraLog(10809): 1320966484936    TIMEOUT
19 I/PHDCameraLog(10809): 1320966484947    DONE #FINISH
```

The third protocol uses wireless send on the laptop, wireless receive on the tablet, optical send on the tablet, and optical receive on the laptop. The third protocol can be used when the other two protocols cannot be used due to partial capability failures (such as the wireless send on the tablet and the optical display on the laptop).

Two different methods for failure were developed for the physical demonstration system. The first method was to use actual hardware failure. This was done by turning off the WiFi on either device or by not pointing the cameras at the images to be scanned. The second method emulates capability failure through software. Messages sent via failed capabilities will not be received and thus the software emulates physical hardware failures.

5.1.3 Physical Demonstration Results

The running of this physical system allows the laptop and the tablet to interact with one another. The tablet has the ability to acquire its location from its GPS receiver. The laptop can acquire its location by interacting with the tablet. Information about the interactions between the agents is logged by both the laptop and the Android tablet. Android provides a logging facility named `LogCat` that logs messages on the tablet. This log can be streamed from the tablet to the computer when the tablet is connected to the laptop via a USB cable. Additionally, the tablet application presents the log as the application is running, so that the system can be debugged without the USB cable being connected to the tablet (shown in Figure 5.2). The desktop application uses a custom logging facility to log messages sent and received. The full logs are shown in Appendix D.

Figure 5.2: *Android Screen Shot*

The laptop possesses a model of the entire system, which is defined using MAIM. Figure 5.3 shows a high level MAIM model of the system. The laptop runs the FIA Autobinding algorithms on the model to obtain a list of valid interaction tuples. Listing 5.5 shows the results of running FIA's algorithms on the model for this system. This list of tuples enumerates the valid interactions, protocols, interaction roles and protocol roles that the laptop can use to communicate with the tablet.

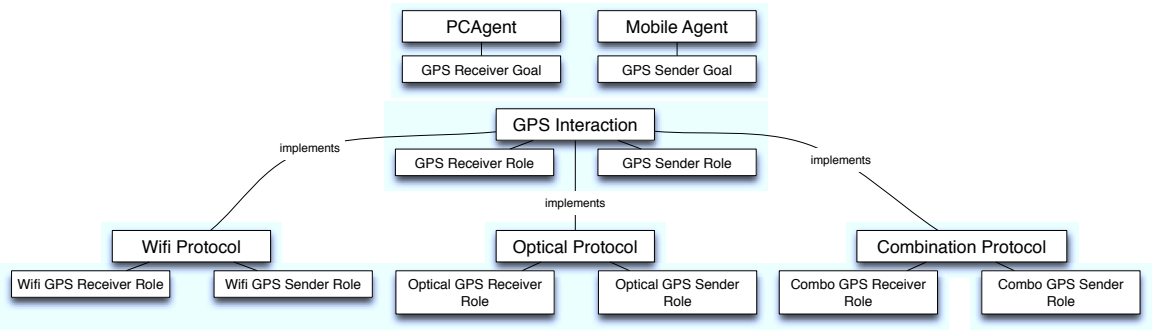


Figure 5.3: *Physical Demonstration Configuration Interaction Model*

Normal Operation

The laptop also has a list of capabilities required for each protocol. Without failure, the laptop uses each capability as designed. However, if the system is using software controlled failure then the laptop must verify which protocols it is capable of using. If the laptop does not have all of the required capabilities for a protocol, then that protocol is not used in the negotiation. When the laptop commences negotiation, it broadcasts its preferred protocols by sending their identifiers. Protocol identifier 0 is the optical protocol, identifier 1 is the WiFi protocol and identifier 2 is the combination protocol. So, if the laptop does not have the `Optical Read` capability, then it will not broadcast the identifier for the Optical Protocol (`0`). In Listing 5.3 the system has had no capability failure and thus all three protocols are broadcast. Line 5 in Listing 5.3 shows the laptop broadcasting the available protocols (`protocol: 0, 1, 2`). Listing 5.4, Line 4 shows the tablet receiving the broadcast from the laptop. Line 6 shows the tablet choosing protocol 0 from the list that it received. Line 7-14 in Listing 5.3 shows the laptop running the optical protocol and Lines 9-19 show the tablet running the protocol. After the optical protocol finishes, the two agents will again commence negotiating. The full listing of the simulation is listed in Listing D.1 and Listing D.2.

Listing 5.3: *Physical Demonstration 0 Laptop Log Excerpt*

```
BROADCAST!protocol:0,1,2
 Reading
SELECTED PROTOCOL!QRPROTOCOL
QR+WRITE
 gps:0    #Send GPS Goal Request
QR+READ
 Latitude = 39.12 Longitude = -96.71 #Recieve GPS Location
QR+ACK
QR+WAIT
QR+ACKTIMEOUT #FINISH
```

Listing 5.4: *Physical Demonstration 0 Tablet Log Excerpt*

```
I/PHDCameraLog(10809): 1320966388451     READ
I/PHDCameraLog(10809): 1320966389473     WAIT
I/PHDCameraLog(10809): 1320966406481     PROCESS
I/PHDCameraLog(10809): 1320966406501     DATA!protocol:0,1,2
I/PHDCameraLog(10809): 1320966406502     WRITE
I/PHDCameraLog(10809): 1320966406503     OUTPUT!0
I/PHDCameraLog(10809): 1320966407123     DONE
I/PHDCameraLog(10809): 1320966421521
I/PHDCameraLog(10809): Optical Protocol
I/PHDCameraLog(10809): 1320966421536     READ
I/PHDCameraLog(10809): 1320966423910     WAIT
I/PHDCameraLog(10809): 1320966427913     READDONE
I/PHDCameraLog(10809): 1320966427933     gps:0 #Recieve Request for GPS
I/PHDCameraLog(10809): 1320966427933     PARSE!gps:0
I/PHDCameraLog(10809): 1320966427934     WRITE
I/PHDCameraLog(10809): 1320966427934     TEXT=Latitude = 39.12 Longitude = -96.71 #WRITE GPS Location
I/PHDCameraLog(10809): 1320966442923     WAIT
I/PHDCameraLog(10809): 1320966484936     TIMEOUT
I/PHDCameraLog(10809): 1320966484947     DONE #FINISH
I/PHDCameraLog(10809): 1320966485085
```

Listing 5.5: *Physical Demonstration Computer Automatic Binding Computation Results*

```
1  Agent:PCAgent Goal:GPSGoal
2          I:GPSInteraction IR:GPSReciever
3          P:QRProtocol PR:QRReciever
4
5
6  Agent:MobileAgent Goal:GPSSender
7          I:GPSInteraction IR:GPSSender
8          P:QRProtocol PR:QRSender
9
10
11 Agent:PCAgent Goal:GPSGoal
12         I:GPSInteraction IR:GPSReciever
13         P:WIFIProtocol PR:WIFIReciever
14
15
16 Agent:MobileAgent Goal:GPSSender
17         I:GPSInteraction IR:GPSSender
18         P:WIFIProtocol PR:WIFISender
19
20
21 Agent:PCAgent Goal:GPSGoal
22         I:GPSInteraction IR:GPSReciever
23         P:WIFIOpticalProtocol PR:WIFIOpticalReciever
24
25
26 Agent:MobileAgent Goal:GPSSender
27         I:GPSInteraction IR:GPSSender
28         P:WIFIOpticalProtocol PR:WIFIOpticalSender
```

Adapting to Failure

Adapting to failure at runtime is the fundamental problem that FIA solves. This next example shows the system adapting to the failure of two critical capabilities. The full logs for this example are shown in Listing D.3 and Listing D.4. Listing 5.6 shows an excerpt of the log for the laptop and Listing 5.7 shows the log excerpt for the tablet. The tablet has a failure of the optical write and the WiFi read capabilities (Lines 59-60 in Listing 5.6). With this type of failure both the WiFi protocol and the optical protocols are not viable for communication, but the combined WiFi/optical (Combination) protocol is viable. This can be observed by looking at the Line 61 in Listing 5.6. The laptop only broadcasts protocol identifier 2 which corresponds to the Combination Protocol. The negotiation process allows the laptop to broadcast the set of valid protocols. After the tablet receives the broadcast, it then chooses one protocol from this list. In this case, the tablet receives a list with only one element in it (`protocol: 2`). This is shown in Listing 5.7 on Line 77.

Listing 5.6: *Demonstration 1 Laptop Log Excerpt*

```
59 Failed MobileQRWrite
60 Failed MobileWifiRead
61 BROADCAST!protocol:2
62 Reading
63 SELECTED PROTOCOL!PCQRMobileWIFI
64 OPTQR+WRITE
65 gps:0
66 OPTQR+READ
67 EMPTY
68 OPTQR+ACK
69 OPTQR+WAIT
70 OPTQR+ACKTIMEOUT
```

Listing 5.7: *Demonstration 1 Tablet Log Excerpt*

```
74 I/PHDCameraLog(16699): Negotiate
75 I/PHDCameraLog(16699): 1321985052999    READ
76 I/PHDCameraLog(16699): 1321985053973    WAIT
77 I/PHDCameraLog(16699): 1321985058007    DATA!protocol:2
78 I/PHDCameraLog(16699): 1321985058008    PROCESS
79 I/PHDCameraLog(16699): 1321985058008    WRITE
80 I/PHDCameraLog(16699): 1321985058049    OUTPUT!2
81 I/PHDCameraLog(16699): 1321985058786    DONE
82 I/PHDCameraLog(16699): 1321985065792
83 I/PHDCameraLog(16699): Comp Opt Read, Tablet Wifi Read Protocol
84 I/PHDCameraLog(16699): 1321985065832    READ
85 I/PHDCameraLog(16699): 1321985065910    WAIT
86 I/PHDCameraLog(16699): 1321985067844    READDONE
87 I/PHDCameraLog(16699): 1321985067867    gps:0
88 I/PHDCameraLog(16699): 1321985067868    PARSE!gps:0
89 I/PHDCameraLog(16699): 1321985067869    WRITE
90 I/PHDCameraLog(16699): 1321985067889    TEXT=EMPTY
91 I/PHDCameraLog(16699): 1321985082869    WAIT
```

The tablet then broadcasts that it chose protocol 2 (the Combination protocol) as shown on Line 80 in Listing 5.7. Lines 83-91 in Listing 5.7 show the tablet running the protocol while Lines 63-70 in Listing 5.6 show the laptop running the same protocol. After both devices have finished the protocol, the negotiation starts all over again and the laptop begins broadcasting the set of protocols that it wishes to use. This physical demonstration clearly shows that a real system can adapt to capability failures.

5.2 Simulated Demonstration System

To provide a more rigorous test of FIA, a simulated system was developed. This system was designed to test the reasoning and negotiation algorithms against all the possible ways that the system can fail.

5.2.1 Simulation Algorithm

The simulation algorithm developed to test FIA is outlined in Algorithm 5. The algorithm commences by generating all possible combinations of goals for each agent in the system (Line 1). This ensures that every possible combination of goals is simulated. The simulation iterates the number of failed capabilities from 0-4 (Line 2). Each protocol requires four of the eight capabilities. When there are five failures, none of the protocols can work and thus four failures was the limit used. Line 4 fails the capabilities in the system and Line 7 runs the negotiation process. When the system is finished running the negotiation, it checks to see if the negotiation was successful (Line 6) and, if is successful, runs the interaction. The system then records the result of the process (Line 9).

Algorithm 5 Simulation Algorithm

```
1:  for all agents : goalCombinations(Agent_1 ... Agent_n) do
2:    for failures = 0; failures ≤ 4; failures++  do
3:      for runs = 0; runs ≤ max; runs++ do
4:        model.failCapabilities(failures)
5:        Boolean success = runNegotiation(agents)
6:        if success then
7:          success = runInteraction(agents)
8:        end if
9:        recordResult(success)
10:     end for
11:   end for
12: end for
```

In the negotiation process, there is one agent that initiates the interaction. This agent uses its knowledge, current goals, interactions and protocols as inputs to the FIA algorithms. The algorithms return a list of interaction tuples that can achieve that agents goals. The Interaction tuples have the following structure.

```
<Interaction, Interaction Role, Protocol, Protocol Role>
```

The initiating agent then iterates through its list of known agents and tries to negotiate with each one. During the negotiation process, the initiating agent sends over its list of interaction tuples. The other agent then computes its own list of interaction tuples. If the

other agent finds a tuple that matches, it replies with the matching tuple. Otherwise the other agent replies with a negotiation failure. If the negotiation is successful, the initiating agent stops negotiating and agents commence interaction.

5.2.2 Simulated System

Figure 5.4 shows the configuration of the system used for this simulation. The setup for this system is similar to the setup for the previous physical demonstration. The `PC Agent` always initiates the negotiation with other agents in the system. The `PC Agent` can participate in the `GPS Interaction` and the `Sell Item Interaction`. The `GPS Interaction` is the same interaction that was used in Section 5.1.2. This interaction is implemented by three protocols: `WiFi Protocol`, `Optical Protocol` and `Combination Protocol`. The `Sell Item Interaction` is implemented by the `Dutch Auction Protocol`. The `PC Agent` has a `GPS Receiver Goal` and a `Sell Item Goal`. The `GPS Receiver Goal` can be achieved by using the `GPS Interaction` and the `Sell Item Goal` can be achieved by the `Sell Item Interaction`.

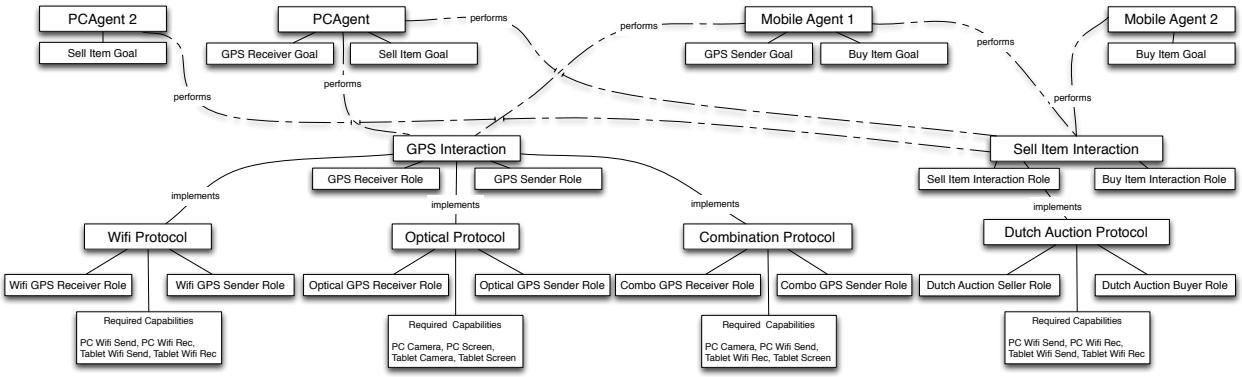


Figure 5.4: *Simulation Setup*

`Mobile Agent 1` has the `GPS Sender Goal` and the `Buy Item Goal`. The `GPS Sender Goal` can be achieved by the `GPS Interaction` and the `Buy Item Goal` can be achieved by the `Sell Item Interaction`. `Mobile Agent 2` has the `Buy Item Goal` which can be

achieved by the `Sell Item Interaction`. `PC Agent 2` has the `Sell Item Goal` which can be achieved by the `Sell Item Interaction`

Table 5.1 shows which goals are *complementary*, which occurs when an Interaction and a Protocol can achieve two goals at the same time (one goal for each agent). The columns represent the goals for the initiating agent (`PC Agent`). The rows show the other agents in the system and goals that they can have. In the table, `Mobile Agent 1` has two goals and thus there are two rows for that agent. One row shows the interactions for the `Buy Item Goal` and the other row for the `GPS Sender Goal`. The value in each cell denotes whether the goals in the row and column are complementary. There may be multiple interactions that enable the value to be true, but if the value is false, then there are **no** interactions that can yield success.

Table 5.1: *Interaction Success without Capability Failure*

		PC Agent's Goals	
Agent	**Goal**	GPS Receiver Goal	Sell Item Goal
Mobile Agent 1	Buy Item Goal	F	T
	GPS Sender Goal	T	F
Mobile Agent 2	Buy Item Goal	F	T
PC Agent 2	Sell Item Goal	F	F

When `PC Agent` begins negotiation, it first computes its set of interaction tuples that can achieve its goals. As described in Algorithm 5, the `PC Agent` iterates through its list of agents. The first agent in the list is the `Mobile Agent 1` and thus the `PC Agent` sends the list of interaction tuples to that agent. Once `Mobile Agent 1` receives this list, it computes its list of valid interaction tuples and selects the tuples that are valid for both it and the `PC Agent`. If there are no tuples that are valid for both agents then the negotiation fails. If there is at least one valid tuple then the negotiation will succeed and the agents will begin interacting with each other.

For example, suppose that the `PC Agent` has both the `GPS Receiver Goal` and the `Sell Item Goal` (Columns 1 & 2 in Table 5.1) and the `Tablet Camera` capability has failed. The set of valid tuples for the `PC Agent` is given in Table 5.2. The protocols that

require the `Tablet Camera` have been eliminated as they are no longer valid. The valid interaction tuples in Table 5.2 are sent to `Mobile Agent 1`, which in turn computes its list of tuples to achieve its goals. In the example `Mobile Agent 1` only has the `Buy Item Goal`. `Mobile Agent 1` proceeds by computing the list of valid interaction tuples given the current capability failures. The FIA computation algorithms, defined in Section 4, have been used to produce the results shown in Table 5.3. Tuple #3 in Table 5.2 is the only matching (or complementary) tuple. When the agents have found complementary goals, each agent looks up the set of roles that they should play in the interaction and the protocol. In this simulated system each agent will play one interaction role and one protocol role.

Table 5.2: *Valid Tuples For PC Agent when Tablet Camera fails*

1. <GPS Interaction, GPS Receiver Role, Wifi Protocol, Wifi GPS Receiver Role>
2. <GPS Interaction, GPS Receiver Role, Combination Protocol, Combo GPS Receiver Role>
3. <Sell Item Interaction, Sell Item Interaction Role, Dutch Auction Protocol, Dutch Auction Seller Role>

Table 5.3: *Valid Tuples For Mobile Agent 1 when Tablet Camera fails*

1. <Sell Item Interaction, Buy Item Interaction Role, Dutch Auction Protocol, Dutch Auction Buyer Role>

If the `PC Agent` fails to negotiate with `Mobile Agent 1`, then the `PC Agent` would try `Mobile Agent 2` and `PC Agent 2`. `PC Agent` would try and send the same interaction tuple to each agent, looking to interact with another agent.

5.2.3 Negotiation Failure

The previous section defined how the negotiation process works in the simulation. This section defines how the negotiation process can fail. There are two basic reasons a negotiation can fail: (I) there is a role mismatch or (II) the agents have no shared interactions. When the agents have no shared interactions (II) there are two possible reasons: (IIa) the agents do not have complementary goals or (IIb) they lack the required capabilities to participate in an interaction.

A *Role mismatch* (Failure I) occurs when two agents are trying to play the same role in the same interaction. In this system, this occurs when `PC Agent` is trying to achieve the `Sell Item Goal` and `PC Agent 2` is trying to achieve the `Sell Item Goal`. If these two agents try and negotiate they will both be able to use the `Sell Item Interaction` and the `Dutch Auction Protocol` but they will both try and play the `Sell Item Interaction Role` and the `Dutch Auction Seller Role`. This mismatch violates the property that all the roles must be played by an Agent in the system and thus this interaction will fail.

An example of agents with non-complementary goals (Failure IIa) is when the `PC Agent` has the `GPS Receiver Goal` and `Mobile Agent 1` has the `Buy Item Goal`. As shown in Table 5.1, there are no interactions that will allow these two agents to achieve their goals. In this case the `PC Agent` can only use the `GPS Interaction` to achieve its goal and the `Mobile Agent 1` can only use the `Sell Item Interaction` to achieve its goal. Clearly these two interactions are not the same, and thus these two agents cannot interact with one another to achieve their goals.

An example of capability failures (Failure IIb) is when `PC Agent` has the `Sell Item Goal` and `Mobile Agent 1` has the `Buy Item Goal` and it has lost the `Tablet WiFi Send` capability. While Table 5.1 indicates that interactions exist that can achieve the goals, without the `Tablet WiFi Send` there are no protocols that implement the `Sell Item Interaction`. Thus, there are no Interactions that can achieve `PC Agent's Sell Item Goal`.

This section has defined the three basic ways that the negotiation can fail in FIA. Knowing which capabilities have failed allows the agents to more accurately determine failures before they commence the protocols used in an interaction. Without this knowledge, the agent interactions would still fail, but they would fail during protocol execution. This failure may require the agents to go through an expensive error recovery process.

5.2.4 Simulating Systems without the Interaction Framework

To show how FIA compares to other multiagent systems, an additional set of simulations has been developed. These simulations emulate how most multiagent systems define interaction. These simulations use the same agents, goals and interactions, as described above, but each interaction only has one implementing protocol. The `Sell Interaction` still only has the `Dutch Auction Protocol` and the `GPS Interaction` is only implemented by the `WiFi Protocol`. The capabilities remain the same and the failures happen at the same rate.

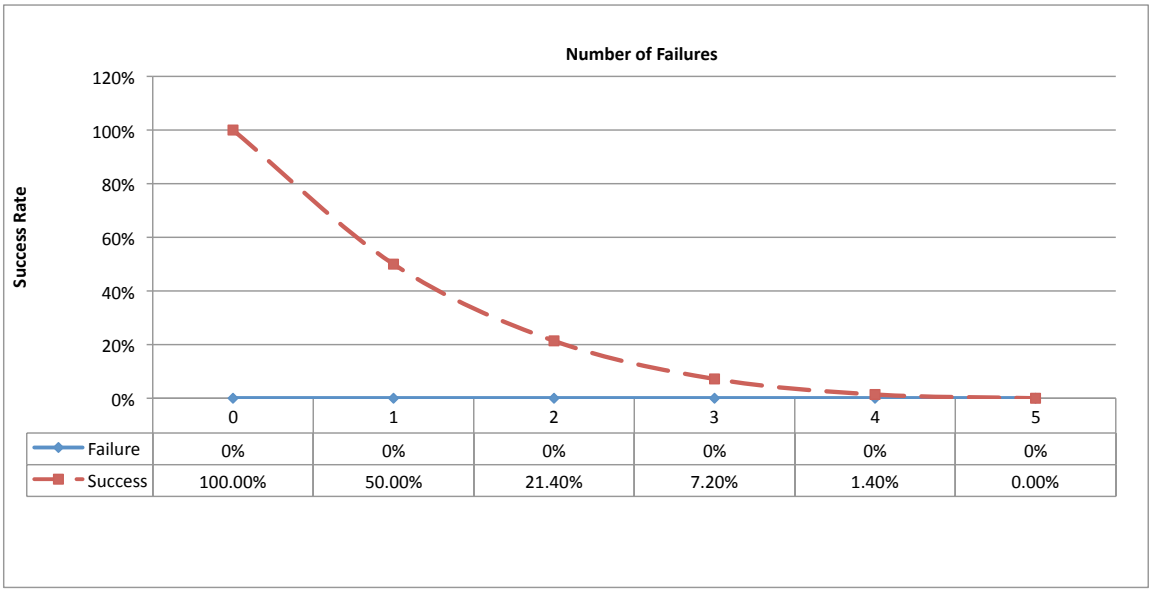


Figure 5.5: *No Framework Simulation Results*

The results of this simulation are summarized in Figure 5.5. This graph shows the system running without FIA. The x axis shows number of failures and the y axis shows the average success of the system. The system is successful if the the initiating agent successfully negotiates with another agent. The table at the bottom of the graph shows the actual success percentage at each point on the graph.

The first curve (**Failure**) shows what happens when the `PC Agent` cannot interact because no other agents have complementary goals. This situation occurs when the `PC Agent` has only the `GPS Receiver Goal` and `Mobile Agent 1` only has the `Buy Item Goal`.

Neither the `PC Agent 2` nor `Mobile Agent 2` has the ability to participate in the `GPS Interaction`, and thus the system fail no matter how many capabilities have failed. In this example, there are no agents that have the complementary goal for the `GPS Receiver Goal`.

The second curve (**Success**) shows what happens when there are other agents with complementary goals and the `PC Agent` can interact with those agents. If there are no capability failures, then the `PC Agent` *can* interact with another Agent to achieve its goals. There are eight capabilities in the system and each protocol requires exactly four of them. Thus, if there is a failure, there is a 50 percent chance that the `Dutch Auction Protocol` or the `WiFi Protocol` will fail. Table 5.4 highlights (with grey cells) the intersection of the goals for the agents in this configuration. Darker grey indicates no suitable interactions and the lighter grey indicates that interaction is possible. `Mobile Agent 1` is the only agent capable of interacting with the `PC Agent`. This is highlighted by the light grey coloring in Table 5.4. There is only one protocol for each interaction and both protocols require the WiFi capability and the four WiFi capabilities makeup half of the capabilities of the system. Thus, with one failure the system will fail 50 percent of the time (shown in Figure 5.5). As the number of failures rises, the likelihood of a protocol failure increases. For example, with three failures there is a 92 percent chance that the system will fail.

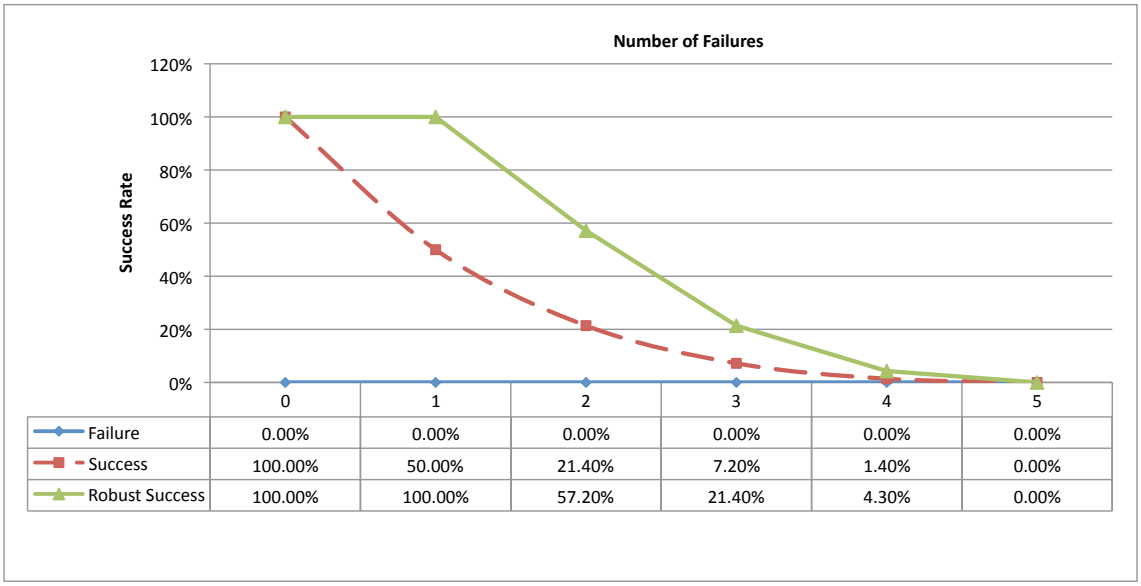
Table 5.4: *Partial Success Situation*

		PC Agent's Goals	
Agent	**Goal**	GPS Receiver Goal	Sell Item Goal
Mobile Agent 1	Buy Item Goal	F	T
	GPS Sender Goal	T	F
Mobile Agent 2	Buy Item Goal	F	T
PC Agent 2	Sell Item Goal	F	F

A system that does not use FIA clearly cannot handle the failure gracefully. The next section shows that the FIA degrades gracefully in the face of failures.

5.2.5 Simulating Results using FIA

The FIA simulation system designed has three protocols that implement the `GPS Interaction`. These protocols allow the system to be robust against capability failure. The results of this simulation are summarized in Figure 5.6. There are three different types of curves shown in this graph. **Failure** and **Success** are the curves from Figure 5.5.



	0	1	2	3	4	5
Failure	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Success	100.00%	50.00%	21.40%	7.20%	1.40%	0.00%
Robust Success	100.00%	100.00%	57.20%	21.40%	4.30%	0.00%

Figure 5.6: *Framework Simulation Results*

The third curve (**Robust Success**) shows how FIA can be used to adapt to capability failures. In the example given in the previous section (as shown in Table 5.4) the system fails 50 percent of the time. However, FIA allows the system to adapt and use protocols that has not been affected by the capability failure. In the example in the previous section, when `PC Agent` loses the `PC WiFi Send` capability, all interaction fails. In the system designed with FIA, the agents can use either the `Optical Protocol` or the `Combination Protocol`,which provides redundancy not found in the previous system. The graph shows that the FIA system does **not** fail if only a single capability fails. It should be noted that the graph for the FIA system is always greater than or equal to the graph for the system without FIA, which clearly shows that FIA is more robust than traditional systems. Using FIA, if the

system has 3 failures then there is only a 79 percent chance that the system will fail, which is much lower than the 92 percent chance that existed in the non-FIA system.

Adding more capabilities and protocols to the system will increase the ability of the system to adapt. For example, a system designer could add a bluetooth protocol that implements the `GPS Interaction`, which would allow the system to handle any two capability failures. A system designed with FIA allows the system designers to design a working system with a minimum number of protocols. As the agents acquire more capabilities, more protocols and interactions can be added to the system and the robustness of the system will increase.

5.3 Demonstration Conclusion

The systems presented in this chapter demonstrate how a designer can create a MAIM model and then use FIA to adapt to capability failure. The physical demonstration also shows that the FIA algorithms can be used to adapt to actual hardware failures at runtime, thus FIA produces a system that is more robust than tradition multiagent systems. The simulations show that FIA is more robust than a system designed without it. FIA had a success rate that was always better than or equal to the success rate of a system designed without the framework.

Chapter 6

Related Work

The Interaction Framework provides a runtime model that allows agents within a multiagent system to choose interactions that can achieve their goals. When an agent encounters failure, the framework provides a runtime model that allows agents to choose a new interaction. There are a variety of models, languages and frameworks that provide different facets of what FIA provides. These related systems are grouped into similar categories. Section 6.1 shows models that can be used at runtime. Section 6.2 shows methods for reasoning over protocols within multiagent systems. There are frameworks that generate multiagent systems protocols from specifications. These are shown in Section 6.3. Section 6.4 enumerates the multiagent system languages that support interaction among agents. Section 6.5 shows models and frameworks that allow system designers to verify that a multiagent system behaves as desired. FIA allows multiagent systems to be created that can adapt to failure. Section 6.6 shows a variety of systems that incorporate fault tolerance to create more robust systems. Section 6.7 describes ADL's and it catalogs several multiagent system metamodels.

6.1 Models at Runtime

Developing system models that can be used at both design and runtime is the ultimate Model Driven Engineering (MDE) process. A system design can be abstracted in such a manner that the design details will not overwhelm developers. A runtime component takes

the model and translates it into a language that can be interpreted or executed. In addition to ease of use, a runtime model can be adapted to new requirements on the fly, without the need for recompiling the entire system.

There are two main types of runtime models. The first type is a static model. Static models do not change over the course of a system or process running within the system. The second type of system is dynamic or self-adaptive model . Self-adaptive models can be modified at runtime on the fly.

6.1.1 Static Models

Static runtime models are good for applications where the requirements do not change rapidly over time. There are some programming languages that use a graphical model to describe the behavior of the program. Some examples are the Scratch programming language and LEGO MINDSTORMS Education NXT Software. These tools give users a model that they can edit and the changes to the model are directly fed into the system. Another example is the Java programming language, which supports dynamic class loading. This allows the model (or a Class) to change at runtime without having to recompile and restart the system.

6.1.2 Self-Adaptive Models

Self-adaptive models allow the running model to adapt to new requirements. Many self-adaptive systems follow the monitor-analyze-plan-execute (MAPE) model for adaptation. The goal of MAPE is to design"computing systems that can manage themselves" [64]. This provides additional flexibility over the static runtime models. Lubbers *et al.*[74] define a network model that adapts to changes in the requirements of applications. Their example application is of a device that uses a standard ethernet for communication. When the application needs more throughput or encryption the model can adapt such that the network provides these attributes. Thus the network model can change dynamically over the course of the systems lifetime. Another self-adaptive model is the Rainbow framework[53]. This

framework implements the MAPE methodology by defining an architectural layer which is an external control component. The architectural layer monitors the running system, proposes adaptations and then uses an adaptation executor to affect the system. The Rainbow architecture also provides a translation infrastructure to translate the commands given by the adaptation executor into the actual effectors of the running system. The separation of the system into layers allows the Rainbow architecture to adapt the behavior of both new and legacy systems. However the layering of components may lead to undesirable performance that may not be acceptable for systems with real time constraints.

6.1.3 Goal Models at Runtime

There have been several adaptive models that are goal oriented. Goals provide a level of abstraction that allows details to hidden. A goal model provides a convenient method of changing behavior at runtime. The background given in Section 2.5 gave a few multiagent systems that use goal models at runtime. Some of these included the GMoDS, OMACS and the Morandini *et al.* Goal Model[82].

Mylopoulos *et al.*[38] define an architecture that models requirements using the Tropos goal model. The system monitors the environment and when changes occur the system adapts to those changes. This architecture is similar in form to the Rainbow architecture. The self-reconfiguration component uses a *Context sensor* to read information about the system and the *Context actuator* and *Support system* modify the system at runtime. The authors added a *Contextual goal model manager* to handle updates to the goal model for the system. Their architecture allows the model to drive the system and when the environment changes the goal model reflects those changes.

Runtime Model Benefits

These runtime models provide an abstract manner of controlling and/or modifying the behavior of a system while it is running. FIA defines a runtime model that can dynamically modify the behavior of a system while it is running. This provides a mechanism for making

an agent that can adapt to both capability and environmental changes.

6.2 Protocol Reasoning Methods

Formalizing protocols allows protocol designers to predict the behavior that a protocol will exhibit. In addition to using tools that verify protocol properties, designers would like to have protocols that self configure, adapt to the needed requirements and run in an efficient manner. There have been several works that have solved different facets of this problem.

6.2.1 Commitments In Multiagent Systems

Event Calculus was created by Robert Kowalski and Marek Sergot in 1986[99]. Event calculus is a logic based formalism that models actions and the effects of those actions. Event calculus defines events using first order logic and temporal operators. The three main entities in event calculus are events, fluents and time points. Fluents are variables whose values change over time. Events modify the values of fluents over time. The calculus also defines a set of axioms that can be used to reason over the events, fluents and time. Figure 6.1 shows the model event calculus defines.

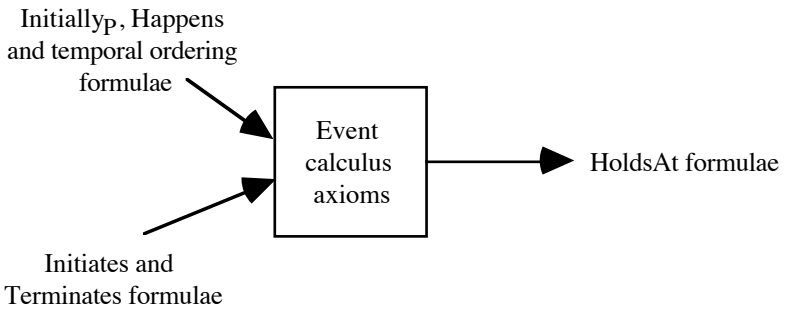


Figure 6.1: *Event Calculus Model*[99]

Systems designed using the event calculus model can derive any one of the three properties shown in Figure 6.1 given the other two properties. Event calculus allows a system to

reason about what actions the system needs to take to achieve a desired system state. Thus if an agent has an action then the agent can predict the future state of the system if the action were to be performed. Classical planning allows an agent to perform specific actions to realize a goal in the distant future.

Yolum and Singh[126] create an approach for specifying and executing protocols by using event calculus and commitments. Commitments are a social contract between two agents. An example of a commitment is $C(x, y, p)$, which means that x commits to y that x will make p true. The authors add axioms that relate actions of different agents. Protocols are specified by defining a set of protocol axioms. These axioms are defined for each action that an agent may take within that protocol. Protocols do not specify states and transitions and agents are allowed to enter execution at any point where the precondition to the protocol matches the current state. The authors define a Commitment Machine (CM) that can reason about commitments and protocols. Agents can use the CM to automatically execute protocols. In addition to executing a protocol, the system can skip unnecessary steps in the protocol if such steps have already been done or those steps do not apply to the current situation. The Commitment Machine lacks the ability to execute concurrent actions and the protocol must specify which axioms (or properties) are associated with each action within the protocol. This specification does not utilize the current state of the art protocol tools, such as SPIN to verify protocols, therefore a protocol designer is required to add and verify an additional specification to the protocol.

Winikoff[120] extends Event Calculus for use in multiagent systems. Protocols are modeled as fluents (or messages) and the protocol can specify the preconditions that are required to enact the fluents. The agent can then use its current state to decide which fluent is appropriate. This structure allows the protocol to be extended or short circuited based on the needs of the agents in the system. The commitment-based protocol does allow a protocol to be run based on the current state of the agent, but there is no automated method for leveraging properties of a protocol such that a choice of protocols that achieve a goal can

be enumerated.

Another runtime model for agents, goals and protocols is defined by Mylopoulos *et al.*[33]. This model is based on the Tropos multiagent model and it provides reasoning over protocols and goals. The reasoning can either be used *a priori* or at runtime. The authors specify protocol messages as commitments. The possible commitments and the agents goals are used to establish a path through the agents goal model. The authors extend their model[37] to allow agents in open systems to adapt at runtime to changes in the environment. This model allows agents to proactively detect and trigger adaptations. The model uses the agent's knowledge to trigger new goals for the agent to achieve. The triggers represent different strategies for dealing with commitment failures. Some example strategies were goal alternatives, redundant goals, and commitment redundancy.

Commitment drawbacks

There are three major differences between the commitment based models and FIA. The first is that they require the protocols to be specified in terms of commitments, whereas FIA uses the standard LTL protocol properties. This means that FIA can leverage current protocol tools. The commitment based models require additional specification. The second difference is that the commitment based models focus on the commitment between a group of agents. This commitment must be backed by some social structure for disciplining agents that do not adhere to the commitments they have entered. FIA does not put such a restriction on multiagent systems that use the framework. The third difference is that FIA helps agent decide which interactions are appropriate for a goal. The Commitment based approach reasons over protocols at the level of sending messages, not the selection of interactions.

6.2.2 Sharing State and Intent within Multiagent Systems

Agents in multiagent systems are typically seen as autonomous. Shared state allows a group of agents to view a portion of the mental state (beliefs of agents) of other agents within the group. The extra knowledge allows agents to reason about actions within that group, given

the shared state.

The Joint Intentions Theory[73] was created by Levesque *et al.*. The Joint Intentions theory describes both the internal state and the shared state (or Intention) of a group of agents. The Joint Intentions Theory describes a formal language for specifying joint intentions. This language includes first order, temporal and equality operators. The Joint Intentions theory describes the shared belief that agents must have in order for them to jointly act together. This theory by itself does not provide a very useful multiagent system; it simply defines the theory behind Joint Intentions.

The Joint Action Model[68] extends the Joint Intentions theory. In this model Kumar *et al.* formalize the states within a protocol and then reason about those states and landmarks within the protocols. They define that a landmark "represent[s] the state of affairs that must be brought about during the goal-directed execution of a protocol." The landmark based approach requires that each protocol is tagged with landmarks, which in turn are labeled as either required or optional. These landmarks allow protocols to be short circuited, but using landmarks requires additional specification. Each agent in the multiagent system must use and support the landmark based protocols for the protocols to work as designed.

The STAPLE language[67] allows agent designers to include Joint Intentions in a multiagent system. STAPLE uses the formal joint intentions theory to derive provably correct communication. STAPLE also provides a runtime model that can automatically execute protocols. Similar to FIA, STAPLE requires each action to have a specification. But STAPLE also requires that there must be executable code that implements that specification.

Joint Intentions Shortfalls

The Joint Intentions theory and Joint Action theory based systems require a shared state. This shared state may not be supported by agents in a heterogeneous multiagent system. The landmark base model also requires that every agent in the multiagent system supports protocol landmarks. The STAPLE runtime model requires access to executable code. FIA works at a higher level of abstraction and thus it does not have any of these requirements.

6.3 Multiagent System Protocols from Specification

The creation of protocols for heterogeneous multiagent systems is not a trivial task. There are multiple avenues for errors and failure. Implementations of low level protocols may be different on various operating systems. Additionally, debugging protocols is difficult due to latency, dropped and delayed traffic. These minor differences add extra complexity to the difficult task of creating correct and flexible multiagent systems.

The Agent Unified Modeling Language (AUML) defines a set of specification diagrams that allow multiagent system designers to graphically design interaction protocols. Odell *et. al.*[84] describe a way to design interactions (or protocols) for multiagent systems. They define the AUML standard for designing protocols and interactions at a high level. The diagrams allow designers to compose protocols and specify protocols as sequence diagrams. They define templates for protocols that can be decomposed into more detailed diagrams. These diagrams can be decomposed to the point that code can be generated from them. The sequence diagrams are parameterized and they include constraints on permissible communicative acts. FIA allows protocols to be parameterized and protocols include temporal specifications, but the details of the actual communicative acts are not required. In addition to specifying protocols, AUML allows designers to specify collaboration diagrams. These diagrams provide a high level overview of the interactions within a multiagent system.

Petri nets are a formal modeling language for creating distributed computer systems. A Petri net consists of a graph (nodes and edges), transitions and tokens. Petri nets allow concurrent execution, which is well suited for designing multiagent systems. Mazouzi *et al.*[76] defined a method of translating AUML protocol diagrams into a Petri net (specifically a Colored Petri net[CPN]) that can implement a protocol within a multiagent system. Their formalism also allows protocols to be used as actions. When this occurs, they require that the protocol includes a set of parameters, preconditions and postconditions. These three specifications are similar to the specifications required for FIA, although the conditions are calculated in FIA. The translation of the AUML diagrams into the CPN is *not* an automated

process, thus there is required human intervention to create CPN from the specification.

Benefits

The creation of protocols from specifications is a useful tool. Developers can use both these automated tools for creating protocols in multiagent systems, but FIA allows a more robust and flexible mechanism than these systems can provide.

6.4 Multiagent Systems Languages

There have been a variety of programming languages devoted to multiagent systems. Many of these languages explicitly define communication through language constructs. Using a language that supports communication makes the task of creating a multiagent system easier.

The Knowledge and Query and Manipulation Language (KQML)[49] attempts to define a method of transferring knowledge in a multiagent system. Finin *et al.* envisioned KQML as a language that agent-based systems "spoke" when they were interacting. The authors acknowledge the problems of sharing semantics, ontologies and knowledge, but they did not efficiently address them. For example, they use the Knowledge Information Format (KIF) as a format for exchanging knowledge. Understanding knowledge in KIF requires a shared ontology among agents. However, current state of the art agents do not have any method for *understanding* an ontology. KQML may have become more prevalent if advances in artificial intelligence had led to agents with true reasoning capabilities.

One of the first agent based programming languages was AgentTalk[69]. AgentTalk defines a *coordination process* as sequence of messages and a *coordination protocol* as a high-level protocol. AgentTalk is not a formal language, but it merely describes the interactions through protocols. *Scripts* are the implementation of protocols. Scripts are state diagrams with guarded transitions. Agents are allowed to reuse and dynamically select scripts at runtime.

The COOL[21] programming language came out about the same time as AgentTalk. COOL defines three levels of interaction (i) information content as specified by a language like KIF, a language for expressing (ii) intentions (such as KQML) and the (iii) conventions that the agents with in the interaction adhere to. The coordination model that COOL uses is a Finite State Machine representation with a set of specialized interaction rules.

The Interaction Oriented Model by Textual representation (IOM/T) is description language for specifying multiagent interactions[45]. IOM/T defines formal semantics for AUML sequence diagrams. The formal AUML semantics allows an IOM/T model to be created from an AUML diagram. IOM/T defines interaction based control structures to make understanding and management of interactions a more manageable task.

Each one of these models provides the programmer with a tool that can aid in the development in a multiagent system. Models allow programmers and designers to create multiagent systems quickly and correctly. However, there are three major downfalls to defining interactions at the language level. The first is that if the interaction falls outside of the model that the language defines, then the programmer must come up with an ad-hoc design to implement the interaction. The ad-hoc design leads at a variety of problems, such as debugging issues and maintenance. The second major problem is that an interaction definition at the language level requires a homogenous multiagent system. In a heterogeneous multiagent system the benefits of language level interactions shall be lost. The last problem is that these languages do not provide any framework for choosing a protocol or interaction at runtime.

6.5 Verifying Agent Behavior

To address some of the shortfalls of low level multiagent system languages, new models have been devised. These models provide a higher level of abstraction and the allow the behavior of a multiagent system to be verified.

The MABLE programming Language is a computer language designed for creating mul-

tiagent systems that support the Belief, Desire and Intention (BDI) paradigm. *"MABLE is essentially a conventional imperative programming language, enriched by constructs from the agent-oriented programming paradigm."*[122] The MABLE language is enriched by external actions and constructs that support beliefs and desires.

```
if (Bel x j > 5 ) then x := x-1
  else x := x+1 unsure x:= 0
```

The example above the shows some of the extra constructs, such as `Bel` and `unsure`. The belief `Bel x j > 5` tests if the current agent believes that Agent j believes that `x > 5`. Designers can specify properties that should hold for a MABLE program by using the MORA specification language. These properties can be then verified by a model checker (such as SPIN). A multiagent system designed in MABLE is limited to a fixed set of agents and MABLE restricts the agent's autonomy. The designers of MABLE plan to implement a Java compiler in future work, which would automate the process of creating a multiagent system.

The MaSE methodology[40] allows system designers to specify tasks that an agent performs as state diagrams. In the MaSE methodology a protocol is implemented by two or more tasks. DeLoach and Lacy define a systematic model for translating agent tasks into Promela models that can be verified by SPIN[71]. This approach eliminates the need for human translation of tasks and thus the task can be verified directly. This process thus reduces errors in translation and reduces the time required to verify a multiagent system. The authors extended the ability of the tool (agentTool)[70], by incorporating the verification into the user interface. This incorporation allows designers to more quickly locate and fix design time errors.

Wang *et al.* use both a goal model and logs of the system to verify that the requirements of the system match the actual running of the system[117]. The system is monitored to verify that pre and post conditions in the running system are the same as the requirements given in the goal model. If inconsistencies are found, the monitoring system adapts and monitors

the incorrect components in a more fine grained manner. A SAT solver is used to verify that the multiagent system meets its requirements.

GroupLog is a coordination language for multiagent systems[20]. GroupLog is defined in terms of "logic based programming abstractions". GroupLog uses Extended Horn Clauses (EHC) as representation of atoms and state. An example EHC clause is shown below.

```
obj(S) :: mess(M) :- method(M) | obj(NewS) :: true
```

In this example the EHC allows GroupLog to handle concurrency of multiagent systems. GroupLog reasons on agents, groups and goals. In GroupLog a group is a dynamic set of agents that share a goal. GroupLog is able to formally reason about actions and the consequences of actions for both an agent and the agents groups.

These models suffer from many of the problems of their predecessors, such as language or framework dependence. Each model allows a designer to verify a model where the agents are heterogeneous and all agents use the same language or framework. Additionally these languages do not aid developers to creating systems that can tolerate and adapt to failure. FIA allows agents outside the framework bounds to utilize the framework.

6.6 Fault Tolerant Systems

Verifying the behavior of multiagent systems is only the first step in creating robust multiagent systems. Adapting to failure is the second major step. The novel part of FIA is the ability to create multiagent systems that can both choose interactions and adapt to interaction failure. Fault tolerance allows systems that have failed partially to recover and possibly continue without interruption. Fault tolerant systems fall into one of two broad categories: (i) replication and (ii) checkpointing[85]. Replication uses redundant systems, agents, or services to ensure that if one goes down there is a replica to compensate for the failure. In checkpointing, the system stores known good states and, when a failure occurs, rolls back to a known good state.

6.6.1 Fault-Tolerant Protocols

The Fault-Tolerant Multi-Agent Development framework (FATMAD) uses checkpoints and recovery techniques to adapt to failure[116]. FATMAD is specifically targeted toward and built on top of the Jade platform. FATMAD provides protocol designers with tools for checkpointing and recovery. After protocol designers have created such protocols, the application designers can use the fault tolerant protocols in their multiagent systems.

FATMAD is designed for developing a multiagent system that uses a single platform (Jade). A multiagent system that uses FATMAD will be more fault tolerant, but there is no automated manner for selecting protocols based on the agent's needs (goals) or abilities (capabilities). Thus agents whose capabilities fail will not be able to use FATMAD to adapt to such failures.

The Fault Tolerant Recovery Agent System (FRASystem) is designed to provide fault tolerance to multiagent systems[72]. FRASystem is similar to the FATMAD framework in the functionality that it provides, however FRASystem does not rely upon a particular framework. A FRASystem requires four different types of agents that provide the rollback and the recovery. There must be a recovery agent, information agent, facilitator agent and garbage collection agent. Each process contains each of these agents, and each agent plays a role that ensures that the system is tolerant to faults. FRASystem is not a multiagent system methodology, but it uses a multiagent system to support distributed computing.

Quenum *et al.* introduced a protocol framework that allows agents to dynamically choose interactions[92]. Their framework allows agents to begin interacting without agreeing on a specific protocol. Each agent monitors the performatives, message structure and message order to determine which protocols the other agent could be using. If there is more than one protocol with the same pattern, the agent guesses which protocol is being used. If the guess is wrong, then the agents can roll back the protocol to a known good state.

There are two major downfalls to the FATMAD, FRASystem and Quenum approaches. First is that *all* actions must be reversible. Clearly there are actions in multiagent systems

that are not reversible (destroying a mine, dropping a victim or moving off a cliff). The second problem is that each protocol requires strict monitoring of messages and possible rollback. This overhead may be acceptable for an ad-hoc group of agents, but in an efficient multiagent system this may not be acceptable.

6.6.2 Fault-Tolerance Framework

The Fault-Tolerance Framework is designed for mobile, heterogeneous multiagent systems[85]. The Fault-Tolerance Framework relies upon a single checkpointing agent and a fault-tolerant protocol that communicates checkpoints from the distributed agents to the checkpointing agent. When a checkpoint occurs, the agent stops executing and saves its *entire* state. The agent then communicates the saved state and resumes executing. The checkpointing that the framework requires allows software agents to migrate across the network. Their model is designed for mobile platforms which is well tailored for agents with small memory footprints. This may be impractical for heavyweight mobile systems that have large memory footprints, but low bandwidth.

6.6.3 Hermes

Hermes[32] is a goal oriented interaction model. The Hermes model defines an Interaction Goal hierarchy, a set of actions and a set of constraints. The messages that are sent are an emergent behavior that rises out of the agents playing interaction roles. This goal hierarchy allows designers to place temporal constraints on the goal model. The Interaction Goal Hierarchy provides goals that each agent wishes to achieve and the agents use the actions to achieve those goals. The actions are discrete steps, akin to FIA's Atomic Actions. The agents in Hermes use the actions to achieve leaf Interaction Goals. In addition to the Interaction Goal model, they also have an action model. The action model defines roles and the suitable actions, decisions and alternative paths of execution. Hermes also defines a failure handling system. This system allows Hermes to handle action failures and goal failures. Hermes allows Interaction Goals to be canceled or rolled back in the event of a

goal failure. This failure mechanism requires the designer to specify the details of how and when a goal can be rolled back.

Unlike FIA, Hermes does not leverage the power of model checking to verify protocols. The Hermes model requires goal designers to redesign a protocol in a non-message centric model, but does not provide the current state of the art proof tools that are available to current protocol designers. Hermes also requires goal specific error handling.

Each of the fault tolerant frameworks provides the designer with a process to adapt to failure within a multiagent system. None of the fault tolerant systems use formal methods for choosing interactions. Therefore FIA can be more flexible when faults occur, while at the same time FIA ensures that agents select an appropriate interaction to achieve their goals.

6.7 ADL and Meta models

Architecture Description Languages (ADL) and meta-models provide mechanisms for creating models that can be tailored to the requirements of a particular domain. These models allow designers to use a common language for defining systems. The common language allows new developers to quickly understand a systems architecture.

6.7.1 Architecture Description Languages

Architecture Description Languages (ADL) allows the architecture of a system to be explicitly defined. There are a variety of ADL's that have been defined over the years, including AADL[48], C2 architecture[108] and Acme[54].

The Architecture Analysis & Design Language (AADL) was originally designed for avionics systems. It is currently used to model both the hardware and software for embedded and realtime systems. The Society of Automotive Engineers (SAE)[111] has defined standard AADL for use in modeling and creating complex systems in automobiles. As AADL's model both hardware and software, they can be used to generate the software used to control the

hardware.

The C2 architecture[108] was developed to address the specification of User Interface (UI) components. C2 is useful for defining systems that contain components that can and should be reused. Message based communication allows the individual comments to be wired together to create a computer system. C2 defines a graphical model that allows the architecture of the system to be visualized.

The Acme[54] architecture, like all ADLs, defines a model for representing the structure of a computer system. Acme defines the structure of a system using the entities such as *components*, *connectors* and *roles*. Acme also defines abstract properties about behavior, including nonfunctional behavior, system constraints and architectural style.

The above ADLs describe formal models for designing software systems. These models provide a formal basis for designing, verifying and creating such systems. However, these models cannot be applied to the multiagent paradigm due to the assumptions that these ADL's make. For example most ADL's assume that a component has a set of ports that can always be used for communication. Multiagent systems have the added complexity of failure that can block communication from occurring.

6.7.2 Meta-models

A meta-model is a model that models how domain specific models should be created. Each model for multiagent systems provides a different set of entities that can be modeled. The meta-model defines the properties that a domain specific model should contain. For example a meta-model for multiagent systems should, at the minimum, include the definition of an agent. Three different meta-models are compared: ADELFE, GAIA and PASSI.

ADELFE[24] is a model that model that defines agents within a cooperative environment. ADELFE models the different attributes that an agent can possess, like aptitudes, communication ability and skills. ADELFE makes the argument that a solution to any " *functionally adequate system*" can be defined using a cooperative multiagent system. ADELFE defines

a set of Cooperative rules that define how agents should try to turn Non-Cooperative Situations (NCS) into cooperative situations. ADELFE's meta-model is defined in terms of a Software Process Engineering Meta-Model. This allows designers to create a tailored process for creating a multiagent system.

The GAIA model defines a meta-model that includes organizational structures. These structures include Organizational Rules, Communication systems (protocols) and agent roles and responsibilities. GAIA provides these structures to constrain the behavior (hopefully to only good behaviors) of the multiagent system.

PASSI[36] defines a meta-model that allow practitioners to bridge the gap between traditional software systems and multiagent system designs. The PASSI meta-model aims to allow software engineers to use off the shelf software as components within an Agent. The PASSI agent then provides the multiagent system with services and the PASSI model defines communication protocols through sequence diagrams.

Bernon *et al.*[23] proposed a meta-model that combines the above meta-models into a unified meta-model. Their analysis suggests that any agent-oriented meta-model should have Agents and Interactions (defined as Protocols). Goals and Roles can be used in such meta-models, but are not required. MAIM defines entities that are consistent with widely accepted multiagent meta-model concepts. Therefore it should be relatively easy to create a SPEM process for the FIA and MAIM.

Beydoun *et al.* defined FAML[25], which is another meta-model for multiagent systems. This meta-model does not combine all the aspects of other meta-model to create a model that contains all other models. Instead, it selects the model components that can be used in almost any multiagent system. This process yields a much smaller meta-model and allows each methodology to create methodology-specific features (such as ADELFE's adaptive agents). FAML consists of four different meta-model definitions that define the aspects of the system that are internal or external to the agent and includes both design time and a runtime components. FAML defines the central concept of an Agent. It defines

Communication (Protocols), Actions, Agent Beliefs (Agent Knowledge) and Agent Goals. They do not include the concept on an Interaction, but Interactions can be seen as a FIA specific feature.

6.8 Summary

This chapter has shown a variety of models, frameworks and languages that provide similar functionality to FIA. None of these works incorporate the formality and the flexibility that FIA provides. FIA provides a formal model that allows agents to independently choose the interactions that best fit their goals. FIA can be modeled using the meta-models applicable to multiagent systems. When a failure occurs in an Interaction, an agent can then use the runtime model to choose a new Interaction that meets its needs.

Chapter 7

Conclusion

7.1 Current State of Interactions in Multiagent Systems

The multiagent paradigm allows designers to model agents within an interconnected distributed system. Agents have the ability to reason about themselves, their environment and their position with their organization. Multiagent systems are both distributed and interconnected and thus multiagent systems rely on communication protocols to exchange information between different agents in the system. Computer scientists have formalized communication protocols for quite some time. Multiagent systems designers can use these formal tools and methods to design communication protocols. These formal tools and methods allow designers to verify the correctness of protocols, using tools like Promela and SPIN. However, most goal-oriented multiagent systems do not provide a formal model for selecting protocols and, the ones that do, have a variety of restrictions on how those protocols can be selected. Additionally, many goal-oriented multiagent systems operate in environments where the likelihood of failure is far greater than in a computer lab. This means that most goal-oriented multiagent systems statically define which protocols are used and even though capabilities the protocols use are prone to failure. A formal model for interaction would allow tools and algorithms to be created that could reason about the situations where protocols and capabilities fail. This reasoning would allow agents to select

the appropriate interactions that can meet their goals.

Having a system that can determine which interactions can achieve an agent's goals is a powerful concept. Most goal-oriented multiagent systems require a direct mapping between goals and the actions that can be used to achieve those goals. Some of these systems use the notion of a role to provide a level of indirection between the goal and the protocols used to achieve the goal. This mapping is done at design time and thus, these systems have no inherent mechanism for adaptation when unexpected events occur. Unexpected events include the addition of new goals and the failure of required capabilities and protocols. Additionally, goal-oriented multiagent systems should be highly adaptive. Goal-oriented systems allow the goals of the agents to change while the system is running. These changes should reflect the agent's current view of the environment and should alter the state of the agent's goals. Thus, systems that are designed with static goal-to-protocol assignments do not reflect the ethos of a goal-oriented system. Goal-oriented systems allow the goals of an agent to dynamically change, but the actual behavior of the Agent can not dynamically adapt to those changes. The static mapping between goals and protocols also makes updating the goal model a difficult task. Adding or modifying a goal can require a protocol modification or the addition of a new protocol. The modification of a protocol requires the designer to verify that the protocol still meets the specification. If that protocol is used for other goals, the designer must verify that the protocol can still achieve those goals. It is clear that a runtime model would provide a more dynamic goal-oriented system.

7.2 The Formal Interaction Framework

The Framework for Interacting Agents (FIA) developed in this thesis provides three major contributions.

(I) It captures the concepts and relationships involved in agent interaction in a formal model that supports automated tools.

(II) It defines a set of reasoning algorithms that can prove whether an interaction is beneficial in achieving an agent's goal.

(III) It defines a set of algorithms that can adapt an agent's behavior at runtime to changes in the agent's capabilities.

The Framework for Interacting Agents (FIA) formalizes the concepts required for agent interaction. Agent interaction has been formalized in other models, but each of these has major drawbacks. For example there are formal models based on Event Calculus, such as the Commitment Machine and Joint Action Model. The Commitment Machine requires that protocols be modeled at a low level instead of the high level used by MAIM. The Joint Action Model requires that the protocols and the agents must maintain a common shared state, which is not always possible. MAIM formally defines entities like Agents, Goals, Atomic Actions, Interactions and Protocols. It also defines how these entities are related to one another. For example, Agents have Goals, they perform Atomic Actions and they participate in Interactions and Protocols. This model formally defines these entities and the relationships between them. Current state-of-the-art protocol specification techniques are performed using a formal model. The behavior of these protocols must be verified by automated tools (such as SPIN). MAIM's formal definition also allows automated tools to reason about the behavior of the model. The use of automated tools reduces the risk that an error can be made. Systems that rely on humans to specify and verify their correctness are error prone. However, once an automated tool is designed, it can perform the verification task correctly every time. The use of automated tools allows systems to be produced that are provably correct. For example, most programming languages provide a type system that ensures that programmers cannot do unsafe operations. Thus, if a programmer uses a type safe language, the resulting system will be type safe. Automated tools do the heavy lifting required to verify that a model behaves as desired through every possible situation. Protocol verification tools allow designers to specify the system in terms of messages, processes and states. Designers can then specify properties

and verify that the protocol has those properties. MAIM provides a model that can be used to define interactions within a multiagent system.

Providing a useful model is the first step in advancing the current state of multiagent system interaction. The second step is to develop the algorithms that can reason over that model. For example, the Promela model is only useful when combined with the SPIN model checker. The verification that SPIN provides makes the Promela model more useful. In a similar way, the FIA algorithms make MAIM more useful. ***The FIA reasoning algorithms can prove to an Agent that an interaction can achieve its Goal.*** The FIA reasoning algorithms enumerate and create proofs based on the agents current state and the MAIM model. These proofs are a mechanical but tedious process and thus are a good candidate for automation. Such processes can be performed by humans, but humans can easily err when performing the same task multiple times. The binding mechanism provides a formal method for specifying how information flows within an Agent. The FIA reasoning algorithms defines a process for enumerating the possible bindings within the system, which increases rapidly with the addition of new protocols. This process is also time consuming and thus automating it will make a designer's job easier. After the possible bindings have been enumerated, each of the bindings can be verified for correctness. The bindings that cannot be verified will not help the Agent to achieve their goals. As agents are goal-oriented, there is no reason for the Agent to perform actions that cannot achieve their goals.

The FIA algorithms can be used to adapt an agent's behavior at runtime. The FIA reasoning algorithms can be integrated into agents such that those agents can dynamically determine which interactions will achieve their goals. This process can be initiated by a goal change, a capability failure or a protocol failure. Goal-oriented multiagent systems rely on the ability of the agents to accept new goals and achieve goals that they possess. Thus, agents that use FIA at runtime can accurately select interactions that will achieve their goals. This reasoning is automated so no additional work must be done by system designers. Agents that have the ability to detect when their capabilities have failed,

gain even more robustness when using FIA. These agents can use FIA's reasoning to find interactions that meet their goals and they can weed out the associated protocols with failed capabilities. This process reduces the likelihood that an agent chooses an interaction that will fail and increases the likelihood that the interaction will achieve the agent's goal. It is clear that a runtime component will provides agents with a more robust mechanism for choosing interactions and achieving their goals.

7.3 Future work enabled by FIA

FIA provides system designer with a powerful tool. Designers can elicit goals for agents and then run the goals, agents, interactions and protocols through the FIA algorithms, which can prove which combinations of interaction and protocols achieve the goals of those agents. If goals do not have any interactions or protocols that achieve them, either the specification is incorrect or new interactions and protocols need to be designed. This process gives designers more flexibility and knowledge about how their systems will behave when executed. The new research enabled by FIA include:

(I) The modification of the model at runtime

(II) The optimization of protocol selection

(III) The development of design tools

(IV) The development of practical negotiation methods

Designers can use FIA to create multiagent systems without having to map goals to protocols *a priori*. This will allow designers to dynamically add and remove goals, protocols, actions and interactions to agents at runtime. When the MAIM model is updated with new specifications, the system can immediately use those protocols, actions and interactions to achieve goals of the agents in the system. This allows the agents to run continuously and, if errors are found, the components that are incorrect can be replaced at runtime. How to easily integrate MAIM with executable code is an open research question.

Multiagent systems can extend MAIM to use performance functions to select from a set of valid interactions and protocols. For example, performance functions can be defined for the Bluetooth and WiFi protocols. One performance function may optimize for bandwidth (choosing WiFi) while another may optimize for power efficiency (choosing Bluetooth). Different protocols may be selected, depending on which performance function is chosen. The process of creating applicable performance functions is an area with valuable research opportunities.

FIA allows a goal to be defined without having to bind a specific protocol to the achievement of that goal. This allows goals and protocols to be specified independently and relies on FIA to dynamically bind protocols to goals. The separation of goal and protocol specification will make designing interactions for goal-oriented multiagent systems more robust and less tedious. There are many opportunities for work that can extend the results of this thesis to make FIA even easier to use. For example, because MAIM is formally defined, a language can be created to represent the model. This language can be modeled graphically using tools that will make the specification easier and the visualization of MAIM better. This will help designers to develop models much more rapidly than if they have to manually encode the model using Java or XML. The Google Android platform uses just this type of model[56]. While most resources within an Android project are specified through text editors, some of the resources (such as layouts) can also be specified by graphical tools. These tools speed up the design process, which in turn decreases the time required to deliver a product.

MAIM also requires a way for agents to negotiate protocol and interaction details. The method used for the demonstration system was direct negotiation, but a brokering system or a service discovery system could also be used. These methods are used in a variety of fields where services may not be known in advance. The negotiation process can be provided by the system or it can be developed for a specific application. A major benefit of FIA is that not all agents in the system must use FIA for it to be useful. It is possible that only one agent uses FIA to select interactions. This agent could register with an interaction brokering

service and other agents could use the brokering service to find appropriate interactions. How this can be used to integrate legacy software into a multiagent system is another interesting research topic.

7.4 FIA Limitations

FIA may not be the suitable for some systems. For example, the current set of FIA algorithms requires access to `XSB` and a `SAT Solver`, which consumes many CPU cycles and large amounts of power. This may not be feasible for systems that are limited in either of these resources. In the physical demonstration system, the algorithms were not run on the tablet due to these limitations. FIA algorithms are, however, composed of a large number of parallel computations. This means that the required processing can be done quickly on a distributed system or a cluster of computers. Redesigning these algorithms to use these resources can lead to a significant decrease in the amount of time required to compute possible interactions and would be an interesting area of research. Another method of decreasing the time required to select an interaction is to cache the results of previous runs of the algorithms, which can help the agents to quickly locate possible interactions that achieve their goals. The cache provides a fast method of determining results, but cached results can lead to incorrect conclusions if the parameters of the goal change. To eliminate the risk of incorrect results, care must be taken to include both the agent's knowledge and the parameters of the goal.

The final major limitation is that FIA can suggest interactions that will never succeed. This is the case when a protocol has a disjunctive property or an interaction has a disjunctive postcondition. In the viable mode, the disjunctive properties are broken down into multiple proofs. It is possible for one agent to use one property to prove that their goal can be achieved while another agent uses an opposing property to prove that their goal can be achieved. It is possible that these two properties are mutually exclusive and thus at most one agent may achieve their desired goal. However, this situation is unlikely to occur in

most system designs.

The simulation results shown in Section 5.2 shows that FIA can be used to create robust multiagent systems. A FIA-based multiagent system can adapt to failures that typical multiagent systems fail to handle. The formal model allows the FIA reasoning algorithms to predict the behavior of the system. This reasoning allows the multiagent system to be adaptive while at the same time maintaining the distributed nature of the agents.

Bibliography

[1] Computer animation. http://www.bbc.co.uk/dna/h2g2/A3421045.

[2] How to write a telegraph. http://www.telegraph-office.com/pages/telegram.html, 2010.

[3] Nunit. http://www.nunit.org/.

[4] Xsb. http://xsb.sourceforge.net/.

[5] Two-character, single error-correcting system compatible with telegraph transmission, 1968. US Patent 3,412,380.

[6] Chanook helicopter, 07 2010.

[7] 11 2011.

[8] Facebook. http://www.facebook.com/, 02 2011.

[9] Java path finder. http://babelfish.arc.nasa.gov/trac/jpf, 08 2011.

[10] Junit website. http://www.junit.org/, 07 2011.

[11] Models at runtime conference. http://www.comp.lancs.ac.uk/~bencomo/MRT11/, 08 2011.

[12] Toyota: Software to blame for prius brake problems. http://articles.cnn.com/2010-02-04/world/japan.prius.complaints_1_brake-system-anti-lock-prius-hybrid?_s=PM:WORLD, 07 2011.

[13] Twitter. http://twitter.com/, 02 2011.

[14] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and M.D. Mickunas. Cerberus: a context-aware security scheme for smart spaces. In *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 489–496. IEEE, 2003.

[15] M. Alavi and D.E. Leidner. Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS quarterly*, pages 107–136, 2001.

[16] F. Appendix and G. Appendix. Artificial intelligence through prolog by neil c. rowe.

[17] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto. Cache-based model checking of networked applications: From linear to branching time. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 447–458. IEEE Computer Society, 2009.

[18] S. Baase. A gift of fire. 2003.

[19] T. Balch and R.C. Arkin. Communication in reactive multiagent robotic systems. *Autonomous Robots*, 1(1):27–52, 1994.

[20] F. Barbosa and J.C. Cunha. A coordination language for collective agent based systems: Grouplog. In *Proceedings of the 2000 ACM symposium on Applied computing-Volume 1*, pages 189–195. ACM, 2000.

[21] M. Barbuceanu and M.S. Fox. Cool: A language for describing coordination in multi agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 17–24. Citeseer, 1995.

[22] Bell Labs. Spin model checker. http://spinroot.com/spin/whatispin.html, November 2008.

[23] C. Bernon, M. Cossentino, M.P. Gleizes, P. Turci, and F. Zambonelli. A study of some multi-agent meta-models. *Agent-Oriented Software Engineering V*, pages 62–77, 2005.

[24] C. Bernon, M.P. Gleizes, S. Peyruqueou, and G. Picard. Adelfe: A methodology for adaptive multi-agent systems engineering. *Engineering Societies in the Agents World III*, pages 70–81, 2003.

[25] G. Beydoun, G. Low, B. Henderson-Sellers, H. Mouratidis, J.J. Gomez-Sanz, J. Pavon, and C. Gonzalez-Perez. Faml: a generic metamodel for mas development. *Software Engineering, IEEE Transactions on*, 35(6):841–863, 2009.

[26] K. Birman, R. Friedman, and M. Hayden. The maestro group manager: A structuring tool for applications withmultiple quality of service requirements. 1997.

[27] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[28] Encyclopedia Britannica. *Claude Chappe*. Encyclopædia Britannica Online, 2011.

[29] DC Bunzli, S. Mena, and U. Nestmann. Protocol composition frameworks a header-driven model. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 243–246. IEEE, 2005.

[30] P. Busetta, R. R
"onnquist, A. Hodgson, and A. Lucas. Jack intelligent agents-components for intelligent agents in java. *AgentLink News Letter*, 2:2–5, 1999.

[31] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. pages 1789–1901, 2006.

[32] C. Cheong and M. Winikoff. Hermes: Designing goal-oriented agent interactions. *LECTURE NOTES IN COMPUTER SCIENCE*, 3950:16, 2006.

[33] Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Reasoning about agents and protocols via goals and commitments. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 457–464, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[34] A. Colmerauer and P. Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.

[35] Survey Equipment Corp. Survey equipment, 2012.

[36] M. Cossentino. From requirements to code with the passi methodology. *Agent-oriented methodologies*, pages 79–106, 2005.

[37] F. Dalpiaz, A. Chopra, P. Giorgini, and J. Mylopoulos. Adaptation in open systems: Giving interaction its rightful place. *Conceptual Modeling–ER 2010*, pages 31–45, 2010.

[38] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. An architecture for requirements-driven self-reconfiguration. In *Advanced Information Systems Engineering*, pages 246–260. Springer, 2009.

[39] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: an environment for goal-driven requirements engineering. *Proceedings of the 19th international conference on Software engineering*, pages 612–613, 1997.

[40] S. DeLoach. The mase methodology. *Methodologies and software engineering for agent systems*, pages 107–125, 2004.

[41] S. DeLoach. Engineering organization-based multiagent systems. *Software Engineering for Multi-Agent Systems IV*, pages 109–125, 2006.

[42] S. DeLoach. Omacs: A framework for adaptive, complex systems. *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, pages 76–98, 2009.

[43] S.A. DeLoach. Analysis and design using mase and agenttool, 2001.

[44] S.A. DeLoach and M. Miller. A goal model for adaptive complex systems. *International Journal of Computational Intelligence: Theory and Practice*, 5(2), 2010.

[45] T. Doi, Y. Tahara, and S. Honiden. Iom/t: an interaction description language for multi-agent systems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 778–785, 2005.

[46] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker and its implementation. *Model Checking Software*, pages 623–624, 2003.

[47] fcc. Emergency alert system. http://transition.fcc.gov/pshs/services/eas/, 08 2011.

[48] P.H. Feiler. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006.

[49] T. Finin, R. Fritzson, D. McKay, and R. McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[50] R. France and B. Rumpe. Domain specific modeling. *Software and Systems Modeling*, 4(1):1–3, 2005.

[51] X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In *Computer Aided Verification*, pages 394–395. Springer, 2004.

[52] J. Garcia-Ojeda, S. DeLoach, W. Oyenan, and J. Valenzuela. O-mase: a customizable approach to developing multiagent development processes. *Agent-Oriented Software Engineering VIII*, pages 1–15, 2008.

[53] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[54] D. Garlan, R.T. Monroe, and D. Wile. *Acme: Architectural description of component-based systems*. Cambridge University Press, 2000.

[55] G. Gazdar and C.S. Mellish. *Natural language processing in PROLOG: an introduction to computational linguistics*. Addison-Wesley Pub. Co., 1989.

[56] Google. Android website, 2012.

[57] K. Havelund, M. Lowry, S.J. Park, C. Pecheur, J. Penix, W. Visser, J.L. White, et al. Formal analysis of the remote agent before and after flight. In *Lfm2000: Fifth NASA Langley Formal Methods Workshop*. Citeseer, 2000.

[58] G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[59] B. Huberman and S.H. Clearwater. A multi-agent system for controlling building environments. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 171–176, 1995.

[60] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge Univ Pr, 2004.

[61] Dell Hymes. *Foundations of Sociolinguistics: An Ethnographic Approach.* U of Pennsylvania, 1974.

[62] P. Kars. The application of promela and spin in the bos project. In *The SPIN verification system: the second Workshop on the SPIN Verification System: proceedings of a DIMACS workshop, August 5, 1996*, volume 32, page 51. Amer Mathematical Society, 1997.

[63] I. Keidar. Distributed computing column 42: game theory and fault tolerance in distributed computing. *ACM SIGACT News*, 42(2):68–68, 2011.

[64] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[65] P. Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[66] 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, 2000.

[67] S. Kumar and P.R. Cohen. Staple: An agent programming language based on the joint intention theory. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1390–1391. IEEE Computer Society, 2004.

[68] Sanjeev Kumar, Marcus J. Huber, and Philip R. Cohen. Representing and executing protocols as joint actions. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 543–550, New York, NY, USA, 2002. ACM.

[69] K. Kuwabara, T. Ishida, and N. Osato. Agentalk: Describing multiagent coordination protocols with inheritance. pages 460–465, 1995.

[70] T. Lacey and S.A. DeLoach. Automatic verification of multiagent conversations. Technical report, NASA Center for AeroSpace Information, 7121 Standard Dr, Hanover, Maryland, 21076-1320, USA, 2000.

[71] T.H. Lacey and S.A. DeLoach. Verification of agent behavioral models. In *Proceedings of the International Conference on Artificial Intelligence*, pages 557–564. Citeseer.

[72] H.M. Lee, D.S. Park, H.C. Yu, and G. Lee. Frasystem: fault tolerant system using agents in distributed computing systems. *Cluster Computing*, 14(1):15–25, 2011.

[73] H.J. Levesque, P.R. Cohen, and J.H.T. Nunes. On acting together. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 94–99. Boston, MA, 1990.

[74] Enno Lubbers, Marco Platzner, Christian Plessl, Ariane Keller, and Bernhard Plattner. Towards adaptive networking for embedded devices based on reconfigurable hardware. *Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2010.

[75] O. Marin, P. Sens, J.P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agent systems. In *Proceedings of ERSADS*, pages 195–201. Citeseer, 2001.

[76] H. Mazouzi, A.E.F. Seghrouchni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 517–526. ACM, 2002.

[77] Erin McKean, editor. *New Oxford American Dictionary*. Oxford University Press, 2005.

[78] K.L. McMillan. Symbolic model checking: an approach to the state explosion problem. Technical report, DTIC Document, 1992.

[79] S. Mellor, K. Scott, A. Uhl, and D. Weise. Model-driven architecture. *Advances in Object-Oriented Information Systems*, pages 233–239, 2002.

[80] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. cactus: Comparing protocol composition frameworks. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 189–198. IEEE, 2003.

[81] Katherine Miller. *Organizational Communication: Approaches and Processes*. Wadsworth Cengage Learning, fifth edition, 2009.

[82] M. Morandini, L. Penserini, and A. Perini. Operational semantics of goal models in adaptive agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 129–136. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[83] I. Nonaka. Takeuchi.(1995) the knowledge creating company. *New York*.

[84] J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in uml. In *Agent-Oriented Software Engineering*, pages 201–218. Springer, 2001.

[85] T. Osman, W. Wagealla, and A. Bargiela. An approach to rollback recovery of collaborating mobile agents. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 34(1):48–57, 2004.

[86] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *Proceedings of the 3rd international conference on Agent-oriented software engineering III*, pages 174–185. Springer-Verlag, 2002.

[87] Laurence Peter. How the poles cracked nazi enigma secret. http://news.bbc.co.uk/2/hi/europe/8158782.stm, July 2009.

[88] Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002.

[89] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[90] J.B. Postel, L.L. Garlick, R. Rom, and STANFORD RESEARCH INST MENLO PARK CALIF AUGMENTATION RESEARCH CENTER. Transmission Control Protocol Specification., 1976.

[91] Princeton. zchaff model checker. http://www.princeton.edu/~chaff/zchaff.html, 7 2011.

[92] J.G. Quenum, S. Aknine, O. Shehory, and S. Honiden. Dynamic protocol selection in open and heterogeneous systems. *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2006 Main Conference Proceedings)(IAT'06)-Volume 00*, pages 333–341, 2006.

[93] A.S. Rao and M.P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco, 1995.

[94] I. Recommendation. 200 (1994)— iso/iec 7498-1: 1994. *Information technology–Open Systems Interconnection–Basic Reference Model: The basic model.*

[95] P. Remagnino, T. Tan, and K. Baker. Multi-agent visual surveillance of dynamic scenes. *Image and Vision Computing*, 16(8):529–532, 1998.

[96] M.B.D. Robby and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, September*, pages 01–05, 2003.

[97] S. Rosenthal, J. Biswas, and M. Veloso. An effective personal mobile robot agent through symbiotic human-robot interaction. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 915–922. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[98] Peter Russell and Stuart Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[99] Murray Shanahan. Event calculus explained. urlhttp://www.doc.ic.ac.uk/ mpsha/E-CExplained.pdf.

[100] CE Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[101] A.D. Smith and F. Offodile. Information management of automatic data capture: an overview of technical developments. *Information Management & Computer Security*, 10(3):109–118, 2002.

[102] Spin. Promela modeling language. http://spinroot.com/spin/Man/promela.html, 12 2008.

[103] D. Stenmark. Information vs. knowledge: The role of intranets in knowledge management. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 928–937. IEEE, 2002.

[104] William Stewart. Arpanet. http://www.livinginternet.com/i/ii_arpanet.htm.

[105] M.E. Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

[106] H. Sullivan. Data demodulator, 1970. US Patent 3,536,840.

[107] F.W. Taylor. *The principles of scientific management*. Harper & brothers, 1919.

[108] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr, and J.E. Robbins. A component-and message-based architectural style for gui software. In *Proceedings of the 17th international conference on Software engineering*, pages 295–304. ACM, 1995.

[109] Sarah Trenholm and Arthur Jensen. *Interpersonal Communication*. Oxford University Press, 2004.

[110] I. Tuomi. Data is more than knowledge: Implications of the reversed knowledge hierarchy for knowledge management and organizational memory. In *System Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, pages 12–pp. IEEE, 1999.

[111] Carnegie Mellon University. Aadl examples, 2008.

[112] R. Van Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[113] M.B. Van Riemsdijk, M. Dastani, and M. Winikoff. Goals in agent systems: a unifying framework. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 713–720. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[114] ML Visinsky, JR Cavallaro, and ID Walker. Robotic fault detection and fault tolerance: A survey. *Reliability Engineering & System Safety*, 46(2):139–158, 1994.

[115] S. Wahl and H. Spada. *Cognitive Science Quarterly*, 1(1):3–32, 2000.

[116] L. Wang, H. Li, D. Goswami, and Z. Wei. A fault-tolerant multi-agent development framework. *Parallel and Distributed Processing and Applications*, pages 126–135, 2005.

[117] Y. Wang, S.A. McIlraith, Y. Yu, and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 293–302. ACM, 2007.

[118] Max Weber. *Economy and society*. University of California Press, 1978.

[119] G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. The MIT press, 1999.

[120] M. Winikoff. Implementing commitment-based interactions. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8. ACM, 2007.

[121] Michael Woodridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2 edition, 2009.

[122] M. Wooldridge, M. Fisher, M.P. Huget, and S. Parsons. Model checking multi-agent systems with mable. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 952–959. ACM, 2002.

[123] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152, 1995.

[124] M. Wooldridge, N.R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[125] P. Xuan, V. Lesser, and S. Zilberstein. Communication decisions in multi-agent cooperation: Model and experiments. In *Proceedings of the fifth international conference on Autonomous agents*, pages 616–623. ACM, 2001.

[126] P. Yolum and M.P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence*, 42(1):227–253, 2004.

[127] E.S.K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. *re*, page 226, 1997.

Appendix A

Additional Interaction Model Definitions

This Appendix formally defines the basic Interaction Model elements in full.

A.1 Interaction Model and Definitions

The base of most formal models starts with types. A type system allows us make more robust systems, as the types allow us to make assumptions on how the program will behave. In Definition 13, a canonical definition of a type system is given.

Definition 13 (*Type System*)

> ***Type System:*** *A tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute*[88].

The Interaction Framework uses a type system to prove properties about program/-mulitagent system behavior. These behavior characteristics allow the Interaction Framework to prove whether or not an Interaction can achieve a Goal for an Agent (Chapter 4). Definition 14 defines the Type entity, and a type has a name attribute. The name allows the model to discriminate between different types.

Definition 14 (*Type Attribute*)

> *name* *String*

In addition to discriminating between types, the Interaction Framework also defines that each type is unique. This uniqueness guarantees that if two types have the same name, then those types have the same semantic meaning. This restriction is codified below.

Restriction 4 (*Type Uniqueness*)

$\forall\ type_1 : Type,\ type_2 : Type \mid type_1.name = type_2.name \Leftrightarrow type_1 = type_2$

The Interaction Framework uses variables within the entity specifications. Definition 15 gives a loose definition of a variable. The value of the variables will change over time, but the Type of the variable restricts the possible values.

Definition 15 (*Variable*)

Variable: *a data item that may take on more than one value during or between programs*[77].

Variables are used in the various parts of the Interaction Model, such as in Goals, Agents, Interactions and Protocols. The variables may change over the course of the system, and the system can adapt its behavior based on those changes. Definition 16 gives the formal definition of the attributes that belong to variables.

Definition 16 (*Variable Attributes*)

name	*String*
value	*unknown* $\vee$ *value*

These attributes give the model the information required to discriminate between variables. Variables in multiagent systems may have an unknown value. This may be the case if the agent that has the variable but they have not sensed the value from the environment. The agent may also acquire the value of the variable via communication with another agent. The *value* of a variable will be used in other parts of the Interaction Framework, such as in Interactions or in the playing of roles. The value, or lack thereof, of the variable will affect how the interactions and roles are carried out. The *name* of the variable allows us to differentiate between variables. There is a relation (Relation 6) between the Variables and the Types of those Variables.

Relation 6 (*Type Variable*)

typeof $\subseteq$ *Type* $\times$ *Variable*

The above relation specifies that Types and Variables are related. This definition by its self is very loose and by just this relation, there could be a Variable that is related to many types. But this violates the premise that each variable has exactly one type. Therefore there is a restriction on this relation in Restriction 5. This restriction guarantees the each variable will have exactly one type.

Restriction 5 (*Variable Types*)

$\forall$ *type: Type, type1: Type, variable:Variable*|(*(type,variable)* $\in$ *typeof*)$\wedge$(*(type1,variable)* $\in$ *typeof*) $\implies$ *type* = *type1*

Now that Types and Variables have been defined for the Interaction Framework a language can be defined that uses those Variables. The language is a specification language that allows system designers to specify properties of different entities in our system. The specification language allows formal properties to be defined. These properties will either hold in the future, they at the current moment in time. This specification language is defined in Definition 17. In some entities (Protocols and Protocol Roles) the properties given are LTL formulas that specify safety and liveness properties. Other entities will not be required to use LTL logic symbols.

Definition 17 (*Specification Language*)

Specification Language: *First Order Logic formula with domain specific variables, types and LTL logic symbols.*

The List of specification language symbols (Definition 18) specifies the valid symbols that can be used to write a sentence in the Specification Language.

Definition 18 (*Specification Language Symbols*)

Quantifiers	$\forall \exists$
Connectors	$\wedge \vee \implies \neg \supset \supseteq$
LTL Connectors	$\Box \Diamond \mathcal{U} \mathcal{R}$
Punctuation	$(){}[]$
Variables	*Set(Variable)*
Equality	$= <> \geq \leq$

An example of a valid sentence is a $\implies$ b, where a and b are variables. Expressions are statements that are either true, false or unknown. In the case of LTL properties the expressions are statements that are true, false or unknown at the current moment, and the truth of the statement is time dependent. The value of unknown comes in to play when you have a variable that does not have a known value.An example of an unknown value would be in a postcondition that states a property like x > 10. In this property the value of x in not known, but it can be inferred that the value is greater than 10. Expressions will be used in the specification of a variety of entities within the Interaction Framework.

Definition 19 (*Expression*)

Expression: *A valid statement in the Specification Language (Definition 17) using the specification language symbols in Definition 18.*

A.2 Agent Relations

The Agents knowledge is information that an Agent possesses. Relation 7 defines knowledge as a relation between Agents and Variables. Each Agent is allowed to add and remove information at their own discretion. The definition of Knowledge abstracts many of the details that an agent may associate with their knowledge.

Relation 7 (*Agent Knowledge*)

knowledge $\subseteq$ *Variable* $\times$ *Agent*

As seen in Figure 3.1 there is a relation between the Agent an the Goal. This relation is formally defined in Relation 8. The relation codifies the notion that an agent has a set

of goals that the agent may be trying to achieve. Each goal belongs to a single Agent in the multiagent system. This restriction follows from the function definition that is given in Section 3.1.1.

Relation 8 (*Agent Goal*)

has $\subseteq Agent \times Goal$

The **has** relation a function from a Goal to an Agent, and thus the restriction below is defined.

Restriction 6 (*One Agent Per Goal*)

$\forall a_1, a_2$: *Agent,goal:Goal*$|(a_1, goal) \in has \wedge (a_2, goal) \in has \implies a_1 = a_2$

Restriction 6 ensures that each goal belongs to one Agent. Many agents may have the same type of goal, but each goal belongs to a unique Agent.

Additionally there is a restriction that ensures that a single agent is allowed to play each role. This is codified in Restriction 7.

Restriction 7 (*Role Played By Single Agent*)

$\forall$ *a,b : Agents , r: Roles* $|$ *(a,r)* $\in$ *plays* $\wedge$ *(b,r)* $\in$ *plays* $\implies$ $a = b$

A.3 Interaction Role Relation & Restrictions

For each Interaction in our system, there must exist at least distinct 2 Interaction Roles. This is codified in Restriction 8.

Restriction 8 (*Non-empty Interaction Role Composition*)

$\forall\ i \in Interaction\ \exists\ r_1,\ r_2 : Role\ |$

$(i, r_1) \in InteractionRoleComposition \wedge (i, r_2) \in InteractionRoleComposition \wedge r_1 \neq r_2$

Relation 9 gives the range of the Interaction Roles. The first integer is the minimum number of instances that a particular interaction must have and the second is the maximum number of instances of the Interaction Role that can exist for a particular Interaction. The

Interaction Role Range relation allows the designer to be more expressive in the specification of the role composition.

Relation 9 (*Interaction Role Range*)

$$\textbf{\textit{InteractionRoleCompositionRange}} \subseteq \textit{Interaction} \times \textit{Interaction Role} \times \textit{Integer} \times \textit{Integer}$$

There is restriction on the Interaction Role Range to ensure that the min is always less than or equal to the max. This is listed in Restriction 9.

Restriction 9 (*Interaction Role Range min $\leq$ max*)

$$\forall\, i \in \textit{Interaction},\, ir \in \textit{Interaction Role},\, min, max \in \textit{Integer} \mid$$
$$(i, ir, min, max) \in \textit{InteractionRoleCompositionRange} \implies min \leq max$$

Additionally there is restrict on the relation such that there is only one minimum and maximum for each Interaction Role pair.

Restriction 10 (*Interaction Role Composition Uniqueness*)

$$\forall\, i \in \textit{Interaction}\ \forall\, ir : \textit{Interaction Role},\, \forall\, min_1, min_2, max_1, max_2 : \textit{Integer} \mid$$
$$(i, ir) \in \textit{InteractionRoleComposition} \wedge$$
$$(i, ir, min_1, max_1) \in \textit{InteractionRoleCompositionRange} \wedge$$
$$(i, ir, min_2, max_2) \in \textit{InteractionRoleCompositionRange} \implies$$
$$min_1 = min_2 \wedge max_1 = max_2$$

A.4 Protocol Role Relations & Restrictions

As stated above and shown in Diagram 3.1, each Protocol has two or more Protocol Roles in its composition.

Restriction 11 (*Non-empty Protocol Role Composition*)

$$\forall\, p \in \textit{Protocol}\ \exists\, r_1,\, r_2 : \textit{Role} \mid$$
$$(p, r_1) \in \textit{ProtocolRoleComposition} \wedge (p, r_2) \in \textit{ProtocolRoleComposition} \wedge r_1 \neq r_2$$

Relation 5 defines that the composition is between Protocols and Protocol Roles. This relation is defined when the protocol designers create the protocol definition. Protocol Roles in literature may be called lifelines or processes.

In Relation 10 a range on the Protocol Roles is defined. This range is just like the range on the Interaction Role range. The first integer is the minimum number of instances that a particular role must have and the second integer is the maximum number of instances that role can have. The Protocol Roles are designed to implement Interaction Roles.

Relation 10 (*Protocol Role Range*)

$$\textbf{\textit{ProtocolRoleCompositionRange}} \subseteq \mathit{Protocol} \times \mathit{Protocol\ Role} \times \mathit{Integer} \times \mathit{Integer}$$

Again, just as in the Interaction Role, there is a restriction on Relation 10 that the minimum must be less than or equal to the maximum. This is shown in Restriction 12.

Restriction 12 (*Protocol Role Range min $\leq$ max*)

$$\forall\ p \in \mathit{Protocol},\ pr \in \mathit{Protocol\ Role},\ min, max \in \mathit{Integer}\ |$$
$$(p, pr, min, max) \in \mathit{ProtocolRoleCompositionRange} \implies min \leq max$$

Additionally, we there is also the restriction on the range. This ensures that it is unique for each Protocol-Protocol Role pair (Restriction 13).

Restriction 13 (*Protocol Role Composition Uniqueness*)

$$\forall\ p \in \mathit{Protocol}\ \forall\ pr : \mathit{Protocol\ Role},\ \forall\ min_1, min_2, max_1, max_2 : \mathit{Integer}\ |$$
$$(p, pr) \in \mathit{InteractionRoleComposition} \wedge$$
$$(p, pr, min_1, max_1) \in \mathit{ProtocolRoleCompositionRange} \wedge$$
$$(p, pr, min_2, max_2) \in \mathit{ProtocolRoleCompositionRange} \implies$$
$$min_1 = min_2 \wedge max_1 = max_2$$

The Protocol definition defines that there is a relationship between the Interaction and the Protocol (Relation 4). This relationship is similar to relationship between the Interaction Role and the Protocol Role. Relation 11 formally defines the implements relationship between Interaction Roles and Protocol Roles.

Relation 11 (*Protocol Role implements Interaction Role*)

implements $\subseteq$ *Interaction Role* $\times$ *Protocol Role*

A.5 Variable Definitions

Free Variables

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(x \implies y) & = & FV(x) \cup FV(y) \\
FV(\forall\ x : A \mid e\ ) & = & FV(e) \backslash \{x\} \\
FV(\exists\ x : A \mid e\ ) & = & FV(e) \backslash \{x\} \\
FV(x \wedge y) & = & FV(x) \cup FV(y) \\
FV(x \vee y) & = & FV(x) \cup FV(y) \\
FV(x = y) & = & FV(x) \cup FV(y) \\
FV(x < y) & = & FV(x) \cup FV(y) \\
FV(x > y) & = & FV(x) \cup FV(y) \\
FV(x \geq y) & = & FV(x) \cup FV(y) \\
FV(x \leq y) & = & FV(x) \cup FV(y) \\
FV(x \subset y) & = & FV(x) \cup FV(y) \\
FV(x \subseteq y) & = & FV(x) \cup FV(y) \\
FV(x\ \mathcal{U}\ y) & = & FV(x) \cup FV(y) \\
FV(x\ \mathcal{R}\ y) & = & FV(x) \cup FV(y) \\
FV(\Box x) & = & FV(x) \\
FV(\Diamond x) & = & FV(x)
\end{array}
$$

The substitution is defined by a set of rules defined for variables and expressions, as given in Table A.5.

A.6 Restrictions

The Interaction Model defines how each of the entities within the Interaction Framework are related. This section defines some restrictions on those relations.

The first restriction that is defined is on the achieves relation between Goals and Actions. Restriction 14 states that if a goal and action are in the achieves relation then there must exist a possible GoalActionBindings. Given an input binding tuple (b_1), the precondition must be true. Also the post condition must imply the goals state given the input binding

Substitution Table

$$
\begin{array}{lcl}
[e \mapsto v]e & = & v \\
[e \mapsto v](x \implies y) & = & ([e \mapsto v]x) \implies ([e \mapsto v]y) \\
[e \mapsto v](\forall\ x : A \mid y\ ) & = & if(e \neq x)(\forall\ x : A\ |[e \mapsto v]y) \\
[e \mapsto v](\forall\ x : A \mid y\ ) & = & if(e = x)(\forall\ x : A\ |y) \\
[e \mapsto v](\exists\ x : A \mid y\ ) & = & \text{if}(e \neq x)(\exists\ x : A\ |[e \mapsto v]y) \\
[e \mapsto v](\exists\ x : A \mid y\ ) & = & if(e = x)(\exists\ x : A\ |y) \\
[e \mapsto v](x \wedge y) & = & ([e \mapsto v]x) \wedge ([e \mapsto v]y) \\
[e \mapsto v](x \vee y) & = & ([e \mapsto v]x) \vee ([e \mapsto v]y) \\
[e \mapsto v](x = y) & = & ([e \mapsto v]x) = ([e \mapsto v]y) \\
[e \mapsto v](x < y) & = & ([e \mapsto v]x) < ([e \mapsto v]y) \\
[e \mapsto v](x > y) & = & ([e \mapsto v]x) > ([e \mapsto v]y) \\
[e \mapsto v](x \leq y) & = & ([e \mapsto v]x) \leq ([e \mapsto v]y) \\
[e \mapsto v](x \geq y) & = & ([e \mapsto v]x) \geq ([e \mapsto v]y) \\
[e \mapsto v](x \subset y) & = & ([e \mapsto v]x) \subset ([e \mapsto v]y) \\
[e \mapsto v](x \subseteq y) & = & ([e \mapsto v]x) \subseteq ([e \mapsto v]y) \\
[e \mapsto v](x \mathcal{U} y) & = & ([e \mapsto v]x) \mathcal{U} ([e \mapsto v]y) \\
[e \mapsto v](x \mathcal{R} y) & = & ([e \mapsto v]x) \mathcal{R} ([e \mapsto v]y) \\
[e \mapsto v](\Box x) & = & (\Box [e \mapsto v]x) \\
[e \mapsto v](\Diamond x) & = & (\Diamond [e \mapsto v]x)
\end{array}
$$

(b_1) and the output binding (b_2).

Restriction 14 (*Action achieves Goal*)

$\forall$*goal:Goal,action:Action*|*(goal,action)* $\in$ *achieves* $\iff$
$\exists$ *agent: Agent* |$(agent, goal) \in has \wedge (agent, action) \in performs \wedge$
$(\exists b_1, b_2 : binding|(goal, action, agent, b_1, b_2) \in GoalActionBindings) \wedge (a.precondition\{\vec{b_1}\} \wedge$
$a.postcondition\{\vec{b_1}, \overleftarrow{b_2}\} \implies g.state\{\vec{b_2}, \overleftarrow{b_1}\})$

Restriction 15 defines that in an Interaction Role composition implies that the parameters of the Interaction Role are a subset of the parameters for the Interaction. This restriction gives the guarantee that if the input parameters are known for the interaction, then the required input parameters for the Interaction Role are also known. The definition of parameters does not imply that the parameters must be used in the precondition or postcondition of the Interaction or the Interaction Role.

Restriction 15 (*Interaction Role Composition Parameters*)

$\forall$*Interaction i, Interaction Role ir*|$(i, ir) \in InteractionRoleComposition \implies$

$ir.parameters \subseteq i.parameters$

A similar restriction is defined for the Protocol composition. The Protocol Role must have a subset of the parameters define in the Protocol.

Restriction 16 (*Protocol Role Composition Parameters*)

$\forall Protocol\ p,\ Protocol\ Role\ pr | (p, pr) \in ProtocolRoleComposition \implies$

$pr.parameters \subseteq p.parameters$

The protocol must do as much or more than the interaction requires (may need meta protocols to achieve the interaction).

Restriction 17 (*Protocol and Protocol Role Imply Interaction and Interaction Role*)

$\forall i{:}Interaction,\ ir{:}Interaction\ Role,\ pr{:}Protocol\ Role,\ p{:}Protocol,\ b_1, b_2{:}bindings |$

$(ir,pr) \in implements \wedge$

$(i, p, b_1, b_2) \in InteractionProtocolBindings \wedge$

$(p, pr) \in ProtocolRoleComposition \wedge$

$(i, ir) \in InteractionRoleComposition$

$\implies$

$(\frac{b_1,b_2}{pr.properties \implies ir.postcondition}) \wedge (\frac{b_1}{ir.precondition \wedge pr.properties})$

If an Interaction and a Protocol are in the implements relation, then there must exist an input and an output binding such that a proof can be made. This proof will show that if the precondition is true, then postcondition will hold.

Restriction 18 (*Implements implies valid binding*)

$\forall i : Interaction, p : Protocol\ | (i, p) \in implements \implies$

$(\exists b_1, b_2 | (i, p, b_1, b_2) \in InteractionProtocolBindings) \wedge$

$(\frac{b_1,b_2}{p.properties \implies i.postcondition}) \wedge (\frac{b_1}{i.precondition \wedge p.properties}))$

If an Interaction implements a Protocol, then each Interaction Role has an associated implementing Protocol Role.

Restriction 19 (*Protocol Role for each Interaction Role*)

$$\forall i_1 : Interaction, p_1 : Protocol\ ir : Interaction\ Role \mid$$
$$(i_1, p_1) \in implements \wedge (i_1, ir) \in InteractionRoleComposition \implies$$
$$(\exists\ pr : Protocol\ Role \mid (p_1, pr) \in ProtocolRoleComposition \wedge (ir, pr) \in implements)$$

Restriction 19 ensures that each Protocol Role has an Interaction Role that can implement it. This guarantees that if the Protocol has a Protocol Role and that Protocol is implemented by an Interaction, then there is an Interaction Role that implements that Protocol Role.

Restriction 20 (*Interaction Role for each Protocol Role*)

$$\forall i_1 : Interaction, p_1 : Protocol\ pr : Protocol\ Role \mid$$
$$(i_1, p_1) \in implements \wedge (p_1, pr) \in ProtocolRoleComposition \implies$$
$$(\exists\ ir : Interaction\ Role \mid (i_1, ir) \in InteractionRoleComposition \wedge (ir, pr) \in implements)$$

Restriction 19 and Restriction 20 ensure that if an Interaction and a Protocol are in the implements Relation, then there is a mapping for each role. These roles are the roles in the Interaction composition, or the roles in the Protocol composition.

A.7 Consistency

The binding algorithms put forth in Chapter 3 enables each agent to run the computations independently, which allows the process of selecting interactions to be distributed. Each agent runs the computation using its own local information and its own local goal possibly using different inputs and different bindings to create the proofs. To ensure that there is no conflict between these proofs, a consistently algorithm has been devised. An example is provided to show the consistency problem and how an interaction can be checked for consistency.

In the below example, `Agent A` and `Agent B` are both computing proofs for their desired interaction. As illustrated in Figure A.1 `Agent A` has an Interaction-Protocol binding $(A_i p)$ where $q \mapsto x$ and $r \mapsto y$. However, `Agent B` has the binding in reverse $q \mapsto y$ and $r \mapsto x$. This situation will produce compatible behavior if the agents are allowed to use these

bindings.

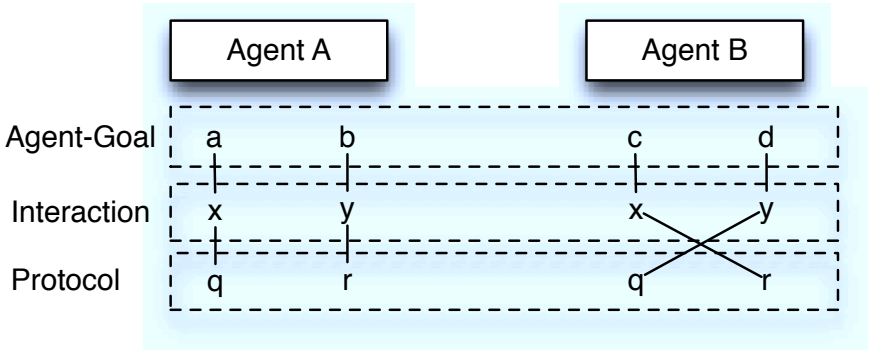


Figure A.1: *Consistent Example*

Algorithm 6 verifies that a group of bindings is consistent for a specific protocol and interaction. Line 3 iterates through all the variables in the interaction-to-protocol binding for $AgentA_{ip}$. In the example $domain(AgentA_{ip}) = \{q, r\}$. Line 4 verifies that the variable is also bound in *Agent B*'s interaction-to-protocol ($AgentB_{ip}$) binding and that each input variable in the protocol is bound to the same input variable in the interaction. If this is not the case, then the verification fails. The second half of the algorithm verifies the same property for the output bindings ($AgentA_{pi}$ and $AgentB_{pi}$).

Algorithm 6 Verifying the Consistency of two bindings

```
1: consistent(Binding AgentA_ip, AgentB_ip, AgentA_pi, AgentB_pi)
2: //Iterate through the input bindings
3: for all ∀ Variable q : domain(AgentA_ip) do
4:    if ¬(q ∈ domain(AgentB_ip) and AgentB_ip(q) = AgentA_ip(q)) then
5:       return false
6:    end if
7: end for
8: //Iterate through the output bindings
9: for all ∀ Variable q : domain(AgentA_pi) do
10:   if q ∉ domain(AgentB_pi) or AgentB_pi(q)! = AgentA_pi(q) then
11:      return false
12:   end if
13: end for
14: return true
```

The consistency algorithm defined above ensures that a pair of agents bindings do not

have inconsistent interaction-to-protocol bindings. There are several methods that the agents can use to perform this verification. The first method is for the initiating agent to generate the input and output bindings for the protocol. The initiating agent can then send those bindings to the other participants. These agents can these use these bindings in their proofs. A second method is for each agent to run their computations independently and then communicate their bindings during the negotiation process. Clearly agents must communicate to negotiate the protocol details, but inputs to the interaction and protocol **must** be communicated at some point for the interaction to commence.

Appendix B

Additional Functions

Algorithm 7 Getting Interaction-Protocol Input Bindings

```
1: Set<Bindings> getInputBindings(Interaction i, Protocol p)
2: Set<Bindings> result
3: for all Variable input ∈ p.getInputs() do
4:     Type inputType = input.getType
5:     Set<Variable> interactionVars = i.getInputs(inputType)
6:     for all Variable interactionInput ∈ interactionVars do
7:         result.add(new Binding(input, interactionInput))
8:     end for
9: end for
10: return result
```

Algorithm 8 Getting Interaction-Protocol Output Bindings

```
1: Set<Bindings> getOutputBindings(Interaction i, Protocol p)
2: Set<Bindings> result
3: for all Variable output ∈ i.getOutputs() do
4:     Type outputType = output.getType
5:     Set<Variable> protocolVars = p.getOutputs(outputType)
6:     for all Variable protocolOutput ∈ protocolVars do
7:         result.add(new Binding(output, protocolOutput))
8:     end for
9: end for
10: return result
```

B.1 LTL Example XSB Code

Listing B.1: *"XSB precondition Code"*

```
prefol(a(B,C)) :- property(Z), Z=ltl(r,A,B), precond(C).

prefol(a(A,C)) :- property(Z), Z=ltl(u,A,B), precond(C).

prefol(a(A,C)) :- property(Z), Z=ltl(g,A,Q), precond(C).

prefol(C):- property(Z), Z=ltl(n,A,Q), precond(C).

prefol(C):- property(Z), Z=ltl(f,A,Q), precond(C).
```

Listing B.2: *"XSB postcondition Code"*

```
postfol(A)     :- (property(Z); postfol(Z)), Z=ltl(g,A,Q).

postfol(true) :- (property(Z); postfol(Z)), Z=ltl(n,A,Q).

postfol(A)     :- (property(Z); postfol(Z)), Z=ltl(f,B,Q),
        B=ltl(g,A,R).

postfol(a(G,n(A)))  :- (property(Z);
         postfol(Z)),
         (property(S); postfol(S)),
         Z=ltl(r,B,A), B=ltl(g,G,Q),S=ltl(f,n(A),M).

postfol(o(G,A))  :- (property(Z); postfol(Z)),
         Z=ltl(r,B,A), B=ltl(g,G,Q).

postfol(o(G,A))  :- (property(Z); postfol(Z)),
         Z=ltl(u,A,B), B=ltl(g,G,Q).

postfol(B)     :- (property(Z); postfol(Z)),
         Z=ltl(g,ltlnot(A),Q), property(Y), Y=ltl(r,A,B).

postfol(A)     :- (property(Z); postfol(Z)),
         Z=ltl(g,ltlnot(B),Q), property(Y), Y=ltl(u,A,B).
```

Appendix C

Binding Formal Definitions

The bindings are the key component responsible for information flow in the Interaction Framework. Figure C.1 has been updated to include the types of the Variables in the Interactive Move example. The format for Variables is $a : b = c$ where a is the variable name, b is the type and c is the value if it exists. The example bindings that will be given are the bindings that lead to a valid proof. There are a vast number of bindings that can be created. Each one of these can use the correct types, but the number of bindings that lead to a valid proof are very few.

C.1 Variables and Expressions

Variables are one of the low level entities that are defined for the Interaction Framework. Variables are inter-entity related within the Interactive Framework via a the Binding System. This section provides a brief overview of Variables which lays the groundwork for the bindings system.

In Specification Expressions, there may be variables that are not bound to a value. The Interaction Move in Figure C.1 has many unbound input Variables. These variables are $\{currentlocation, startlocation, finishlocation, duration, starttime, time\}$. Each of these inputs does not have a value. Running an Interaction or executing a Protocol requires that the values of the inputs are known. The execution of the Interactive Move Interaction requires a value for each of these variables. The variables that do not have values are

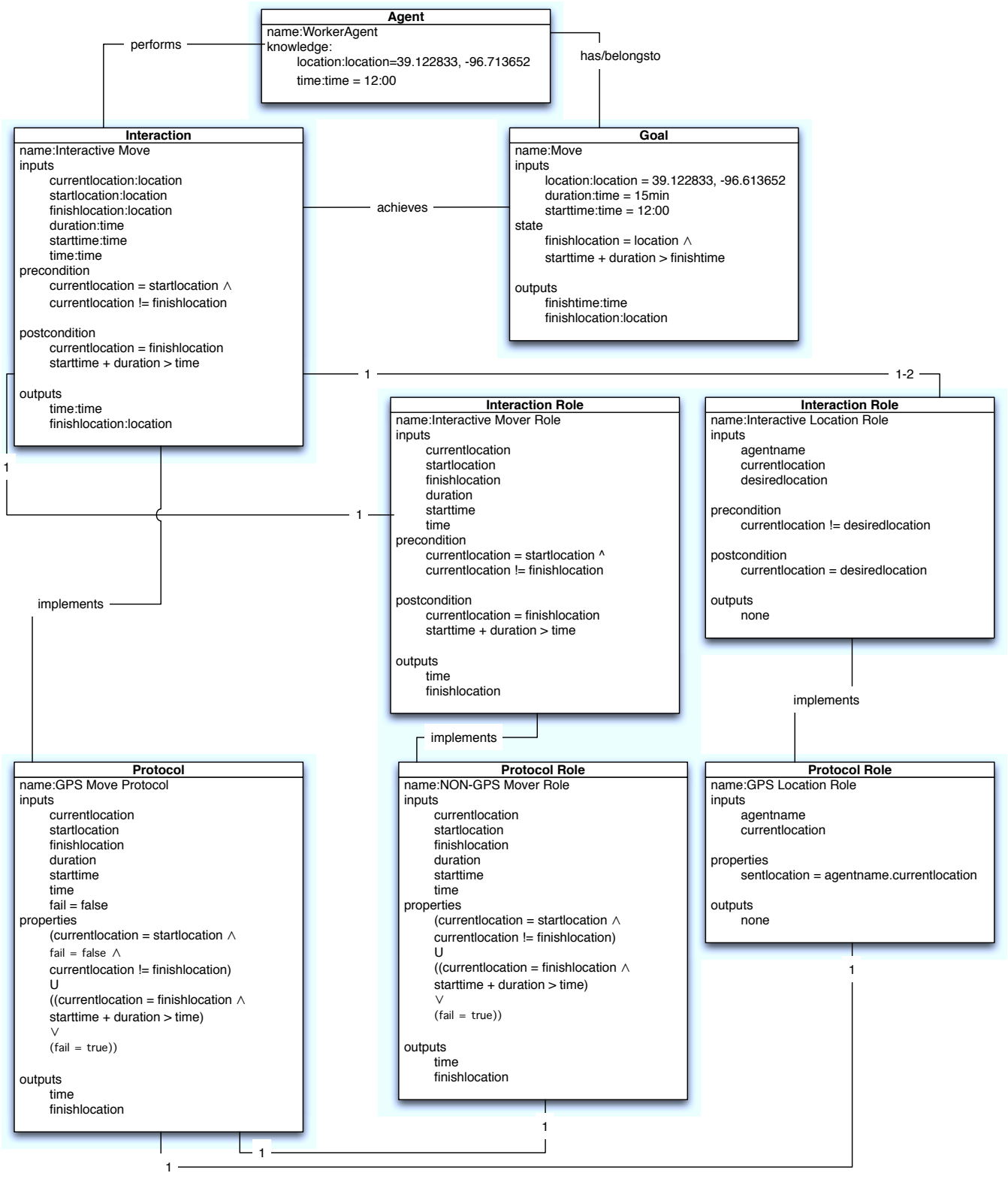


Figure C.1: *Move Specification with Types*

known as free variables. This section defines free variables and shows how the set of free variables can be derived. Intuitively a *Free Variable* is a Variable that is unbound within a Specification Expression (Definition 17) and the definition of the *free* relation is shown below.

Relation 12 (*Free Variable*)

$free \subseteq Variable \times Expression$

The *free* relation is between variables and expressions. A Specification Expression has a finite number of variables. A subset of those variables may actually be free and not bound within a context (such as a quantification). For example in the Expression $\forall x : Y | x \implies z$, there are two variables, x and z. In this example the variable x is bound by the universal quantifier, and thus the only free variable is z.

Functions have been defined to take a Specification Expression and calculate the set of free variables. This definition is located in Appendix A.5. In addition to the free variable definition, there is also a set of substitution rules for each binary and unary operator in the Specification Language.

C.2 Variable Convenience Functions

The following two algorithms (Algorithm 9 and 10) define the bindable Variables for each entity in the Interaction Model. A bindable Variable is a variable that can be used in a binding. Definition 20 defines that a variable of an Entity is the union of the inputs and the outputs.

Definition 20 (*Variables Fuction*)

$variables(Entity\ e) = inputVariables(e) \cup outputVariables(e)$

Algorithm 9 Input Variables

```
1: inputVariables (entity)
2: if entity is Goal then
3:     return Goal.inputs
4: else if entity is Agent then
5:     return Agent.knowledge
6: else if entity is Action then
7:     return Action.inputs
8: else if entity is Interaction Role then
9:     return Interaction Role.inputs
10: else if entity is Protocol then
11:     return Protocol.inputs
12: else if entity is Protocol Role then
13:     return Protocol Role.inputs
14: end if
```

Algorithm 10 Output Variables

```
1: outputVariables (entity)
2: if entity is Goal then
3:     return Goal.outputs - Goal.inputs
4: else if entity is Agent then
5:     return Agent.knowledge
6: else if entity is Action then
7:     return Action.outputs - Action.inputs
8: else if entity is Interaction Role then
9:     return Interaction Role.outputs - Interaction Role.inputs
10: else if entity is Protocol then
11:     return Protocol.outputs - Protocol.inputs
12: else if entity is Protocol Role then
13:     return Protocol Role.outputs - Protocol Role.inputs
14: end if
```

C.3 Binding formal definition

Binding are relation between Variables of the same equivalence class. Variables are of the same equivalence class if they have the same type. The Type System definition gives a guarantee that if the Variables have the same type then the variables have the same semantics (Appendix A, Restriction 4). If two Variables are bound (in a binding) then values at runtime will be the same for both Variables. In the Move Example in Figure C.1 a binding between Agent.location and Interaction Move.currentlocation would mean that the initial value, at runtime, for the Variable Interaction Move.currentlocation would be $39.122833, -96.713652$.

Relation 13 (*Binding Definition*)

$$binds \subseteq Set(Variable) \times Set(Variable)$$

Restriction 21 defines that the types must match for elements of the binds relation. This ensures that bindings are between equivalent Variables.

Restriction 21 (*Binding Type*)

$$\forall b : binds, v_1, v_2 : \ Variable \mid (v_1, v_2) \in binds \implies v_1.type = v_2.type$$

Not every Variable in the system needs to be bound and thus the binding relation is a partial function. An example of why a binding needs to be a function is as follows. Suppose that there is a variable x that is bound to y and z. Now suppose that $y = 10$ and $z = 20$. The variable x cannot have the same value as both y and z. Restriction 22 restricts the binding relation to a partial function.

Restriction 22 (*Binding Partial Function*)

$$\forall b : binds, v_1, v_2, v_3 : \ Variable \mid (v_1, v_2) \in binds \land (v_1, v_3) \in binds \implies v_2 = v_3$$

C.3.1 Forming Bindings

The next few sections define how a binding is formed between specific entities within the Interaction Model. These bindings are defined for both inputs and outputs of the associated

Goals, Agents, Actions and Protocols.

C.4 Action Bindings

The Action bindings are required in order for an Agent to perform an Action. The information will flow from the Agent and the Goal, into the Action. When the Action has been performed, the results of the Action flow into the Agents knowledge and the Agents Goal. The result of the Action should satisfy a Goal of the Agent. These bindings are defined for a given Action, Goal, and Agent.

C.4.1 Action Input Bindings

Agents are working to achieve Goals. Each Goal has a set of required inputs. The inputs allow the Agent to tailor the Actions that they perform. The Action definition requires that all inputs are known when that Action is performed. In addition to the values of the inputs, the Action's precondition must be satisfiable. One of the possible locations that the values can be bound to is the Agent's Knowledge. The relation between Actions and Knowledge in defined in Relation 14. Restriction 23 ensures that the Action Knowledge relation is a valid member of the Binding relation.

Relation 14 (*A-K Relation*)

$$ActionKnowledge \subseteq Action.inputs \times Agent.knowledge$$

Restriction 23 (*A-K is Binding*)

$$ActionKnowledge \subset binds$$

In Figure C.1 the Interaction Interactive Move has several inputs. Some of those inputs will be bound to the Agents Knowledge. For example the *startlocation* Variable can be bound to one Variable in the Agent's Knowledge. The type of the Variable *startlocation* is *location*. The only Variable in the Agent's Knowledge that has the type *location* is the *location* Variable. Additionally the Interactive Move input *time* can only be bound to the

time in the Agent's Knowledge (based on the type). Thus the ActionKnowledge relation would be as follows.

$\{(startlocation, location), (time, time)\} = ActionKnowledge$

It should be noted that the relation is an ordered pair. Thus in the relation Action-Knowledge, the first element in the binding pair is the Action input and the second element in the pair is the Agent's Knowledge.

In addition to the binding from the Action to the Knowledge of the Agent there is a binding from the Agent's Knowledge to the inputs of the Goal. These bindings allows information to flow from the Goal into the Agent's Knowledge. The KnowledgeGoal relation is defined in Relation 15 and Restriction 24 restricts the KnowledgeGoal relation to be a binding.

Relation 15 (*K-G Relation*)

$KnowledgeGoal(K\text{-}G) \subseteq Agent.knowledge \times Goal.inputs$

Restriction 24 (*K-G is Binding*)

$KnowledgeGoal \subset binds$

In Figure C.1 there are two pairs of variables that could be bound. The two variables in the Move Goal that have the type *time* are *starttime* and *duration*. In the successful binding the WorkerAgents knowledge of *time* is bound to the Goals input *starttime*. Thus the KnowledgeGoal relation would be as follows.

$(time, starttime) = KnowledgeGoal$

The previous two bindings defined how information can move from the Goal to the Agent's Knowledge and how information can move from the Agent's Knowledge to the inputs of an Action. This section defines how these two bindings can be composed. The compositional binding provides all the necessary inputs that are required for an Action. An Agent can verify that an Action's precondition is valid, if the Agent is given the compositional binding. This binding will also be used in the proof that the Action will satisfy the Goal of the Agent (postcondition proof). The ActionKnowledgeGoal relation is defined in

Relation 16. Restriction 25 ensures that the ActionKnowledgeGoal relation is a binding.

Relation 16 (*A-K-G Relation*)

$ActionKnowledgeGoal \subseteq Action.inputs \times (Agent.knowledge \cup Goal.inputs)$

Restriction 25 (*A-K-G is a Binding*)

$ActionKnowledgeGoal \subset binds$

Restriction 26 ensures that the composition is part of the binding. This guarantees that if the previous two bindings are defined then the composition will be part of the Action-KnowledgeGoal relation.

Restriction 26 (*A-K Composition*)

$ActionKnowledge \circ KnowledgeGoal \subseteq ActionKnowledgeGoal$

The composition restriction (Restriction 26) gives the guarantee that if there is a binding from $x \rightarrow y$ in ActionKnowledge and $y \rightarrow z$ in Knowledge Goal, then the composition $x \rightarrow z$ must be in ActionKnowledgeGoal binding. Restriction 26, shows that the composition is a subset, therefore there can be additional bindings that map Action inputs directly to the Goal information.

The example ActionKnowledgeGoal binding for the Interactive Move example is as follows. In the ActionKnowledge there was a binding between *InteractiveMove.time* and *WorkerAgent.time*. Also in the KnowledgeGoal binding there was a binding between *WorkerAgent.time* and *Move.starttime*. The composition [(*InteractiveMove.time*,*Move.starttime*)] is part of the ActionKnowledgeGoal relation. The successful binding is shown below.

Example 5 (*ActionKnowledgeGoal binding example*)

$\{(InteractiveMove.currentlocation, Agent.location),$
$(InteractiveMove.startlocation, Agent.location),$
$(InteractiveMove.finishlocation, Move.location),$
$(InteractiveMove.duration, Goal.duration),$
$(InteractiveMove.starttime, Goal.starttime),$

$(InteractiveMove.time, Agent.time)\}$
$= ActionKnowledgeGoal$

C.4.2 Goal Output Bindings

The Goal state is the state that the agent is trying to reach. The Interaction Framework provides a mechanism for creating proofs. These proofs can show whether or not a Goal can be achieved by an Action. The required input to create a proof is an input binding and an output binding. This section defines how the required output binding can be created. The first output binding defined is from the Goal of the Agent to the Knowledge of the Agent (Relation 17).

Relation 17 (*G-K*)

$GoalKnowledge \subseteq Variables(Goal.state) \times Agent.Knowledge$

Restriction 27 (*G-K is Binding*)

$GoalKnowledge \subset binds$

In the Move Example (Figure C.1) there are two outputs for the goal and there are two pieces of Knowledge in the Agent. The GoalKnowledge binding is as follows.

$\{(finishtime, time), (finishlocation, location)\} = GoalKnowledge$

The next binding binds the Knowledge of the Agent to the outputs of the Action. This binding models how an Action can affect the Knowledge of the Agent. This binding is in Relation 18 and it is also a binding (Restriction 28).

Relation 18 (*K-A*)

$KnowledgeAction \subseteq Agent.knowledge \times Variables(Action.postcondition)$

Restriction 28 (*K-A is Binding*)

$KnowledgeAction \subset binds$

There are two outputs for the Interactive Move Interaction and there are two elements in the Agent's Knowledge. There is only one binding for the outputs, which is listed below.

$\{(time, time), (location, finishlocation)\} = KnowledgeAction$

The two above relations can be combined using a transitive binding function that goes from the Action to the Goal. This relation is defined in Relation 19 and the valid binding restriction is in Restriction 29.

Relation 19 (*G-A*)

$GoalAction \subseteq Action.outputs \times (Agent.knowledge \cup Goal.outputs)$

Restriction 29 (*G-A is Binding*)

$GoalAction \subset binds$

The Goal Action binding includes the composition (Restriction 30), just as the Action-KnowledgeGoal included the composition in the relation.

Restriction 30 (*Goal Action Composition*)

$GoalAction \subseteq GoalKnowledge \circ KnowledgeAction$

In The Interactive Move Example there are two transitively composed bindings. The Goal Action binding, derived from the previous two bindings, is listed below.

Example 6 (*Goal Action bindings for Interactive Move output binding*)

$\{(Move.time, InteractiveMove.finishtime),$
$(Move.finishlocation, InteractiveMove.finishlocation)\} = GoalAction$

C.4.3 Goal-Action Bindings

The *GoalActionBindings* is a relation between Goals, Actions, and bindings. The first binding is to bind the input binding. The second binding binds the output of the Action to the desired state of the Goal. The tuple in Relation 20 defines a way to represent the possible permutations of Goals, Actions and Agents. For each of these tuples, there exists a pair of input and output bindings. An Agent can take a valid member of the Relation GoalActionBindings and verify that the following is true: If the Agent performs the Action then the Goal will be satisfied. The bindings provide Agents with the ability to prove whether or not the Goal Satisfaction is true or false. The Autobindings Chapter (Section

4) relaxed this condition such that it reads: If the Agent performs the Action then the Goal **may** be satisfied.

Relation 20 (*Goal Action Agent Bindings Tuple*)

$GoalActionBindings \subseteq Goal \times Action \times Agent \times ActionKnowledgeGoal \times GoalAction$

Section A.6 defines some additional restrictions on Relation 20. This restriction combines additional relations, thus that restriction definition is deferred.

C.5 Interaction Role Bindings

The Interaction Role bindings are binding that are required in order to decide if an Interaction role will satisfy an Agents' Goal. The information will flow from the agent and the goal, into the Interaction Role. The results of the Interaction Role will flow to the agent and satisfy a Goal, or a set of Goals. These bindings are defined under the assumption that we are given an Interaction Role , a Goal, and an Agent. These bindings are essentially the same as the bindings from the Action Bindings, but with the Interaction Role replacing Action for the bindings.

C.5.1 InteractionRole Input Bindings

The Interaction Role Knowledge (IR-K) binding (Relation 21) provides a mapping from the inputs of the Interaction Role to the Agents Knowledge. The binding restriction is in Restriction 31.

Relation 21 (*IR-K*)

$InteractionRoleKnowledge \subseteq InteractionRole.inputparameters \times Agent.knowledge$

Restriction 31 (*IR-K is Binding*)

$InteractionRoleKnowledge \subset binds$

In addition to the binding from the Interaction Role to the knowledge of the Agent, there is also a need for a binding from the Agents knowledge to the input parameters of the goal.

This relation is defined in Relation 22. The binding restriction is defined in Restriction 32. This binding will help to prove that the state of the Goal can be achieved.

Relation 22 (*K-G*)

$KnowledgeGoal \subseteq Agent.knowledge \times Goal.inputparameters$

Restriction 32 (*K-G is Binding*)

$KnowledgeGoal \subset binds$

The binding in Relation 23 will map the inputs of the Interaction Role to inputs from the agents knowledge and values from the goal.

Relation 23 (*IR-K-G*)

$InteractionRoleKnowledgeGoal \subseteq InteractionRole.inputparameters \times (Agent.knowledge \cup Goal.inputparameters)$

There are two restrictions on the InteractionRoleKnowledgeGoal binding. The first is that it is a valid binding.

Restriction 33 (*IR-K-G is Binding*)

$InteractionRoleKnowledgeGoal \subset binds$

The second is that the composition must be in the binding.

Restriction 34 (*IR-K-G Composition*)

$InteractionRoleKnowledge \circ KnowledgeGoal \subseteq InteractionRoleKnowledgeGoal$

C.5.2 Goal-Interaction Role Output Bindings

Relation 24 (*G-K*)

$GoalKnowledge \subseteq Variables(Goal.state) \times Agent.Knowledge$

Restriction 35 (*G-K is Binding*)

$GoalKnowledge \subset binds$

Binding the knowledge of the Agent to the output of the Interaction Role defines how the Interaction Role will affect the knowledge of the agent.

Relation 25 (*K-IR*)

$KnowledgeInteractionRole \subseteq Agent.knowledge \times Variables(InteractionRole.postcondition)$

Restriction 36 (*K-IR is Binding*)

$KnowledgeInteractionRole \subset binds$

The above function can be combined to define a transitive function that goes from the InteractionRole to the Goal.

Relation 26 (*G-IR*)

$GoalInteractionRole \subseteq Variables(InteractionRole.postcondition) \times Variables(Goal.state)$

Restriction 37 (*G-IR is Binding*)

$GoalInteractionRole \subset binds$

Restriction 38 (*G-IR Composition*)

$GoalInteractionRole \supseteq GoalKnowledge \circ KnowledgeInteractionRole$

C.5.3 Goal-Interaction Role Bindings

The *GoalInteractionRoleBindings* is a relation from goals to Interactions , and a tuple of bindings. The first binding is to bind the parameters of the Action to the Knowledge of the Agent and the parameters of the Goal. The second binding binds the output of the Action to the desired state of the Goal.

Relation 27 (*Goal Interaction Role Bindings*)

$GoalInteractionRoleBindings \subseteq Goal \times Interaction\ Role\ \times Agent \times$
$InteractionRoleKnowledgeGoal \times GoalInteractionRole$

The Goal Interaction Role tuple defined in Relation 27 returns the same type of tuple as define in the Section on Goal-Action Bindings(Section C.4.3). This tuple states that if the Agent plays the Interaction Role then the agent can verify the achievement of the Goal, using the pair of bindings.

C.6 Interaction Protocol Bindings

In Sections C.4.1 and C.5 a binding was defined from the Goal to the Action/Interaction/Interaction Role and back again. A method to show how Interactions are realized through a Protocol is needed. This section lays out the framework for this method.

C.6.1 Interaction Protocol Input Bindings

Each the Protocol has a set of input parameters. The input parameters define how the Protocol will be tailored. When performing the Protocol, the values for all of the inputs is required. Given the values for all the inputs, and the protocol's properties are valid when the protocol starts, then there are properties that will be true at the end of the protocol. A binding for each of the input parameter in protocol is needed. This done through the *ProtocolInteraction* Binding shown in Relation 28. The model also ensures that this relation is a binding in Restriction 39.

Relation 28 (*P-I*)

$ProtocolInteraction \subseteq Protocol.inputparamaters \times Variables(Interaction)$

Restriction 39 (*P-I is Binding*)

$ProtocolInteraction \subset binds$

C.6.2 Protocol Interaction Output Bindings

The outputs work in a similar manner as the inputs. The model does not require a binding for every output variable in the Protocol. The Protocol is allowed to have extra information(output parameters) that are not needed or bound to the output variables of the Interaction.

Relation 29 (*I-P*)

$InteractionProtocol(I\text{-}P) \subseteq Interaction.outputparameters \times Variables(Protocol.properties)$

Restriction 40 (*I-P is Binding*)

InteractionProtocol $\subset$ *binds*

Relation 29 defines a way to prove that the properties of the protocol prove that the Interaction is achieved. There is a conversion that must be performed on the protocol properties in order to derive a proof from those properties. In Section 3.4 roles were defined for the conversion process. This process is also used in the valid precondition check for the Protocol and Interaction.

C.7 Binding Proofs

The previous sections defined the bindings between the different entities within the Interaction Model. This section defines how a group of bindings can be used to modified a specification expression by applying a substitution algorithm. The notation for the input and output bindings uses the following format $\frac{b_1,b_2}{e_1 \Longrightarrow e_2}$.

To guide the reader a concrete example is given. This example should make the concepts in this chapter more clear. In the example expression listed in Table C.7, there are two expressions: e_1 and e_2. Expression e_1 has the variables $\{a, b, c\}$ and expression e_2 has variables $\{x, y\}$ The bindings are listed it Table C.7. In binding b_1 a maps to x. In binding b_2 y maps to b. Binding b_1 is the input binding for the expressions, and binding b_2 is the output binding.

$$\begin{array}{lcl} e_1 & = & (a = 10) \wedge (b < 20) \wedge (c = 100) \\ e_2 & = & (x = 10) \wedge (y < 20) \end{array}$$

Table C.1: *Example Expression*

$$\begin{array}{llll} b_1 & a & \mapsto & \text{x} \\ b_2 & y & \mapsto & \text{b} \end{array}$$

Table C.2: *Example Binding*

Given the above expressions, it should be possible to show that $\frac{b_1, b_2}{e_1 \Longrightarrow e_2}$ holds. There are a few restrictions that are placed on the expressions and bindings. Restriction 41 defines the input binding restriction. This restriction ensures that the domain of the input binding is a subset of the variables in the expression on the right hand side of the implication. In the example the input binding $b_1 = x \mapsto a$. The domain of b_1 is $\{a\}$, and the variables in the expression e_1 are $\{a, b, c\}$. Thus the first half of the restriction holds as $\{a\} \subseteq \{a, b, c\}$. The range of b_1 is $\{y\}$ and and the variables in expression e_2 are $\{x, y\}$ and thus the second half of the restriction also holds ($\{y\} \subseteq \{x, y\}$). This restriction ensures that the input binding maps variables in e_1 to variables in e_2.

Restriction 41 (*Input Binding*)

$$Domain(b_1) \subseteq Variables(e_1) \wedge Range(b_1) \subseteq Variables(e_2)$$

There is a similar restriction on the output binding. Restriction 42 lists this restriction. This restriction ensure that the output binding maps variables from e_2 to e_1. In the example the domain of binding b_2 is $\{y\}$ and the variables in expression e_2 are $\{x, y\}$, and thus the first half of the restriction holds ($\{y\} \subseteq \{x, y\}$). The range of binding b_2 is $\{b\}$ and the variables in expression e_1 are $\{a, b, c\}$ and thus the second half of the restriction holds ($\{b\} \subseteq \{a, b, c\}$).

Restriction 42 (*Output Binding*)

$$Domain(b_2) \subseteq Variables(e_2) \wedge Range(b_2) \subseteq Variables(e_1)$$

The binding notation allows allows the proof to utilize the input binding b_1 and the output binding b_2. In order to take specification expressions from two different entities there are two required steps. The first is the binding of variables, such that the variables have the same meaning and value in both expressions. The second is the substitution of those variables in the expressions. To provide this substitution, a method of generating variables that are fresh must be defined. A fresh variable is a variable that has not already been bound. In the example, the bound (or declared) variables are $\{x, y, a, b, c\}$. A function can be defined that generates fresh variables for the substitution process. This definition is

shown in Definition 21.

Definition 21 (*Fresh Variables*)

$$fresh(e_1 \dots e_n) = \exists x : Variables | x \notin Variables(e_1 \dots e_n)$$

When trying to prove that the $e_1 \implies e_2$ there is a substitution process. This process creates a new expression that can be used in a proof. The process begins by substituting for each binding in b_1 and then the binding b_2. An example algorithm for the substitution is shown in Algorithm 11. Definition 22 formally defined how to take a binding and substitute a fresh variable for the pair of bound input variables. Definition 22 defines that if v_1, v_2 are bound together, then a substitution with a fresh variable f for v_1 in expression e_1 should be done. Additionally the same variable f should be used in the substitution for v_2 in e_2.

Definition 22 (*Input Substitution*)

$$\forall v_1, v_2 : Variables | (v_1, v_2) \in b_1 \implies \exists e : Expression, f : fresh(e_1, e_2) | e = [v_1 \mapsto f]e_1 \implies [v_2 \mapsto f]e_2$$

Definition 23 formally defines how to take a binding and substitute a fresh variable for pair of bound output variables. Definition 23 defines that if v_1, v_2 are bound together, then the algorithm must substitute a fresh variable f for v_1 in expression e_2 and it must use that same variable f to substitute for v_2 in e_1.

Definition 23 (*Output Substitution*)

$$\forall v_1, v_2 : Variables | (v_1, v_2) \in b_2 \implies \exists e : Expression, f : fresh(e_1, e_2) | e = [v_2 \mapsto f]e_1 \implies [v_1 \mapsto f]e_2$$

Algorithm 11 iterates through all the input bindings (Lines 3 through 7). On Line 4 a fresh variable f_1 (fresh in the context of e_1, e_2) is generated. On Line 5 the algorithm substitutes the fresh variable f_1 for the variable v_1 in expression e_1. On Line 6 the algorithm substitutes f_1 for variable v_2 in expression e_2. The second half of the algorithm iterates through all the output bindings (Lines 9 through 13). The algorithm then generates a fresh variable f_2 on Line 10. On Line 11 the algorithm substitutes the fresh variable f_2 for the value v_2 in expression e_1. Line 12 substitutes f_2 for v_1 in expression e_2. The pair of newly substituted

expressions is returned by the algorithm.

Algorithm 11 Substitution Algorithm

```
1: Substitute(e_1, e_2, b_1, b_2)
2: {Do the substitution for the input bindings}
3: for all (v_1, v_2) ∈ b_1 do
4:     f_1 = fresh(e_1, e_2)
5:     e_1 = [v_1 ↦ f_1] e_1
6:     e_2 = [v_2 ↦ f_1] e_2
7: end for
8: {Do the substitution for the output bindings}
9: for all (v_1, v_2) ∈ b_2 do
10:    f_2 = fresh(e_1, e_2)
11:    e_1 = [v_2 ↦ f_2] e_1
12:    e_2 = [v_1 ↦ f_2] e_2
13: end for
14: return e_1, e_2
```

The results of Algorithm 11 can be applied to the example proof. For simplicity, the fresh variable generator will generate new strictly increasing hexadecimal variable names. By applying the input substitution (Lines 3 through 7), using the fresh variable 0x1, the following specification expressions is created (Table C.7).

$$\begin{array}{lcl} e_1 & = & (0\text{x}1 = 10) \wedge (b < 20) \wedge (c = 100) \\ e_2 & = & (0\text{x}1 = 10) \wedge (y < 20) \end{array}$$

Table C.3: *Example Expression Input Binding Substitution*

The output substitution (Lines 9 through 13) can then be applied to the expressions e_1, e_2. The result is shown in Table C.7

$$\begin{array}{lcl} e_1 & = & (0\text{x}1 = 10) \wedge (0\text{x}2 < 20) \wedge (c = 100) \\ e_2 & = & (0\text{x}1 = 10) \wedge (0\text{x}2 < 20) \end{array}$$

Table C.4: *Example Expression Input and Output Binding Substitution*

The prove that $e_1 \implies e_2$ can be easily verified by hand.

C.7.1 Binding shorthand notation

In order to make some of the restrictions more readable, a short hand notation for the bindings was created. If there is only one binding, then only the first substitution 22 is required. For example in $\frac{b_1}{e_1 \implies e_2}$ there is only substitute for the input b_1. This type of substitution is used to check that preconditions are satisfiable.

Expression that do not fit the above pattern, can be defined using the alternate notation. This notation denotes which direction the expression derives its variables from.

Definition 24 (*Directed Binding*)

$$DirectedBinding = (\overrightarrow{b} || \overleftarrow{b})$$

In Definition 24 there can either be an input binding denoted by the right arrow or there can be an output binding denoted by the left arrow. Definition 25 states that if there is an expression e, then that expression can be followed with a set of directed bindings.

Definition 25 (*Binding Set with Expression*)

$$e\{Set(DirectedBinding)\}$$

The example expression $(a\{\overrightarrow{x}\} \wedge b\{\overleftarrow{y}\} \implies c\{\overrightarrow{x}\})$ has the following meaning: a and c use the input binding x and expression b uses the output binding y.

Appendix D

Results

This chapter lists results that were found by running algorithms designed for this thesis.

D.1 Auction Example Verification

```
DutchAuctionBuyer and BuyerRole ARE compatible
GOAL Buy Item Goal and SellAnItemInteraction are compatible
BuyingAgent DutchAuctionBuyer and BuyerRole are compatible Buy Item Goal
BuyingAgent DutchAuctionSeller and BuyerRole NOT compatible
SellerAgent EnglishAuctionBuyer and SellerRole NOT compatible
EnglishAuctionSeller and SellerRole ARE compatible
GOAL Sell Item Goal and SellAnItemInteraction are compatible
SellerAgent EnglishAuctionSeller and SellerRole are compatible Sell Item Goal
EnglishAuctionBuyer and BuyerRole ARE compatible
GOAL Sell Item Goal and SellAnItemInteraction are compatible
SellerAgent EnglishAuctionSeller and BuyerRole NOT compatible
BuyingAgent EnglishAuctionBuyer and SellerRole NOT compatible
EnglishAuctionSeller and SellerRole ARE compatible
GOAL Buy Item Goal and SellAnItemInteraction are compatible
EnglishAuctionBuyer and BuyerRole ARE compatible
GOAL Buy Item Goal and SellAnItemInteraction are compatible
```

```
BuyingAgent EnglishAuctionBuyer and BuyerRole are compatible Buy Item Goal

BuyingAgent EnglishAuctionSeller and BuyerRole NOT compatible

----------------------

Agent:SellerAgent Goal:Sell Item Goal Interaction:SellAnItemInteraction InteractionRole:

Agent:BuyingAgent Goal:Buy Item Goal Interaction:SellAnItemInteraction InteractionRole:B

Agent:SellerAgent Goal:Sell Item Goal Interaction:SellAnItemInteraction InteractionRole:

Agent:BuyingAgent Goal:Buy Item Goal Interaction:SellAnItemInteraction InteractionRole:B

Time: 473 seconds
```

Listing D.1: *Simulation 0 Computer log*

```
BROADCAST!protocol:0,1,2
 Reading
BROADCAST!protocol:0,1,2
 Reading
BROADCAST!protocol:0,1,2
 Reading
SELECTED PROTOCOL!QRPROTOCOL
QR+WRITE
gps:0    #Send GPS Goal Request
QR+READ
 Latitude = 39.12 Longitude = -96.71 #Recieve GPS Location
QR+ACK
QR+WAIT
QR+ACKTIMEOUT #FINISH
BROADCAST!protocol:0,1,2
 Reading
BROADCAST!protocol:0,1,2
 Reading
SELECTED PROTOCOL!QRPROTOCOL
QR+WRITE
gps:0
QR+READ
 Latitude = 39.12266577859547 Longitude = -96.71365547516562
QR+ACK
QR+WAIT
QR+ACKTIMEOUT
BROADCAST!protocol:0,1,2
 Reading
BROADCAST!protocol:0,1,2
 Reading

SELECTED PROTOCOL!PCQRMobileWIFI
OPTQR+WRITE
gps:0
OPTQR+READ
BROADCAST!protocol:0,1,2
 Reading
BROADCAST!protocol:0,1,2
 Reading
SELECTED PROTOCOL!QRPROTOCOL
QR+WRITE
gps:0
QR+READ
```

```
44 Latitude = 39.12271015460612 Longitude = -96.71367765800677
45 QR+ACK
46 QR+WAIT
47 QR+ACKTIMEOUT
48 BROADCAST!protocol:0,1,2
49 Reading
50
51 SELECTED PROTOCOL!WIFI
52 WIFI!WRITE
53 wifi:gps:0
54 WIFI!READ
55 Latitude = 39.12271015460614 Longitude = -96.71367765800676
56 BROADCAST!protocol:0,1,2
57 Reading
58 BROADCAST!protocol:0,1,2
59 Reading
60 WIFITIMEOUT
61 BROADCAST!protocol:0,1,2
62 Reading
```

Listing D.2: *Simulation 0 Tablet log*

```
1  I/PHDCameraLog(10809): 1320966388451    READ
2  I/PHDCameraLog(10809): 1320966389473    WAIT
3  I/PHDCameraLog(10809): 1320966406481    PROCESS
4  I/PHDCameraLog(10809): 1320966406501    DATA!protocol:0,1,2
5  I/PHDCameraLog(10809): 1320966406502    WRITE
6  I/PHDCameraLog(10809): 1320966406503    OUTPUT!0
7  I/PHDCameraLog(10809): 1320966407123    DONE
8  I/PHDCameraLog(10809): 1320966421521
9  I/PHDCameraLog(10809): Optical Protocol
10 I/PHDCameraLog(10809): 1320966421536    READ
11 I/PHDCameraLog(10809): 1320966423910    WAIT
12 I/PHDCameraLog(10809): 1320966427913    READDONE
13 I/PHDCameraLog(10809): 1320966427933    gps:0 #Recieve Request for GPS
14 I/PHDCameraLog(10809): 1320966427933    PARSE!gps:0
15 I/PHDCameraLog(10809): 1320966427934    WRITE
16 I/PHDCameraLog(10809): 1320966427934    TEXT=Latitude = 39.12 Longitude = -96.71 #WRITE GPS Location
17 I/PHDCameraLog(10809): 1320966442923    WAIT
18 I/PHDCameraLog(10809): 1320966484936    TIMEOUT
19 I/PHDCameraLog(10809): 1320966484947    DONE #FINISH
20 I/PHDCameraLog(10809): 1320966485085
21 I/PHDCameraLog(10809): Negotiate
22 I/PHDCameraLog(10809): 1320966485085    READ
23 I/PHDCameraLog(10809): 1320966486110    DATA!protocol:0,1,2
24 I/PHDCameraLog(10809): 1320966486141    PROCESS
25 I/PHDCameraLog(10809): 1320966486141    WRITE
26 I/PHDCameraLog(10809): 1320966486142    OUTPUT!0
27 I/PHDCameraLog(10809): 1320966486671    DONE
28 I/PHDCameraLog(10809): 1320966487626    READ
29 I/PHDCameraLog(10809): 1320966487654
30 I/PHDCameraLog(10809): Optical Protocol
31 I/PHDCameraLog(10809): 1320966488793    WAIT
32 I/PHDCameraLog(10809): 1320966504799    READDONE
33 I/PHDCameraLog(10809): 1320966504829    gps:0
34 I/PHDCameraLog(10809): 1320966504829    PARSE!gps:0
35 I/PHDCameraLog(10809): 1320966504829    WRITE
36 I/PHDCameraLog(10809): 1320966504830    TEXT=Latitude = 39.12266577859547 Longitude = -96.71365547516562
37 I/PHDCameraLog(10809): 1320966519815    WAIT
38 I/PHDCameraLog(10809): 1320966549837    DONE
39 I/PHDCameraLog(10809): 1320966549838    TIMEOUT
40 I/PHDCameraLog(10809): 1320966549978
41 I/PHDCameraLog(10809): Negotiate
42 I/PHDCameraLog(10809): 1320966550023    READ
43 I/PHDCameraLog(10809): 1320966551004    WAIT
```

```
I/PHDCameraLog(10809): 1320966556013    PROCESS
I/PHDCameraLog(10809): 1320966556040    WRITE
I/PHDCameraLog(10809): 1320966556040    DATA!protocol:0,1,2
I/PHDCameraLog(10809): 1320966556040    OUTPUT!2
I/PHDCameraLog(10809): 1320966556541    DONE
I/PHDCameraLog(10809): 1320966557236
I/PHDCameraLog(10809): Comp Opt Read, Tablet Wifi Read Protocol
I/PHDCameraLog(10809): 1320966557260    READ
I/PHDCameraLog(10809): 1320966557261    WAIT
I/PHDCameraLog(10809): 1320966567247    gps:0
I/PHDCameraLog(10809): 1320966567247    READDONE
I/PHDCameraLog(10809): 1320966567247    PARSE!gps:0
I/PHDCameraLog(10809): 1320966567247    WRITE
I/PHDCameraLog(10809): 1320966567248    TEXT=Latitude = 39.122692877441295 Longitude = -96.71367846746864
I/PHDCameraLog(10809): 1320966582245    WAIT
I/PHDCameraLog(10809): 1320966618255    TIMEOUT
I/PHDCameraLog(10809): 1320966618274    DONE
I/PHDCameraLog(10809): 1320966618436
I/PHDCameraLog(10809): Negotiate
I/PHDCameraLog(10809): 1320966618437    READ
I/PHDCameraLog(10809): 1320966619431    WAIT
I/PHDCameraLog(10809): 1320966665510    PROCESS
I/PHDCameraLog(10809): 1320966665550    DATA!protocol:0,1,2
I/PHDCameraLog(10809): 1320966665550    WRITE
I/PHDCameraLog(10809): 1320966665550    OUTPUT!0
I/PHDCameraLog(10809): 1320966666054    DONE
I/PHDCameraLog(10809): 1320966666962
I/PHDCameraLog(10809): Optical Protocol
I/PHDCameraLog(10809): 1320966666986    READ
I/PHDCameraLog(10809): 1320966668149    WAIT
I/PHDCameraLog(10809): 1320966676150    READDONE
I/PHDCameraLog(10809): 1320966676174    gps:0
I/PHDCameraLog(10809): 1320966676174    PARSE!gps:0
I/PHDCameraLog(10809): 1320966676174    WRITE
I/PHDCameraLog(10809): 1320966676175    TEXT=Latitude = 39.12271015460612 Longitude = -96.71367765800677
I/PHDCameraLog(10809): 1320966691159    WAIT
I/PHDCameraLog(10809): 1320966729171    TIMEOUT
I/PHDCameraLog(10809): 1320966729228    DONE
I/PHDCameraLog(10809): 1320966729361
I/PHDCameraLog(10809): Negotiate
I/PHDCameraLog(10809): 1320966729361    READ
I/PHDCameraLog(10809): 1320966730355    WAIT
I/PHDCameraLog(10809): 1320966735585    PROCESS
I/PHDCameraLog(10809): 1320966735585    DATA!protocol:0,1,2
I/PHDCameraLog(10809): 1320966735585    WRITE
I/PHDCameraLog(10809): 1320966735586    OUTPUT!1
I/PHDCameraLog(10809): 1320966736085    DONE
I/PHDCameraLog(10809): 1320966736522
I/PHDCameraLog(10809): WIFI Protocol
I/PHDCameraLog(10809): 1320966736523    READ
I/PHDCameraLog(10809): 1320966736523    WAIT
I/PHDCameraLog(10809): 1320966744521    READDONE
I/PHDCameraLog(10809): 1320966744554    wifi:gps:0
I/PHDCameraLog(10809): 1320966744555    WRITE
I/PHDCameraLog(10809): 1320966744555    wifi:Latitude = 39.12271015460614 Longitude = -96.71367765800676
I/PHDCameraLog(10809): 1320966744555    WAIT
```

Listing D.3: *Simulation 1 Computer log*

```
BROADCAST!protocol:2
Reading
Failed MobileQRRead
Failed MobileWifiRead
Complete Failure
Failed MobileQRRead
```

```
7  Failed MobileQRWrite
8  BROADCAST!protocol:1
9  Reading
10 SELECTED PROTOCOL!WIFI
11 WIFI!WRITE
12 wifi:gps:0
13 WIFI!READ
14 GyroEvent [x=0.16808061, y=-0.1536889, z=-0.14244157]
15 Failed PCQRWrite
16 Failed MobileWifiWrite
17 Complete Failure
18 Failed MobileQRWrite
19 Failed PCQRRead
20 BROADCAST!protocol:1,2
21 Reading
22 SELECTED PROTOCOL!WIFI
23 WIFI!WRITE
24 wifi:gps:0
25 WIFI!READ
26 GyroEvent [x=-5.326322E-7, y=-5.326322E-7, z=-5.326322E-7]
27 Failed MobileQRWrite
28 Failed PCQRWrite
29 BROADCAST!protocol:1
30 Reading
31 BROADCAST!protocol:1
32 Reading
33 BROADCAST!protocol:1
34 Reading
35 SELECTED PROTOCOL!WIFI
36 WIFI!WRITE
37 wifi:gps:0
38 WIFI!READ
39 GyroEvent [x=-5.326322E-7, y=-5.326322E-7, z=-5.326322E-7]
40 Failed MobileWifiRead
41 Failed MobileWifiWrite
42 BROADCAST!protocol:0
43 Reading
44
45 SELECTED PROTOCOL!QRPROTOCOL
46 QR+WRITE
47 gps:0
48 QR+READ
49 EMPTY
50 QR+ACK
51 QR+WAIT
52 QR+ACKTIMEOUT
53 Failed MobileWifiRead
54 Failed MobileQRRead
55 Complete Failure
56 Failed PCWifiRead
57 Failed PCQRRead
58 Complete Failure
59 Failed MobileQRWrite
60 Failed MobileWifiRead
61 BROADCAST!protocol:2
62 Reading
63 SELECTED PROTOCOL!PCQRMobileWIFI
64 OPTQR+WRITE
65 gps:0
66 OPTQR+READ
67 EMPTY
68 OPTQR+ACK
69 OPTQR+WAIT
70 OPTQR+ACKTIMEOUT
71 Failed PCWifiWrite
```

```
72 Failed PCQRRead
73 BROADCAST!protocol:2
74 Reading
75 BROADCAST!protocol:2
76 Reading
```

Listing D.4: *Simulation 1 Tablet log*

```
1  I/PHDCameraLog(16699): 1321984875892    READ
2  I/PHDCameraLog(16699): 1321984876892    WAIT
3  I/PHDCameraLog(16699): 1321984884942    PROCESS
4  I/PHDCameraLog(16699): 1321984884949    DATA!protocol:1
5  I/PHDCameraLog(16699): 1321984884960    WRITE
6  I/PHDCameraLog(16699): 1321984884993    OUTPUT!1
7  I/PHDCameraLog(16699): 1321984885821    DONE
8  I/PHDCameraLog(16699): 1321984886189
9  I/PHDCameraLog(16699): WIFI Protocol
10 I/PHDCameraLog(16699): 1321984886189    READ
11 I/PHDCameraLog(16699): 1321984886189    WAIT
12 I/PHDCameraLog(16699): 1321984894213    READDONE
13 I/PHDCameraLog(16699): 1321984894235    wifi:gps:0
14 I/PHDCameraLog(16699): 1321984894264    WRITE
15 I/PHDCameraLog(16699): 1321984894264    wifi:GyroEvent [x=0.16808061, y=-0.1536889, z=-0.14244157]
16 I/PHDCameraLog(16699): 1321984894264    WAIT
17 I/PHDCameraLog(16699): 1321984914633
18 I/PHDCameraLog(16699): Negotiate
19 I/PHDCameraLog(16699): 1321984914657    READ
20 I/PHDCameraLog(16699): 1321984915692    WAIT
21 I/PHDCameraLog(16699): 1321984919934    PROCESS
22 I/PHDCameraLog(16699): 1321984919934    DATA!protocol:1,2
23 I/PHDCameraLog(16699): 1321984919935    WRITE
24 I/PHDCameraLog(16699): 1321984919935    OUTPUT!1
25 I/PHDCameraLog(16699): 1321984920699    DONE
26 I/PHDCameraLog(16699): 1321984920844
27 I/PHDCameraLog(16699): WIFI Protocol
28 I/PHDCameraLog(16699): 1321984920893    READ
29 I/PHDCameraLog(16699): 1321984920893    WAIT
30 I/PHDCameraLog(16699): 1321984928906    READDONE
31 I/PHDCameraLog(16699): 1321984928923    wifi:gps:0
32 I/PHDCameraLog(16699): 1321984928948    WRITE
33 I/PHDCameraLog(16699): 1321984928948    wifi:GyroEvent [x=-5.326322E-7, y=-5.326322E-7, z=-5.326322E-7]
34 I/PHDCameraLog(16699): 1321984928950    WAIT
35 I/PHDCameraLog(16699): 1321984949205
36 I/PHDCameraLog(16699): Negotiate
37 I/PHDCameraLog(16699): 1321984949274    READ
38 I/PHDCameraLog(16699): 1321984950252    WAIT
39 I/PHDCameraLog(16699): 1321984954337    PROCESS
40 I/PHDCameraLog(16699): 1321984954337    DATA!protocol:1
41 I/PHDCameraLog(16699): 1321984954337    WRITE
42 I/PHDCameraLog(16699): 1321984954338    OUTPUT!1
43 I/PHDCameraLog(16699): 1321984955094    DONE
44 I/PHDCameraLog(16699): 1321984955646
45 I/PHDCameraLog(16699): WIFI Protocol
46 I/PHDCameraLog(16699): 1321984955688    READ
47 I/PHDCameraLog(16699): 1321984955712    WAIT
48 I/PHDCameraLog(16699): 1321984960727    READDONE
49 I/PHDCameraLog(16699): 1321984960778    wifi:gps:0
50 I/PHDCameraLog(16699): 1321984960814    WRITE
51 I/PHDCameraLog(16699): 1321984960814    wifi:GyroEvent [x=-5.326322E-7, y=-5.326322E-7, z=-5.326322E-7]
52 I/PHDCameraLog(16699): 1321984960815    WAIT
53 I/PHDCameraLog(16699): 1321984981122
54 I/PHDCameraLog(16699): Negotiate
55 I/PHDCameraLog(16699): 1321984981165    READ
56 I/PHDCameraLog(16699): 1321984982154    WAIT
57 I/PHDCameraLog(16699): 1321984989398    PROCESS
```

```
I/PHDCameraLog(16699): 1321984989398    DATA!protocol:0
I/PHDCameraLog(16699): 1321984989398    WRITE
I/PHDCameraLog(16699): 1321984990201    DONE
I/PHDCameraLog(16699): 1321984990202    OUTPUT!0
I/PHDCameraLog(16699): 1321984990334
I/PHDCameraLog(16699): Optical Protocol
I/PHDCameraLog(16699): 1321984990365    READ
I/PHDCameraLog(16699): 1321984991610    WAIT
I/PHDCameraLog(16699): 1321985005629    gps:0
I/PHDCameraLog(16699): 1321985005663    READDONE
I/PHDCameraLog(16699): 1321985005663    PARSE!gps:0 I/PHDCameraLog(16699): 1321985005687          WRITE
I/PHDCameraLog(16699): 1321985005687    TEXT=EMPTY
I/PHDCameraLog(16699): 1321985020682    WAIT
I/PHDCameraLog(16699): 1321985052702    TIMEOUT
I/PHDCameraLog(16699): 1321985052775    DONE
I/PHDCameraLog(16699): 1321985052999
I/PHDCameraLog(16699): Negotiate
I/PHDCameraLog(16699): 1321985052999    READ
I/PHDCameraLog(16699): 1321985053973    WAIT
I/PHDCameraLog(16699): 1321985058007    DATA!protocol:2
I/PHDCameraLog(16699): 1321985058008    PROCESS
I/PHDCameraLog(16699): 1321985058008    WRITE
I/PHDCameraLog(16699): 1321985058049    OUTPUT!2
I/PHDCameraLog(16699): 1321985058786    DONE
I/PHDCameraLog(16699): 1321985065792
I/PHDCameraLog(16699): Comp Opt Read, Tablet Wifi Read Protocol
I/PHDCameraLog(16699): 1321985065832    READ
I/PHDCameraLog(16699): 1321985065910    WAIT
I/PHDCameraLog(16699): 1321985067844    READDONE
I/PHDCameraLog(16699): 1321985067867    gps:0
I/PHDCameraLog(16699): 1321985067868    PARSE!gps:0
I/PHDCameraLog(16699): 1321985067869    WRITE
I/PHDCameraLog(16699): 1321985067889    TEXT=EMPTY
I/PHDCameraLog(16699): 1321985082869    WAIT
```