



**TRANSFORMING ANALYSIS MODELS
INTO DESIGN MODELS FOR THE
MULTIAGENT SYSTEMS ENGINEERING
(MASE) METHODOLOGY**

THESIS

Clint H. Sparkman, 1st Lieutenant, USAF

AFIT/GCS/ENG/01M-12

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

TRANSFORMING ANALYSIS MODELS INTO DESIGN
MODELS FOR THE MULTIAGENT SYSTEMS
ENGINEERING (MASE) METHODOLOGY

THESIS

Presented to the faculty of the Graduate School of Engineering & Management
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the Requirements for the Degree of
Master of Science

Clint H. Sparkman, B. S.

1st Lieutenant, USAF

March 2001

Approved for public release, distribution unlimited.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or United States Government.

TRANSFORMING ANALYSIS MODELS INTO DESIGN
MODELS FOR THE MULTIAGENT SYSTEMS
ENGINEERING (MASE) METHODOLOGY

THESIS

Clint H. Sparkman, B. S.
1st Lieutenant, USAF

Approved:



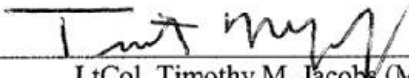
Maj. Scott A. DeLoach (Chairman)

21 Feb 01
date



Dr. Thomas C. Hartrum (Member)

21 Feb 01
date



LtCol. Timothy M. Jacobs (Member)

21 Feb 01
date

ACKNOWLEDGMENTS

I would first like to thank my Lord and Savior Jesus Christ for giving me the ability and strength to pursue this goal. Next, I must thank my wife, Casey. Words alone cannot express the love and admiration I have for her. Her sacrifice and devotion to our family and home allowed me focus on the work necessary to complete this graduate program, and she was an unwavering reminder of what is truly important in life. To my advisor, Maj Scott DeLoach, I extend a special thanks for his guidance and for his insightful and challenging feedback throughout this research. I would also like to thank my committee members, Lt Col Tim Jacobs and Dr Tom Hartrum, for their assistance during this thesis. I would also like to thank my fellow classmates for our discussions that provided insight and focus to my research. More importantly, their friendship and the laughter we shared made my time here bearable, even enjoyable. They will never be forgotten.

Clint Houston Sparkman

Table of Contents

ACKNOWLEDGMENTS..... IV

TABLE OF FIGURES IX

TABLE OF TABLES XIII

ABSTRACT XIV

ABSTRACT XIV

I. INTRODUCTION 1

 1.1 Background 3

 1.1.1 Capturing Goals 4

 1.1.2 Applying Use Cases 4

 1.1.3 Refining Roles 5

 1.1.4 Creating Agent Classes 6

 1.1.5 Constructing Conversations 7

 1.1.6 Assembling Agent Classes 8

 1.1.7 System Deployment 8

 1.1.8 agentTool 9

 1.2 Problem 9

 1.3 Scope 10

 1.4 Thesis Overview 11

II. PROBLEM APPROACH..... 13

 2.1 Expanding the Role Model..... 13

 2.2 Transforming Concurrent Tasks to Conversations and Components 15

 2.3 Model Definitions 18

 2.3.1 Analysis Models 19

 2.3.2 Design Models 21

2.3.2.1 Agents	22
2.3.2.2 Components.....	22
2.3.2.3 Conversations.....	23
2.3.3 State Tables.....	23
2.3.3.1 States	25
2.3.3.2 Transitions.....	26
2.3.3.2.1 Concurrent Task Diagram.....	27
2.3.3.2.2 Component State Table.....	28
2.3.3.2.3 Communication Class Diagram	28
2.3.3.3 Actions and Events.....	29
2.4 Summary.....	31
III. TRANSFORMATIONS	33
3.1 Formal Notations	34
3.2 Generating the Agent Model.....	35
3.2.1 Determining Protocols for External Events.....	36
3.2.2 Creating Components for Agents from Tasks.....	41
3.2.3 Replicating Protocols Between Components	42
3.2.4 Transforming External Events into Internal Events.....	44
3.3 Annotating Component State Diagrams.....	45
3.3.1 Splitting Transitions.....	46
3.3.2 Determining the Protocols for Transitions	50
3.3.3 Start Label for Transitions	56
3.3.4 End Label for Transitions	61
3.3.5 Matching Conversation Halves	63
3.3.6 Splitting Transitions with a ReceiveEvent and Multiple Conversation Names.....	67
3.3.7 Creating Conversations.....	68
3.3.8 Propagating the Set of Conversations	70
3.4 Harvesting the Conversations	71
3.4.1 Combining Conversation End States	72

3.4.2 Preparing Variables and Parameters	76
3.4.3 Initiator Conversation Halves	81
3.4.4 Responder Conversation Halves	85
3.4.5 Moving States and Transitions From Components to Conversations.....	88
3.5 Summary	91
IV. DEMONSTRATION	92
4.1 Transformation System Overview	92
4.2 Integration with agentTool.....	93
4.2.1 Transformation Classes.....	94
4.2.2 Model Classes.....	95
4.3 Example	95
4.3.1 Starting Point – Role Model and Initial Agent Classes.....	96
4.3.2 Stage One – Creating Agent Components.....	100
4.3.2.1 Determining the Protocols for External Events	100
4.3.2.2 Determining the Mode for the SearchRequest Protocol	102
4.3.2.3 Agent Components.....	103
4.3.3 Stage Two – Annotating Component State Diagrams.....	105
4.3.3.1 Matching up the First Messages of the Conversations.....	105
4.3.3.2 Annotated Component State Diagrams.....	107
4.3.4 Stage Three – Creating Conversations.....	109
4.4 Summary	113
V. CONCLUSIONS AND FUTURE WORK.....	114
5.1 Conclusions.....	114
5.2 Future Research Areas	115
5.2.1 Transformation Enhancements	115
5.2.2 Formal Transformations for Mixed-Initiative Systems	117
5.2.3 Formal Proof.....	117
5.3 Summary	118

VI. BIBLIOGRAPHY.....	119
APPENDIX A. BACKGROUND.....	121
A.1 Multiagent System Methodologies.....	121
A.1.1 Multiagent System Engineering Methodology.....	122
A.1.1.1 Capturing Goals.....	123
A.1.1.2 Applying Use Cases.....	124
A.1.1.3 Refining Roles.....	125
A.1.1.4 Creating Agent Classes.....	127
A.1.1.5 Constructing Conversations.....	129
A.1.1.6 Assembling Agent Classes.....	130
A.1.1.7 System Deployment.....	130
A.1.1.8 Transitioning from Analysis to design - MaSE.....	131
A.1.2 Gaia Methodology.....	132
A.1.2.1 Analysis Phase - Gaia.....	132
A.1.2.2 design Phase - Gaia.....	133
A.1.2.3 Transitioning from Analysis to design – Gaia.....	134
A.1.3 MAS-CommonKADS.....	134
A.1.3.1 Analysis Phase - MAS-CommonKADS.....	135
A.1.3.2 design Phase - MAS-CommonKADS.....	136
A.1.3.3 Transitioning from Analysis to design – MAS-CommonKADS.....	137
A.2 Formal Methods.....	137
A.2.1 Transformational Programming.....	138
A.2.2 Formalisms for Multiagent Systems.....	140
A.3 Summary.....	141
APPENDIX B. FUNCTIONS USED IN THE TRANSFORMATIONS.....	142
B.1 The isAssigned Function.....	142
B.2 The usedInAction Function.....	142
B.3 The usedInTransition Function.....	143
B.4 The isNeeded Function.....	143

TABLE OF FIGURES

FIGURE 1 – MASE METHODOLOGY	3
FIGURE 2 – SEQUENCE DIAGRAM [5]	4
FIGURE 3 – A ROLE MODEL[8]	5
FIGURE 4 – CONCURRENT TASK DIAGRAM	6
FIGURE 5 – AGENT CLASS DIAGRAM	7
FIGURE 6 – COMMUNICATION CLASS DIAGRAM	8
FIGURE 7 – EXPANDED ROLE MODEL	15
FIGURE 8 – MODEL INFLUENCES.....	17
FIGURE 9 – EXAMPLE GRAPHICAL REPRESENTATION OF A TYPE	19
FIGURE 10 – CLASS DIAGRAM OF THE EXPANDED ROLE MODEL IN MASE	19
FIGURE 11 – ROLE TYPE	20
FIGURE 12 – TASK TYPE	20
FIGURE 13 – PROTOCOL TYPE.....	21
FIGURE 14 – CLASS DIAGRAM FOR THE TYPES USED IN THE DESIGN PHASE OF MASE	21
FIGURE 15 – AGENT TYPE.....	22
FIGURE 16 – COMPONENT TYPE.....	22
FIGURE 17 – CONVERSATION TYPE.....	23
FIGURE 18 – CONVERSATIONHALF TYPE.....	23
FIGURE 19 – STATE TABLE CLASS DIAGRAM	24
FIGURE 20 – STATE TABLE TYPE.....	25
FIGURE 21 – STATE TYPE.....	25
FIGURE 22 – TRANSITION TYPE.....	26
FIGURE 23 – TRANSITION WITH TWO SENDEVENTS TO THE SAME AGENT.....	27
FIGURE 24 – TWO ORDERINGS FOR RECEIVEEVENTS.....	28
FIGURE 25 – ACTION TYPE	29

FIGURE 26 – FUNCTIONCALL TYPE..... 30

FIGURE 27 – PARAMETER TYPE 30

FIGURE 28 – RECEIVEEVENT TYPE 30

FIGURE 29 – SENDEVENT TYPE 31

FIGURE 30 – EVENT TYPE 31

FIGURE 31 – THREE STAGES OF THE TRANSFORMATION PROCESS..... 33

FIGURE 32 – STAGE 1 IN THE TRANSFORMATION PROCESS 36

FIGURE 33 – SENDEVENT WITH MULTIPLE PROTOCOLS..... 37

FIGURE 34 – EXAMPLE OF TRANSFORMATION 1..... 38

FIGURE 35 – EXAMPLE OF TRANSFORMATION 2..... 39

FIGURE 36 – EXAMPLE OF TRANSFORMATION 3..... 39

FIGURE 37 – AMBIGUOUS PROTOCOLS FOR SENDEVENTS..... 40

FIGURE 38 – AMBIGUOUS PROTOCOLS FOR RECEIVEEVENTS 41

FIGURE 39 – ROLE MODEL EXAMPLE 41

FIGURE 40 – AGENT COMPONENTS CREATED FROM THE ROLES' TASKS 42

FIGURE 41 – AGENT DIAGRAM EXAMPLE 43

FIGURE 42 – STAGE 2 IN THE TRANSFORMATION PROCESS 46

FIGURE 43 – EXAMPLE OF SPLITTING A TRANSITION 48

FIGURE 44 – EXAMPLE 2 OF SPLITTING A TRANSITION 49

FIGURE 45 – TRANSITIONS WITH NO EVENTS..... 54

FIGURE 46 – PROTOCOLS DETERMINED FOR TWO TRANSITIONS..... 55

FIGURE 47 – PROTOCOLS DETERMINED FOR ALL TRANSITIONS..... 56

FIGURE 48 – EXAMPLE OF TRANSFORMATION 19..... 57

FIGURE 49 – EXAMPLE OF TRANSFORMATION 20..... 58

FIGURE 50 – EXAMPLE OF TRANSFORMATION 22..... 59

FIGURE 51 – EXAMPLE OF TRANSFORMATION 24..... 61

FIGURE 52 – EXAMPLE OF TRANSFORMATION 25..... 62

FIGURE 53 – EXAMPLE OF TRANSFORMATION 29..... 64

FIGURE 54 – EXAMPLE OF TRANSFORMATION 30..... 65

FIGURE 55 – TWO STATE DIAGRAMS ANNOTATED DIFFERENTLY 66

FIGURE 56 – TRANSITION WITH A RECEIVEEVENT AND MULTIPLE CONVERSATION NAMES..... 67

FIGURE 57 – STATE DIAGRAMS AFTER TRANSFORMATION 31 68

FIGURE 58 – DUPLICATE CONVERSATIONS BETWEEN AGENTS 69

FIGURE 59 – STATE DIAGRAMS WITH DIFFERENT ACTIONS IN STATE1 70

FIGURE 60 – EXAMPLE OF PROPAGATING THE SET OF CONVERSATIONS..... 71

FIGURE 61 – STAGE 3 IN THE TRANSFORMATION PROCESS 72

FIGURE 62 – CONVERSATION WITH MULTIPLE EXIT STATES 73

FIGURE 63 – STATE DIAGRAM AFTER TRANSFORMATIONS..... 73

FIGURE 64 – STATE DIAGRAM AFTER TRANSFORMATION 33..... 74

FIGURE 65 – STATE DIAGRAM AFTER TRANSFORMATION 34..... 75

FIGURE 66 – STATE DIAGRAM AFTER TRANSFORMATION 35..... 76

FIGURE 67 – STATE DIAGRAM BEFORE TRANSFORMATION 37..... 78

FIGURE 68 – STATE DIAGRAM AFTER TRANSFORMATION 37..... 78

FIGURE 69 – STATE DIAGRAM AFTER TRANSFORMATION 39..... 79

FIGURE 70 – STATE DIAGRAM AFTER TRANSFORMATION 43..... 81

FIGURE 71 – STATE DIAGRAM BEFORE TRANSFORMATION 44..... 83

FIGURE 72 – STATE DIAGRAM AFTER TRANSFORMATION 44..... 83

FIGURE 73 – STATE DIAGRAM BEFORE TRANSFORMATION 44..... 84

FIGURE 74 – STATE DIAGRAM AFTER TRANSFORMATION 44..... 84

FIGURE 75 – STATE DIAGRAM BEFORE TRANSFORMATION 45..... 85

FIGURE 76 – STATE DIAGRAM AFTER TRANSFORMATION 45..... 85

FIGURE 77 – STATE DIAGRAM BEFORE TRANSFORMATION 46..... 87

FIGURE 78 – STATE DIAGRAM AFTER TRANSFORMATION 46..... 87

FIGURE 79 – STATE DIAGRAM BEFORE TRANSFORMATION 46..... 87

FIGURE 80 – STATE DIAGRAM AFTER TRANSFORMATION 46..... 87

FIGURE 81 – THREE STAGES OF THE TRANSFORMATION PROCESS 93

FIGURE 82 – TRANSFORMATION MENU IN AGENTTOOL 94

FIGURE 83 – ROLE MODEL..... 96

FIGURE 84 – FULFILLSERCHREQUEST TASK FOR THE MANAGER ROLE..... 97

FIGURE 85 – BID TASK FOR THE BIDDER ROLE 98

FIGURE 86 – SEARCH TASK FOR THE SEARCHER ROLE 99

FIGURE 87 – INITIAL AGENT CLASS DIAGRAM..... 99

FIGURE 88 – AMBIGUOUS PROTOCOLS DIALOG 100

FIGURE 89 – FIRST PROTOCOL DECISION 101

FIGURE 90 – SECOND PROTOCOL DECISION 102

FIGURE 91 – DIALOG TO CHOOSE A PROTOCOL’S MODE 103

FIGURE 92 – MOBILESEARCHER AGENT’S BID COMPONENT 104

FIGURE 93 – MOBILESEARCHER AGENT’S SEARCH COMPONENT 104

FIGURE 94 – FIRST EVENT MATCH DECISION 106

FIGURE 95 – SECOND EVENT MATCH DECISION 106

FIGURE 96 – THIRD EVENT MATCH DECISION 107

FIGURE 97 – ANNOTATED FULFILLSERCHREQUESTS COMPONENT..... 108

FIGURE 98 – ANNOTATED BID COMPONENT 109

FIGURE 99 – AGENT CLASS DIAGRAM WITH CONVERSATIONS 110

FIGURE 100 – FULFILLSERCHREQUESTS COMPONENT AFTER STAGE THREE 110

FIGURE 101 – BID COMPONENT AFTER STAGE THREE..... 111

FIGURE 102 – INITIATOR HALF OF CONVERSATION13-1 112

FIGURE 103 – RESPONDER HALF OF CONVERSATION13-1 112

FIGURE 104 – PHASES IN THE MASE METHODOLOGY 123

FIGURE 105 – GOAL HIERARCHY DIAGRAM [5] 124

FIGURE 106 – SEQUENCE DIAGRAM [5] 125

FIGURE 107 – A ROLE MODEL [8]	126
FIGURE 108 – SAMPLE TASK IN MASE.....	126
FIGURE 109 – AGENT CLASS DIAGRAM	128
FIGURE 110 – COMMUNICATION CLASS DIAGRAM	129
FIGURE 111 – DEPLOYMENT DIAGRAM [8]	131
FIGURE 112 – ABSTRACT ANALYSIS HIERARCHY [15].....	132
FIGURE 113 – TYPICAL TRANSFORMATION SYSTEM [11].....	139

TABLE OF TABLES

TABLE 1 – RULES FOR DETERMINING A TRANSITION’S SET OF PROTOCOLS	51
---	----

ABSTRACT

Agent technology has received much attention in the last few years because of the advantages that multiagent systems have in complex, distributed environments. For multiagent systems to be effective, they must be reliable, robust, and secure. AFIT's Agent Research Group has developed a complete-lifecycle methodology, called Multiagent Systems Engineering (MaSE), for analyzing, designing, and developing heterogeneous multiagent systems. However, developing multiagent systems is a complicated process, and there is no guarantee that the resulting system meets the initial requirements and will operate reliably with the desired behavior.

The purpose of this research was to develop a semi-automated formal transformation system for the MaSE methodology, as one part of formal agent synthesis, that derives the system design based on the analysis. Since each transform in the transformation system preserves correctness, the designer can be sure that the resulting system design is correct with respect to the system specification. A secondary goal of this research was to develop a proof-of-concept module for agentTool that implements the transforms.

TRANSFORMING ANALYSIS MODELS INTO DESIGN

MODELS FOR THE MULTIAGENT SYSTEMS

ENGINEERING (MASE) METHODOLOGY

I. Introduction

A software engineer just received the requirements for a new computer system needed to support changing mission requirements in the midst of a hostile contingency. The requirements for the system include components working cooperatively in a distributed heterogeneous environment, adapting to changing conditions, and using various types of media to communicate. The warfighters must have the system by tomorrow morning for mission success. The software engineer takes the requirements, and through some interaction with the user develops a formal specification for a multiagent system, taking advantage of some pre-existing components in a stored knowledge base. After developing the system specification, the code for the system is automatically generated and a reliable and secure system is operationally deployed ahead of schedule.

This is just an example of what could be reality in the near future with the use of software tools that generate executable code automatically from a high-level graphical specification of the system. This type of next-generation technology could be the determining factor in whether or not our military can remain the most advanced and dominant military in the world throughout the next several decades. Documents such as Joint Vision 2010 [1] and Air Force 2025 [2] clearly detail the Air Force's need for distributed C³I applications to achieve information superiority in the 21st Century. If warfighters are going to trust computer systems in an increasingly complex information environment, then those systems must be reliable, robust, and secure. This thesis merges two enabling technologies, agent technology and formal

methods, that can be used together in order to develop reliable distributed systems that operate in complex and dynamically changing environments.

Agent technology has received much attention in the last few years because of advantages that agent systems have in complex, distributed environments. As agent technology has matured and become more accepted in the software industry, agent-oriented (AO) software engineering has become an important topic for software system developers who wish to develop practical and reliable agent-based systems. For agents to be useful in complex, distributed environments, they must work in cooperation with other agents, which is the domain of multiagent systems [3]. Engineering multiagent systems presents some unique challenges that are not found in Object-Oriented Software Engineering. Sycara [4] attempts to capture some of these challenges:

1. How to decompose problems and allocate tasks to individual agents.
2. How to coordinate agent control and communications
3. How to make multiple agents act in a coherent manner.
4. How to make individual agents reason about other agents and the state of coordination.
5. How to reconcile conflicting goals between coordinating agents.
6. How to engineer practical multiagent systems.

Methodologies for AO software engineering attempt to provide a solution to the sixth challenge and provide a framework for solving the first five. There are currently only a few AO software engineering methodologies for multiagent systems, and many of those are still under development. Additionally, most of the existing methodologies lack specific guidance on how to transform the specification of the system to the corresponding design.

The focus of this thesis is to mature an existing AO software engineering methodology developed at AFIT by applying formal methods to produce a transformation system that semi-automatically derives the system design from the analysis. A transformation can be thought of as a function, where a model or properties of a model are taken as input and the result is either a modified or an entirely new model. In order to accomplish this, the relationships between the different models and the points at which design

decisions are made must be identified. The result is a more completely defined and robust methodology that has precisely defined steps for designing a multiagent system based on the analysis specification.

1.1 Background

At AFIT, recent effort has focused around developing and maturing a methodology for developing multiagent systems, called Multiagent Systems Engineering (MaSE), that is intended to cover the complete life cycle of a multiagent system. A full description of MaSE can be found in Appendix A, as well as [3, 5-8]. This section presents a short overview of MaSE in order to provide the foundation of the problem being addressed in this thesis.

The MaSE methodology consists of the seven steps depicted in Figure 1. The boxes represent the different models used in the steps and the arrows indicate the flow of information between the models. While similar to the waterfall approach, MaSE is also intended to be applied iteratively. The first three steps represent the analysis phase of the methodology, while the last four steps represent the design phase.

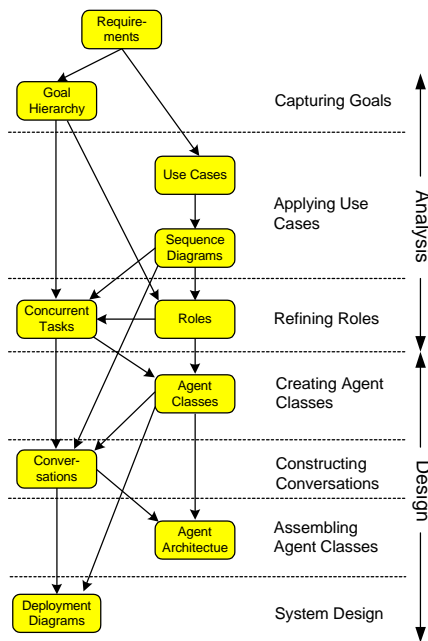


Figure 1 – MaSE Methodology

1.1.1 Capturing Goals

The first step in MaSE is *Capturing Goals*, where the system analyst takes the system requirements and develops a Goal Hierarchy Diagram, which is a structured set of system-level goals. Goals embody *what* the system is trying to achieve, and generally remain constant throughout the rest of the analysis and design process. After roles have been identified, the analyst will assign each role a set of goals. Intuitively, if all of the system requirements have been embodied as goals and all of the goals are being fulfilled by roles (which are later played by agents), the system should meet the initial requirements.

1.1.2 Applying Use Cases

Applying Use Cases is the next step in MaSE, where use cases are developed and then restructured as Sequence Diagrams. Uses Cases are defined from the system requirements and are a narrative description of a sequence of events that capture desired system behavior. Use Cases can be extracted from the requirements specification, user stories, or any other available source. Each Use Case should describe a particular instance of how the system will be used. Those sequences of interactions are then captured in the more structured representation of a Sequence Diagram. Sequence Diagrams, as shown in Figure 2, capture a sequence of messages between the different roles being played in the system. Sequence Diagrams provide a high-level view of how different roles interact to accomplish their goals, and are useful when constructing the tasks that each role has.

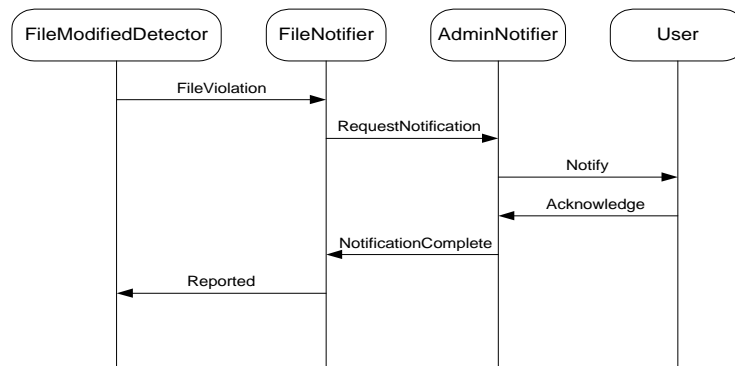


Figure 2 – Sequence Diagram [5]

1.1.3 Refining Roles

The next step is *Refining Roles*, where the analyst determines which roles will be played in the system and defines what tasks will be accomplished by each role. The Sequence Diagrams along with the Goal Hierarchy Diagram give the analyst insight into what roles should be played in the system. Each participant in the Sequence Diagrams is a candidate to become a role. Roles are defined much like an actor in a play, or a position in an organization (President, Vice President, Manager, etc). Each role must be responsible for accomplishing one or more goals in the Goal Hierarchy Diagram, and there must be at least one role responsible for each goal.

In *Refining Roles*, a Role Model is used to graphically depict the roles in the system and the communication paths between those roles. Role Models can also enable the reuse of roles from previous systems. The basic idea is that patterns of agent roles are constructed, labeled, and archived. When a new system is developed, the patterns are recognized and a Role Model can be re-applied from an archive, resulting in a collection of agent roles that satisfy a subset of the system goals. As shown in Figure 3, the arrows on Role Models are paths of communication connecting roles, and the dots indicate multiplicity.



Figure 3 – A Role Model[8]

As part of defining the roles, the analyst also defines the tasks that each role has. Tasks describe the behavior that a role must exhibit in order to accomplish its goal and are specified graphically using a finite state automaton as shown in Figure 4. A single role may have multiple concurrent tasks that define the complete behavior of the role. As a minimum, the messages in the sequence diagrams should also be messages being passed within a task. Concurrent tasks can be used to implement complex communication protocols such as Contract Net, Dutch Auction, etc. [9]. This is a very important part of the analysis as it allows the user to define how the system components will coordinate and interact with each other, which is the strength of multiagent systems. These tasks also lay the foundation for conversations between agent classes in the design phase of MaSE.

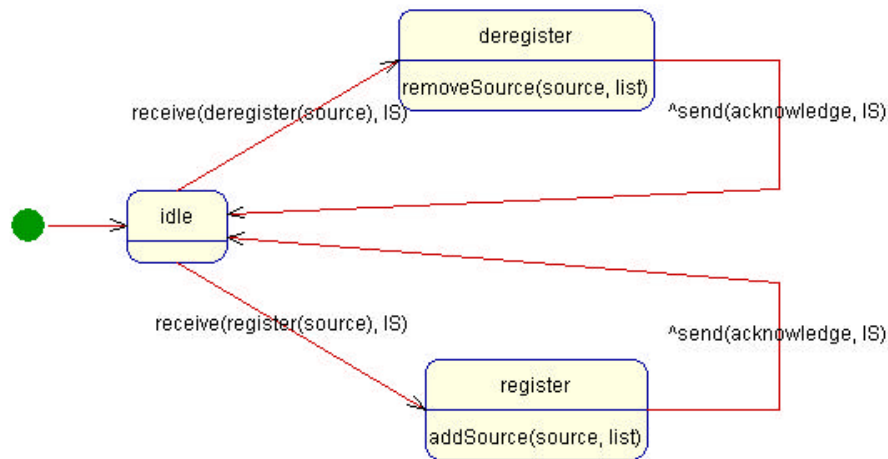


Figure 4 – Concurrent Task Diagram

One important property of a task is that they are able to communicate with multiple tasks in order to accomplish their goals. The tasks can belong to the same role, or they may belong to a different role. Tasks that belong to the same role can coordinate with each other through internal events. In order for a task to communicate to a task of another role, events that represent external communication are specified using *send* and *receive* events. These events are defined to send and retrieve messages from an implied message-handling component of the agent. In Figure 4, the *receive(register(source), IS)* event on the transition from the idle state to the register state is an example of a *receive* event, and the *send(acknowledge, IS)* event on the transition from the register state to the idle state is an example of a *send* event.

1.1.4 Creating Agent Classes

Creating Agent Classes is the first step in the MaSE design phase. Agent classes are defined from roles and an Agent Class Diagram is produced, as shown in Figure 5, that depicts the agent classes and the conversations between them. In order to ensure that all system goals are being met, each role must be played by at least one agent class. Thus the roles are the traceable link from the goals in the analysis phase to the agents in the design phase. In general, there is a one-to-one mapping from roles to agents, where each role becomes an agent class. There may be some instances however where the designer decides to

either combine multiple roles into an agent class, or allow a role to be played by more than one agent class. This is done either to share the capabilities and responsibilities of a role, or for performance enhancements by reducing communication overhead. In an Agent Class Diagram, each rectangle represents an agent class and a directed line represents a conversation between the agent classes. The arrows on the lines indicate the initiator and responder in the conversation.

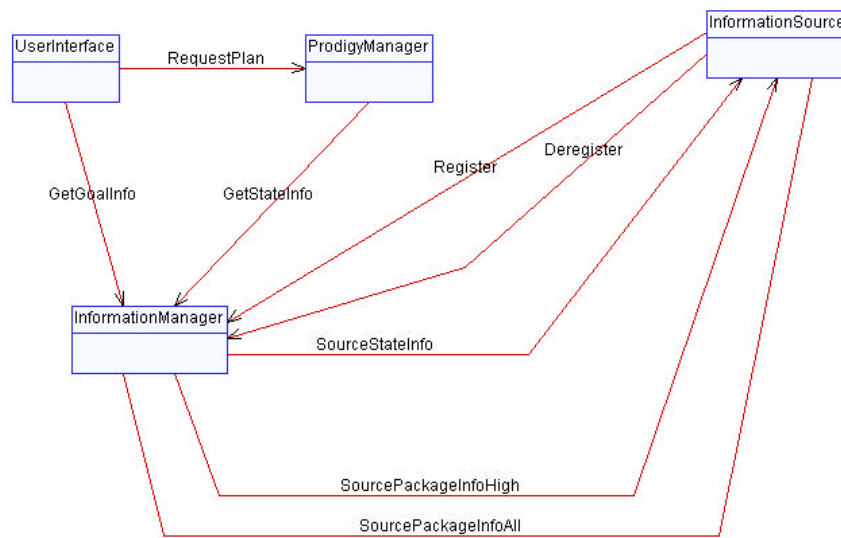


Figure 5 – Agent Class Diagram

1.1.5 Constructing Conversations

Constructing Conversations is the next step defined in MaSE, where the details of each conversation are defined from the tasks and sequence diagrams. Conversations are detailed coordination protocols between two agents and consist of two Communication Class Diagrams, one each for the initiator and responder. Conversations are at the heart of any multiagent system, as they detail how the different agents will communicate with each other. Like tasks, Communication Class Diagrams are described by finite state automaton that define the states and transitions for each half of a conversation. One example of a Communication Class Diagram is shown in Figure 6. Conversations are defined to be point-to-point communication between just two agents. Therefore, every event within the Communication Class Diagram is defined to be a message to or from the other agent instance participating in the conversation.

Conversations do not allow for communication with multiple agents simultaneously or for internal events to be exchanged with components internal to the agent.

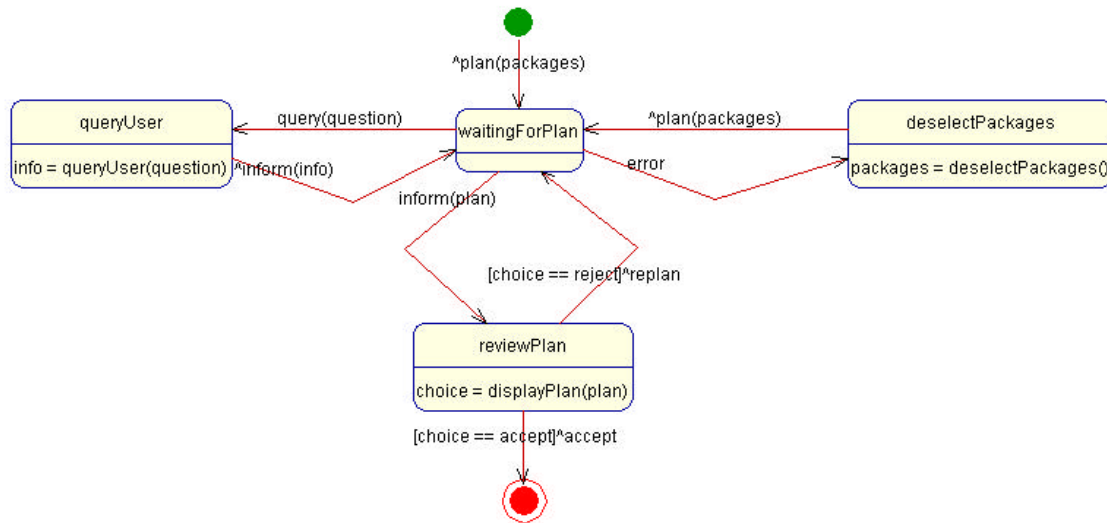


Figure 6 – Communication Class Diagram

1.1.6 Assembling Agent Classes

In *Assembling Agent Classes*, the internal components of an agent are defined. Robinson [10] details how to assemble agents from a set of standard or user-defined architectures. Each agent component is defined using an architectural modeling language combined with the Object Constraint Language. This allows the user to define attributes and functions that belong to the agent. Each component can also have a finite state automaton defining the dynamic characteristics of the component. The events passed within a component’s dynamic model are limited to internal events with other components that belong to that agent. In the design phase, external communication is defined strictly through conversations, so there are no external send or receive events with other agents in the component’s dynamic model.

1.1.7 System Deployment

The final step defined in MaSE is *System Deployment*. In this step, the designer develops a Deployment Diagram, which provides all of the detailed information necessary to deploy the system.

Deployment Diagrams define system parameters such as the actual number, types, and locations of the agents within the system. Three dimensional boxes represent agents, and lines connecting them represent conversations between those agents. A dashed-line box indicates that agents are housed on the same physical platform.

1.1.8 agentTool

In addition to the MaSE methodology, AFIT has developed a CASE tool named agentTool that serves as a validation platform and a proof of concept for MaSE. agentTool has a graphical user interface that allows a user to develop a multiagent system using the analysis and design models as defined in MaSE. agentTool is also able to generate Java code for a system based on the design models. Currently, the code generator is able to generate code for two different frameworks, agentMom and Carolina, but work is currently being done to integrate agentTool with the AFIT Wide Spectrum Object Modeling Environment that is looking at the more general code generation problem [11].

1.2 Problem

One main goal of AFIT's Agent Research Group has been to define a methodology specifically for formal agent system synthesis. To accomplish such a goal, the analysis models must be transformed into the design models, and then the design models must undergo another series of transformations that produce executable code. If each step in the transformation process preserves correctness, then the system engineer is guaranteed that the executable code is at least correct with respect to the analysis. A transformation system should also be able to provide traceability from the system requirements through the development process to the final executable code. In doing so, the system developer is able to verify that all of the system requirements have been fulfilled. Furthermore, if the system engineer is able to adequately decompose the problem and capture the system behavior in the analysis phase then there is hope that undesirable system behavior, to which multiagent systems are prone, can be avoided.

The problem being addressed by this research is the development of formal user-assisted transformations for transitioning from analysis models to design models within the MaSE methodology. Feasibility is demonstrated by developing and integrating the appropriate software components in AFIT's agentTool.

While the basic concepts of roles and tasks are defined in MaSE, exactly how a designer should transform them into agent classes, conversations, and internal agent components has not fully been explored. It is clear that roles are related to agent classes and the tasks that the roles perform are then related to the conversations between those agent classes. Similarly, tasks are also related to agent class components and the transformation process may be able to derive some high-level definitions of those components from the tasks. There is strong indication that much of the transformation process can be automated, with little user input. The main focus of this research is defining exactly what those transformations are and what is the most suitable way to implement them.

One difficulty in this transformation process revolves around the user being able to determine when coordination between two tasks is external communication and when it is internal to a role. In order to facilitate this transformation system and overcome this problem, this research will also focus on how a user should specify coordination protocols. A protocol defines a communication pattern designed to accomplish coordination between roles, or more specifically between tasks performed by those roles. Although protocols can be described through concurrent tasks [9], there may be another way to capture those protocols at a higher level of abstraction that would help determine the properties of the protocol and the tasks associated with it, which could be necessary information required for the transformation process.

1.3 Scope

This research effort will focus on defining the transformations that translate analysis models into design models for the MaSE methodology. Particular attention is given to defining protocols and concurrent tasks and their relation to conversations and agent components. Sufficiently complex examples

of multiagent systems that require open system protocols such as Contract Net, Dutch Auction, English Auction, etc. are used for demonstration purposes as well as several simple agent systems.

The following are assumptions concerning the transformations being presented in this thesis:

- The user has a good understanding of the MaSE methodology. This research will not address how to determine goals, transform goals to roles, which protocols should be used for a given system specification, which tasks need to be defined based on roles, or when to combine multiple roles into a single agent class.
- Transformations start with user-defined roles, tasks, and protocols so it is assumed that those models have been defined correctly. If there is deadlock within the tasks, then there will also be a deadlock situation in the resulting conversations.
- The transformations should use the current models and their semantics. The semantics of the models will only be changed when there is ambiguity or inconsistency in the current definition of the semantics.
- The transformations should not limit the design to a single platform or multiagent framework. For example, a developer should be able to deploy the resulting design using both agentMom and Carolina.
- The transformations should preserve correctness, but they do not need to create optimal solutions. If optimality is desired, then either another set of optimizing transformations could be applied, or the user could manually manipulate the modes for optimization.

1.4 Thesis Overview

The remainder of this document is organized as follows. Chapter 2 describes the approach taken for defining the relationships between the analysis and design model within MaSE, and presents the types that are used to formally define the transformations. Chapter 3 presents the actual transformations as a three stage process and describes each transformation using a predicate logic statement. Chapter 4 describes how the transformations were demonstrated by implementing the transformations as a component of agentTool. Chapter 5 discusses conclusions reached during this study and possible future research.

Appendix A has further background information that may assist the reader in understanding this thesis, and Appendix B presents functions used to define the transformations in Chapter III.

II. Problem Approach

A formal transformation system can be defined as a series of small steps that manipulate one model into an alternative representation. Each transformation must be deterministic in its execution and should not introduce inconsistencies between the two models. This chapter describes the approach taken to define a formal transformation system that takes analysis models and produces the corresponding design models within the context of the MaSE methodology. Specifically, this chapter explains the relationship between the models in the MaSE analysis and design phases, and presents an expanded Role Model in the analysis phase and a new organizational structure for agents and their components and conversations in the design phase.

Before formal transformations can be defined over the MaSE analysis and design models, each model that is involved in the transformations must be formally defined and the semantics of the models clarified. The models must have precise semantics so that the transformations can manipulate the models with predictable behavior. The details of the models presented in this chapter also include attributes defined specifically for the transformations in Chapter III, and have no meaning outside of the context of the transformations. Those attributes are not discussed in this chapter because they are not relevant to how the analysis models relate to the design models. They will be explained as they are introduced in Chapter III.

2.1 Expanding the Role Model

The first step in defining a transformation system is to determine which parts of the initial model map to which parts in the resulting model. The MaSE methodology makes it clear that the roles that an agent class plays, in conjunction with the communication paths between the roles' tasks, determine the conversations each agent class will have. However, further investigation proved this level of detail to be insufficient. When an agent class plays more than one role, it may be the case that some of the communication between the roles that was specified as external communication between the roles can now

be internal communication within the agent. Additionally, communication between tasks of the same role is not necessarily internal communication, but could in fact be external communication. The analysis models in MaSE do not specifically deal with role instances and multiplicity. That is something typically left to the design phase. However, the analyst may decide while developing the roles and their concurrent tasks that multiple instances of a single role will need to cooperatively work together in order to achieve some goal. In such a case, the communication being specified is external communication between the different role instances.

This deficiency led to an expanded view of the role model, as shown in Figure 7, that allows for a more detailed representation of the properties associated with roles and their tasks. As in the traditional Role Model, roles are depicted as rectangles. In the new Role Model, the goals that a role is responsible for are listed under the role. This allows the analyst to ensure that all of the goals from the Goal Hierarchy Diagram have been assigned to a role. Next, since roles may have one or more concurrent tasks associated with them, each task that a role has is denoted by an oval attached to the role. The lines between the tasks denote communication protocols that occur between the tasks. The protocols represent events that pass back and forth between the tasks, although which events are not specifically determined before the transformation process begins. The arrows indicate which task is the initiator and which task is the responder in the protocol, with the arrow pointing from initiator to responder. Solid lines indicate peer-to-peer communication, which is external communication either between two tasks of different roles, or between two tasks of different instances of the same role. External protocols involve messages being passed between roles and will become messages in a conversation between the agent classes that play those roles. Conversely, dashed lines denote communication between two tasks that belong to the same role instance. Roles are not allowed to share or duplicate a task. If the analyst finds that two roles should have the same task, then further role decomposition needs to take place. Shared tasks should be placed under a separate role, and those roles can then be combined back together into a single agent class in the design phase.

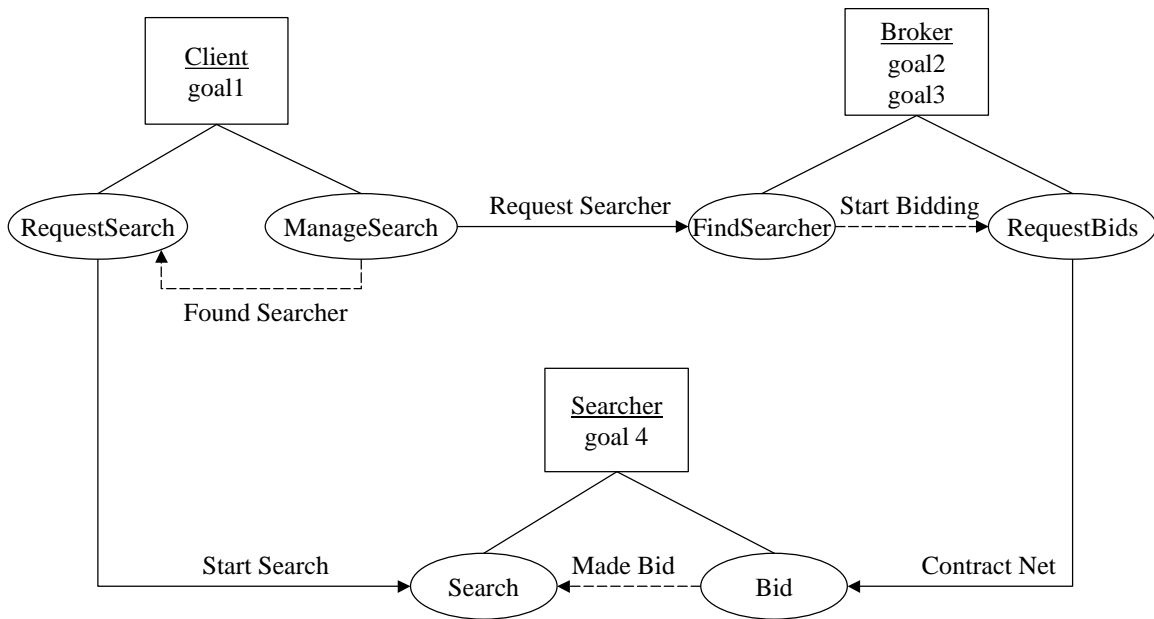


Figure 7 – Expanded Role Model

2.2 Transforming Concurrent Tasks to Conversations and Components

The next step in defining how to transform the analysis models into the design models was to determine the relationship between concurrent tasks and agent conversations and components. When examining how the Role Model mapped into the Agent Class Diagram, some interesting discoveries were made. First, when two roles are combined into a single agent class, the designer must determine whether the inter-role protocols should remain as external communication or if the communication that the protocol represents is now internal communication within the agent class. Since external protocols represent messages that will pass between the agents, they will become one or more conversations. The reason they may not be a single conversation is because the communication between the agents may not be continuous. There could be other coordinating communication that must take place internally, or with other agents. However, internal protocols (either initially defined or changed when roles are combined) will not result in conversations that represent that communication. Secondly, if more than one agent class plays a role, then for each external protocol that involves that role, conversations will be created for each agent class that plays that role. This means that there will be multiple instances of the same conversation between different agents in the system.

From experience using the MaSE methodology to develop several projects, concurrent tasks in the analysis phase do an excellent job in sufficiently capturing the coordination between the system roles. However, after transitioning to the design phase some of that coordination information was lost. Even if the roles mapped one-to-one into agents, when conversations were created from the tasks, there seemed to be nothing left that tied the conversations together to coordinate their execution. This was problematic when generating executable code from the design models. The programmer was left looking back to the concurrent task models in the analysis phase to figure out how the conversations should be coordinated.

The problem was that the finite state diagram that represented the task could include coordination with multiple tasks, both externally with many different roles as well as internally, while the conversations extracted from the tasks only dealt with the external communication between two agents at time. The parts that contained the coordination between the conversations were being discarded, and MaSE gave no guidance for recapturing the missing pieces. As a matter of fact, all internal events within the concurrent task diagrams were not being captured anywhere in the design phase. The approach taken to resolve this problem is that when a role is played by an agent class, a component is created in that agent's internal architecture for each of the role's tasks. The conversations can then be harvested from the component's state diagram and replaced with an action on a transition that represents the execution of the conversation. The component's own state diagram then retains the coordination that was previously missing, including passing internal events with other components of that agent class, as well as how the different conversations fit together.

This change led to a slightly different model of the relationship between an agent, its components, and its conversations. Since concurrent tasks are assumed to execute under their own thread of control and tasks now correspond to components, to maintain the analysis phase semantics the components must also execute under their own thread of control. Furthermore, if the conversations are harvested from components, then the conversations will logically belong to components, not directly to agents. Figure 8 illustrates how the models in the analysis phase translate to the models in the design phase as well as the relationship between the design models.

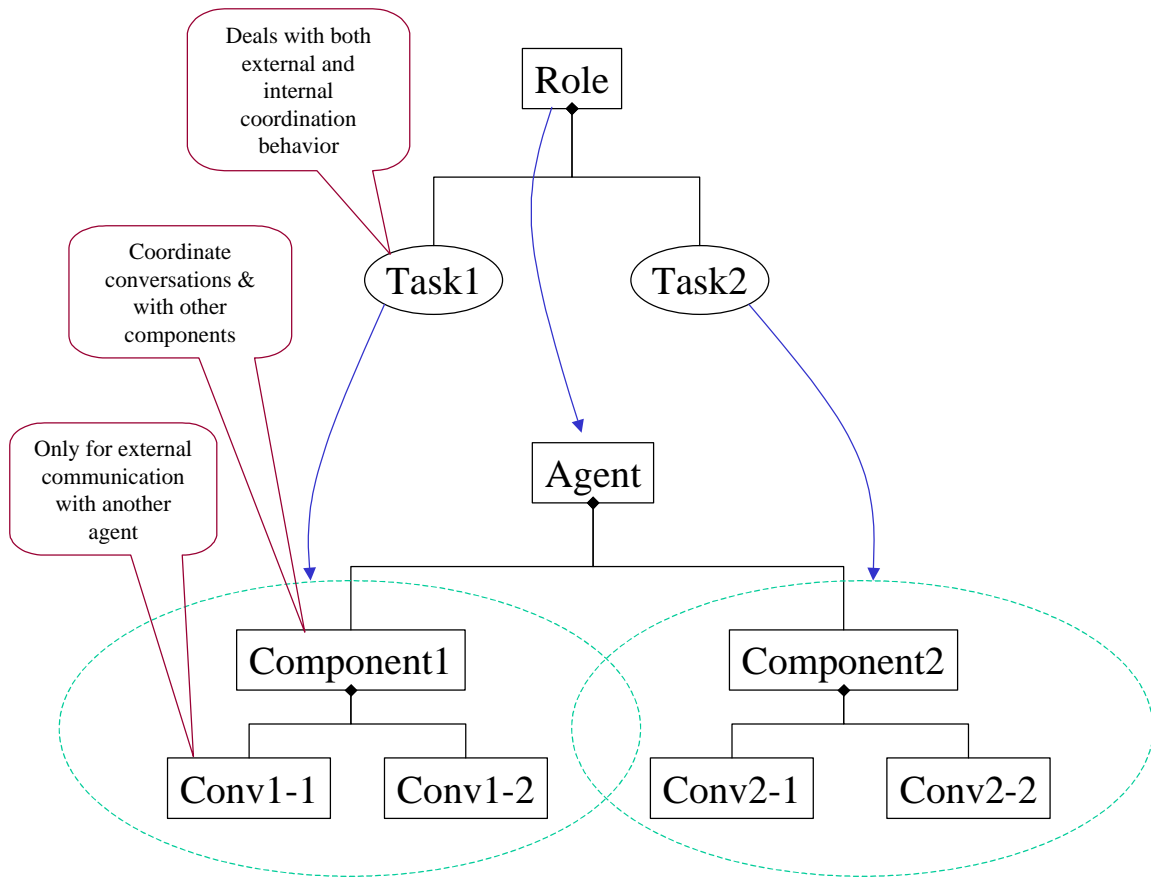


Figure 8 – Model Influences

This thesis does not propose that this is the only way to model the organizational structure of agents, components, and conversations in the design phase. Rather it is an attempt to capture all of the information that is present in the analysis models and retain the same basic idea of a conversation, which is independent of the multiagent framework in which it will be implemented. Some multiagent frameworks, such as Carolina [12], do not require that the conversations be broken out from the components. All of the external messaging could be captured adequately in the finite state automaton from the task due to the way in which messaging is accomplished within the Carolina framework. However, agentMom [13] is a multiagent framework that has a predefined class explicitly for implementing conversations. In agentMom, conversations operate under their own thread of control and a separate socket connection is established for the communication of each conversation. Therefore, the communication which conversations represent

(peer-to-peer) should be modeled independently from the internal events and messages that belong to other conversations.

One side-effect of this approach is that the conversations that are harvested from the tasks may be small pieces that fit together to form the overall communication that takes place between two agents. The reason that this communication will be broken up into multiple pieces is because there is other unrelated communication, either internal events or communication with another agent, that must take place in-between the different pieces. An alternative approach would be to capture all of the communication with another agent as a single conversation and allow the agent to somehow communicate with the conversation when other events that are unrelated to the conversation occur, such as internal events or communication with other agents. Doing so would alter the definition of a conversation within the context of MaSE so that this agent-to-conversation communication could take place. The approach that was chosen in this thesis seemed to be the most straightforward while still retaining the fundamental definitions of the models in the methodology.

2.3 Model Definitions

In order to define the models used in the transformations in Chapter III, each type in the models will be defined using an object format as demonstrated in Figure 9. Square brackets [and] denote that the attribute is a sequence of the type, while curly brackets { and } are similarly used to represent sets. In addition to defining the object types, graphical class diagrams using the Unified Modeling Language (UML) syntax are provided to give the reader a more complete picture of how the types in the models fit together. In general, a class in the class diagram is represented by a single type that will be used in the transformations. Aggregate components in the class diagrams become an attribute for the type that represents the parent class in the aggregate relationship.

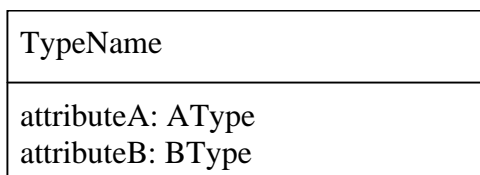


Figure 9 – Example Graphical Representation of a Type

2.3.1 Analysis Models

The Role Model and The Concurrent Task Models are the only models in the analysis phase of MaSE used in the transformation process. The UML class diagram in Figure 10 shows the classes used to defined the Role Model and Concurrent Task Models. The type StateTable, which is a component of the type Task is not defined in this section. Since the Tasks, Components, and Conversations all use a StateTable to represent their dynamic properties, the StateTable is discussed at length in Section 2.3.3.

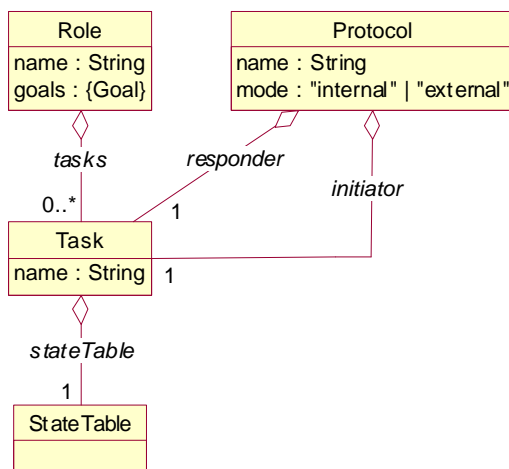


Figure 10 – Class Diagram of the Expanded Role Model in MaSE

The first model of interest in the analysis phase of MaSE is the Role Model. Role Models describe the roles in the system, the tasks they have, and the protocols that capture the communication paths between the tasks. A Role (Figure 11) is defined by its name, the set of goals it is responsible for, and a set of tasks that define how the role will accomplish its goals. Each role in the system has a name that uniquely identifies it from any other role in the system.

Role
name: String goals: {Goal} tasks: {Task}

Figure 11 – Role Type

A Task (Figure 12) is defined by its name and a state table (equivalent to a state diagram) that is used to describe the behavior of that task. Tasks must also have a name which uniquely identifies it from other tasks within the system. As previously stated, a task can not be duplicated within the analysis phase. If a the analyst feels like a task will need to be shared by more than one agent class later in the design phase, then a separate role should be created to perform the task. That role and its tasks can then be played by multiple agent classes.

Task
name: String stateTable: StateTable

Figure 12 – Task Type

A protocol (Figure 13) is defined by the name of that protocol, the initiator and responder tasks, and the mode, that specifies whether the protocol is internal or external communication. Multiple protocols in the Role Model may have the same name. A protocol simply represents a sequence of events being passed between entities, roles in the analysis phase and then the agents that play those roles in the design phase. Since the communication patterns are captured with the state diagrams in the tasks, the protocols more precisely capture the events being passed between the tasks of the roles, and likewise between the agent components and conversations. The attributes *initComp* and *respComp* point to the agent components created that implement the tasks from the Role Model.

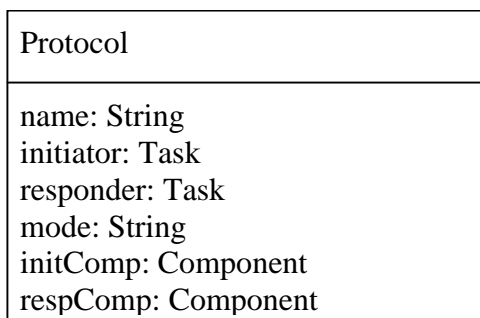


Figure 13 – Protocol Type

2.3.2 Design Models

This section defines the types that make up the design models of MaSE, which include the Agent Class Diagram, the Component State Diagrams, and the Communication Class Diagrams. The UML class diagram in Figure 14 shows the types used to define these models. Again, since the StateTable type is also used in the tasks defined in the analysis phase, Section 2.3.3 is devoted to their explanation.

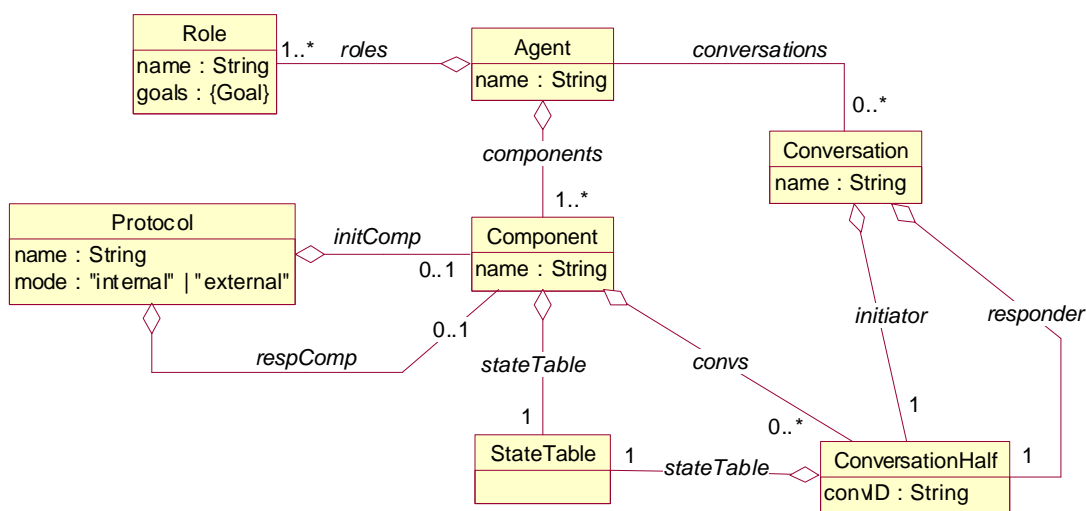


Figure 14 – Class Diagram for the Types Used in the Design Phase of MaSE

The first model in the design phase of MaSE is the Agent Class Diagram. The Agent Class Diagram simply depicts the agent classes in the system, the roles those agents play, and the conversations between the agents. Based on the discussion in Section 2.2, the way the pieces of the Agent Class Diagram fit together is a bit more complicated.

2.3.2.1 Agents

An agent type represents the agents defined in the Agent Class Diagram. An agent type (Figure 15) is defined by its name, the roles it plays, the components it has, and the conversations it participates in. Each agent type has a name that uniquely identifies it from any other agent in they system.

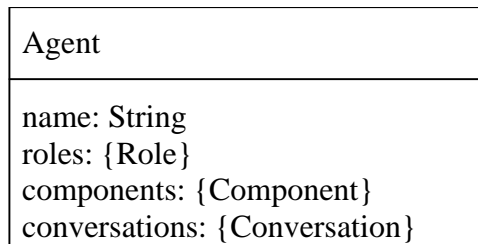


Figure 15 – Agent Type

2.3.2.2 Components

Component types (Figure 16) are defined by their name, a state table, and a set of conversation halves. If a component is created from a task during the transformation process, its name comes from the task that it was created to implement. Therefore, while there may be multiple agent classes that have components with the same name, no agent class will have two components that are named the same. A component's state table will initially be the same as the state table of the task it was created from, but after the transformation process it will only contain internal events and actions that the component must perform. The *convs* attribute is the set of ConversationHalves that are extracted from the state table of the component.

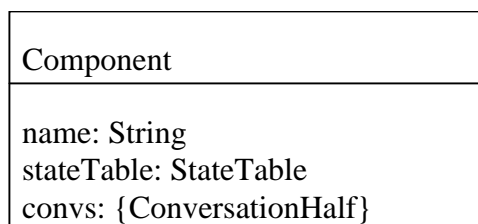


Figure 16 – Component Type

2.3.2.3 Conversations

Conversations (Figure 17) are made up of two ConversationHalves, one that is the initiator and one that is the responder. Each Conversation also has a name that uniquely identifies it within the system. The ConversationHalf type (Figure 18) corresponds to the Communication Class Diagrams within MaSE, and is composed of a state table and a *convID* that is the name of the Conversation it belongs to. The state table of a ConversationHalf details the external communication and internal actions that defines the behavior of one agent within a Conversation.

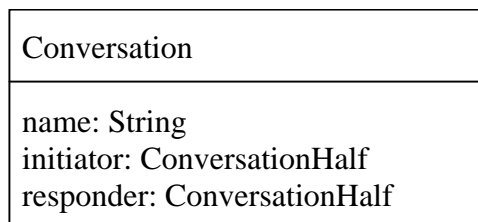


Figure 17 – Conversation Type

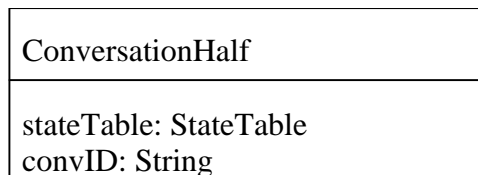


Figure 18 – ConversationHalf Type

2.3.3 State Tables

Several key models within MaSE (Concurrent Task Diagram, Communication Class Diagram, and the dynamic model for Components) are defined using a finite state diagram, or equivalently a state table. Each of these models are also key components to the transformation system defined in Chapter III. The state tables in the different models have various restrictions and slightly different semantics. This section defines the state table types and explains the differences between the models. The UML class diagram in Figure 19 gives a graphical overview of the different types used to define a StateTable and shows the different relationships between them.

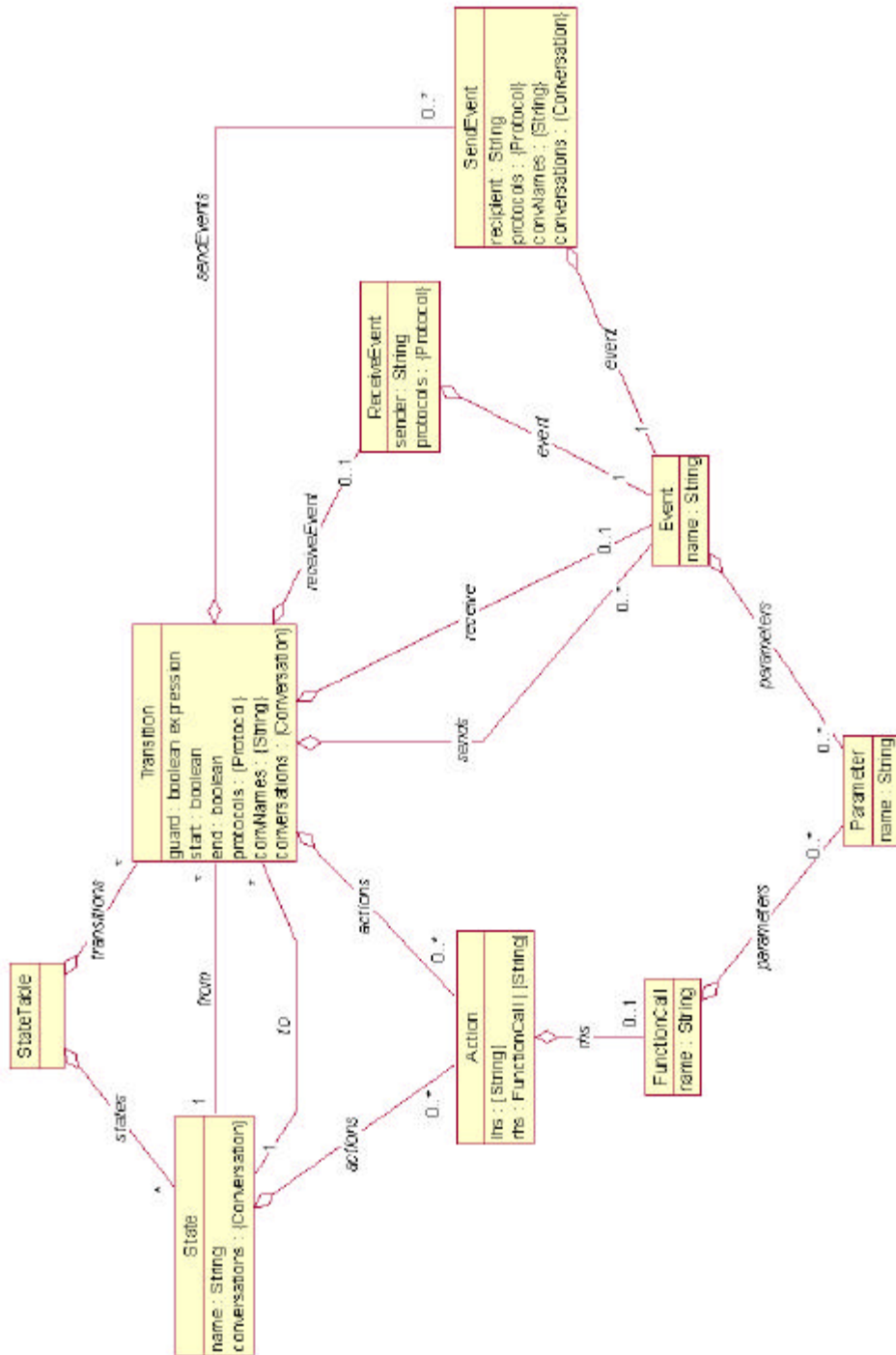


Figure 19 – StateTable Class Diagram

A StateTable (Figure 20) is used to define the behavior of an entity through a set of states that it may be in at any point in time and the set of transitions that occur as the entity goes from one state to the next. State tables also describe the communication patterns that take place between the different entities in the system through the events that are sent and received on the transitions.

StateTable
states: {State} transitions: {Transition}

Figure 20 – StateTable Type

2.3.3.1 States

A State (Figure 21) represents internal processing and is defined by a name and a sequence of actions that take place within the state. Each state within a state table must have a unique name. Upon entering a state, the sequence of actions will be executed in the given order. The *conversations* attribute is only used during the transformations defined in Chapter III and holds the set of conversations that the state will be in.

State
name: String actions: [Action] conversations: {Conversation}

Figure 21 – State Type

The beginning state of every state table is the *start* state. In a state diagram this is represented by a solid circle, and in a state table it simply has the name “start”. Every state table will continue execution until reaching the *end* state. In a state diagram the *end* state is represented by solid circle inside a hallow circle, and in a state table it is the state named “end”.

2.3.3.2 Transitions

Transitions specify how an entity moves from one state to another and define communication that takes place within the system. A transition is typically defined by its origin and destination states, the received event that triggers the transition, a guard condition, a set of actions that take place (if allowed), and a set of transmission events. In a state diagram, the syntax for the transition label would be:

```
trigger [guard] / action(s) ^ transmission(s)
```

For this thesis, a Transition type (Figure 22) is defined that is used for every model that has a state table, and is therefore usable throughout the transformation process in Chapter III. The differences in the semantics for transitions are discussed for each model. Several of the attributes shown for a transition (*start*, *end*, *conversations*, *convNames*, and *protocols*) are used only for the transitions in Chapter III.

Transition
from: State receive: Event receiveEvent: ReceiveEvent guard: Boolean Expression to: State actions: [Action] sends: {Event} sendEvents: {SendEvent} start: Boolean end: Boolean conversations: {Conversation} convNames: {String} protocols: {Protocol}

Figure 22 – Transition Type

Transitions occur instantaneously and move the entity from one state to another (or possibly back to the same state). A transition is enabled if all of the following conditions are true.

1. The transition's *from* state is the current state.
2. The transition's trigger event (if it has one) has been generated.

3. The transition's guard condition (if it has one) evaluates to true.
4. All actions in the transition's *from* state have been completed

If a transition does not have a trigger or a guard, both conditions are assumed to hold and the transition is enabled. If there is no trigger, but there is a guard that is true, then the transition will also be enabled.

2.3.3.2.1 Concurrent Task Diagram

Concurrent Task Diagrams allow the user to define both internal and external events that take place between the tasks. Therefore, the trigger for a transition can either be an event that is received internally (the *receive* attribute) or externally (the *receiveEvent* attribute). A transition cannot have both. A transition can also have send events, both internal and external. Once the transition is triggered, all transmission events are sent. The *sends* attribute denotes the internal events that are sent and the *sendEvents* attribute denotes the external events that are sent. All transmissions are assumed to take place instantaneously, so there is no implied ordering within the transmissions. Therefore, multiple transmissions to a single task (i.e. that belong to the same protocol) are not allowed on the same transition. As the state tables are designed and implemented, an ordering is necessarily applied to the transmissions because in reality they cannot take place instantaneously. However, the order the transmissions are received is already defined in the analysis phase, because only one message can be received on a transition. In order to receive two events, two transitions are required, and transitions are always enabled in a specific order.

To illustrate why a transition cannot have two transmissions to the same task, first consider Figure 23. The transition has two SendEvents that we assume are being sent to the same task. Figure 24 shows the corresponding state diagram with two different orderings for the received events. This situation is not allowed to avoid choosing the wrong order when the state diagram in Figure 23 is implemented.

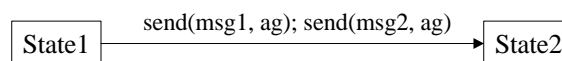


Figure 23 – Transition with Two SendEvents to the Same Agent

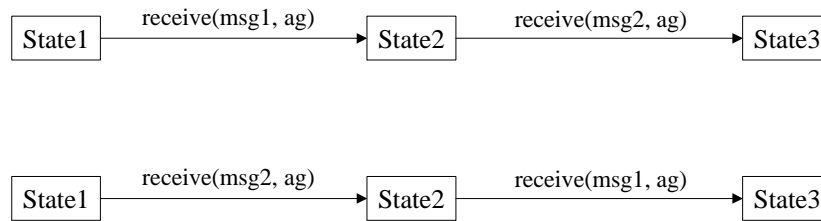


Figure 24 – Two Orderings for ReceiveEvents

Concurrent Task Diagrams also have special restrictions on where actions are allowed. All actions are defined to take place within the states, so every transition’s *actions* attribute will be the empty sequence.

2.3.3.2.2 Component State Table

Each component for an agent has a state table that defines its behavior. Any events on the transitions in the state table are defined to be internal events to other components of the same agent instance. Therefore, the *receiveEvent* and *sendEvents* attributes will not be used. During the transformation process in Chapter III, a Component is created for each task and starts with an identical StateTable that may have transitions with non-null *receiveEvent* or *sendEvents* attributes. However, the transformation process removes those external events and create conversations with them, so that by the end of the transformation process, component state tables have only internal events defined by the *receive* and *sends* attributes.

Some of the transformations in Chapter III also add actions to the transitions. The semantics of actions on a transition is that once the transition is enabled, the actions are executed in the given order *before* any events are sent.

2.3.3.2.3 Communication Class Diagram

Communication Class Diagrams define the communication that takes place within a conversation between two agents. Therefore, all events on the transitions represent external messages to or from the other agent participating in the conversation. However, these messages are represented using the *receive*

(incoming message) and *sends* (outgoing messages) attributes, not *receiveEvent* or *sendEvents* like in the Concurrent Task Diagram. Therefore, the transformation system must take the external events defined in the Concurrent Task Diagrams with the *receiveEvent* and *SendEvent* attributes and transform them into *receive* and *sends* that represent the same communication in the Communication Class Diagrams. Communication Class Diagrams also allow for actions on the transitions, that are defined to take place before any outgoing messages are sent.

2.3.3.3 Actions and Events

Actions represent the actual processing that takes place in the state table. Actions can be used to represent internal reasoning, reading a percept from sensors, or causing an effector to make a change in the environment. Originally, actions (or activities) were defined purely in the form of functions, where each function would have a number of input parameters and could return one result, either as a single value or as a tuple [5]. The syntax of an action was of the form:

```
result = action-name(param1, param2, ... paramN)
```

The definition of an action has since been expanded to allow tuple-to-tuple assignments, such as:

```
<x,y> = position(object) and <a,b> = <x,y>
```

The Action type is shown in Figure 25. The *lhs* attribute of an action represents the left-hand-side of an assignment and is a sequence of strings that can be used to represent either a single value or a tuple. The *rhs* attribute is the right-hand-side of the assignment and is either a FunctionCall or another sequence of strings.

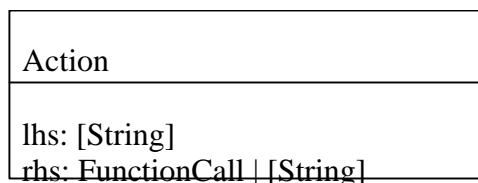


Figure 25 – Action Type

FunctionCalls (Figure 26) represent processing being done by a role or agent within the action. FunctionCalls are defined by their name and a sequence of input parameters. Parameters (Figure 27) are simply defined by a string that represents the parameter's name. Parameters would generally have a type and a value associated with the identifying name. These are not necessary for the transformations in this thesis, however they would be required for future transforms that translate the design into code or another formal language syntax. Similarly, the FunctionCall type would also reference a Function type (not defined here) that has pre- and post-conditions that define its behavior.

FunctionCall
name: String parameters: [Parameter]

Figure 26 – FunctionCall Type

Parameter
name: String

Figure 27 – Parameter Type

Since Concurrent Task Diagrams distinguish internal events from external events, a different type is defined for each. The ReceiveEvent type (Figure 28) was defined to represent external events that are received to trigger a transition in a Concurrent Task Diagram. Each ReceiveEvent represents an event on a transition of the form `receive(event, sender)`. The *event* attribute represents the external message that is being received and the *sender* attribute represents the role instance that sent the message. The *protocols* and *convName* attributes are only used in the transformations in Chapter III.

ReceiveEvent
event: Event sender: String protocols: {Protocol} convName: String

Figure 28 – ReceiveEvent Type

Just as the ReceiveEvent type was defined to represent an external event received in a Concurrent Task Diagram, a SendEvent type (Figure 29) was defined to represent an external event that is sent. Each SendEvent represents an event on a transition of the form `send(event, recipient)`. Like a ReceiveEvent, a SendEvent also has an *event* attribute that represents the message being sent, but has a *recipient* attribute that defines who the message is being sent to. If the *recipient* attribute is of the form “<list-name>”, then the recipient is a list of agents the message the will be sent to, and the SendEvent represents a multicast. The *protocols*, *conversations*, and *convName* attributes were added for the transformations in Chapter III.

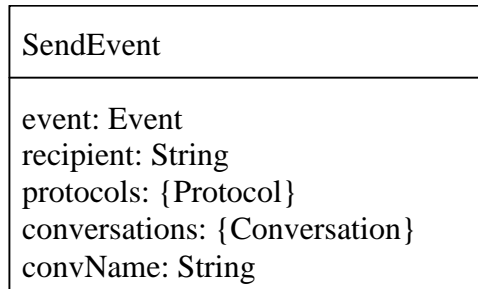


Figure 29 – SendEvent Type

An Event (Figure 30) is used to define a message that is passed in the system, either internally or externally. The *name* attribute represents the performative, which is the intent of the message, and the sequence of parameters represents the content of the message.

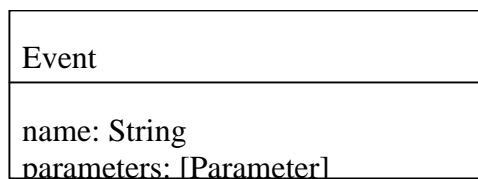


Figure 30 – Event Type

2.4 Summary

This chapter described the approach taken for developing a formal transformation system that semi-automatically creates MaSE design models based on the analysis models. This involved determining the relationships between the models and where input is required from the designer. An expanded Role

Model for the analysis phase was presented, and the new organizational structure for the agents, components, and conversations in the design phase was described. This chapter also presented each model used in the transformations by defining the individual types and their attributes, as well as the semantics and constraints for the models. Chapter III follows the approach laid out in this chapter and presents the detailed transformations as a three-stage process.

III. Transformations

Having defined the analysis and design models for MaSE in Chapter II, this chapter now develops the specific transformations that will use the Role Model and the Concurrent Task Diagrams to generate the Agent Class Diagram, the Communication Class Diagrams for the conversations between the agent classes, and the agent components that constitute the agents' internal architectures. The transformation system presented is actually a series of small steps that incrementally change the roles and tasks from the analysis phase into agent classes and their components and conversations in the design phase. The process can be broken down into the three stages shown in Figure 31. The transformations are designed to be applied in the order they are presented, although some of them are to be applied iteratively.

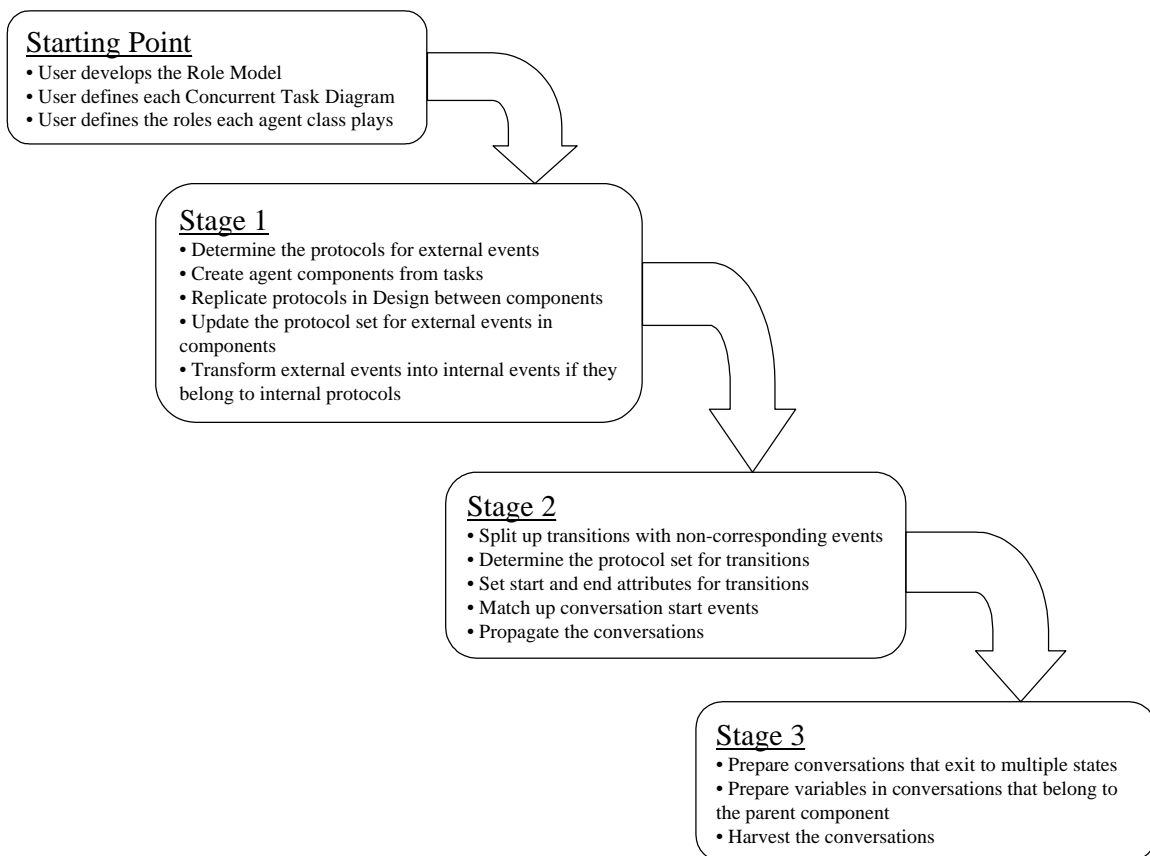


Figure 31 – Three Stages of the Transformation Process

Within this chapter, Section 3.1 defines the notations used to present the transformations. Section 3.2 describes the first stage of the transformation process, where the components for the agent classes are defined based on the user's decision about which agent classes will play which roles. Section 3.3 describes the second stage, centered around annotating the component state diagrams and matching external events in the different components that become the initial messages of the conversations. Section 3.4 provides details for the last stage of the transformation process, where component state diagrams are prepared for the removal of the states and transitions that belong to conversations. They are then removed and added to the state diagrams of the corresponding conversation halves.

3.1 Formal Notations

In order to formally define the transformations presented in this chapter, this section presents the notations used. Each transformation is defined by a predicate logic equation of the form: $\text{condition} \Rightarrow \text{result}$, where the condition is the set of requirements that must be true for the transformation to take place, and the result describes what is guaranteed to be true after the transformation is performed. This notation is similar to defining functions with pre-conditions and post-conditions. These transformations describe *what* must take place, not *how* it must be done. The types used in the transformations are the types described in Chapter II, and the following describes how they are used in this chapter:

- The universe of discourse is the models in the system currently being developed
- Sets are indicated by the pair of symbols { and }, and items in the set are separated by , when explicitly delineated
- The Union of two sets is indicated by the symbol \cup
- The Intersection of two sets is indicated by the symbol \cap
- The subset relationship is indicated by the symbol \subseteq
- Sequences are indicated by the pair of symbols [and], and items in the sequence are separated by , when explicitly delineated and are assumed to appear in the order required by the sequence

- Sequence concatenation is indicated by the symbol ζ
- The symbol $\#$ is used to indicate the cardinality of a set or sequence
- An element of a set is indicated by the symbol \hat{I}
- The sub-field of a type is indicated by the dot notation, such as `type.attribute`
- String concatenation is indicated by the symbol $+$
- The tick symbol $'$ indicates that the variable being referenced is the variable after the transformation

3.2 Generating the Agent Model

This section discusses the first stage of transformation process from the analysis phase to the design, highlighted in Figure 32. Before these transformations can begin, the designer must have developed the Role Model and the Concurrent Task Diagrams in the analysis phase. Additionally, the designer must define the initial set of agent classes, but only to the extent of deciding which set of roles each agent class will play, ensuring that each role is played by at least one agent class. In this stage, the transformations must first determine to which set of protocols each external event belongs. Then, components are created for agent classes to represent the tasks that the agent's roles must perform. Since the internal architecture of the agent consists of its components and their relationships, this step essentially derives the architecture of each agent based on the analysis models. Whenever roles with an external protocol between their tasks are combined, the user may determine that that protocol is now internal communication within the agent. When this happens, every external event that belongs to that protocol must be transformed into an internal event.

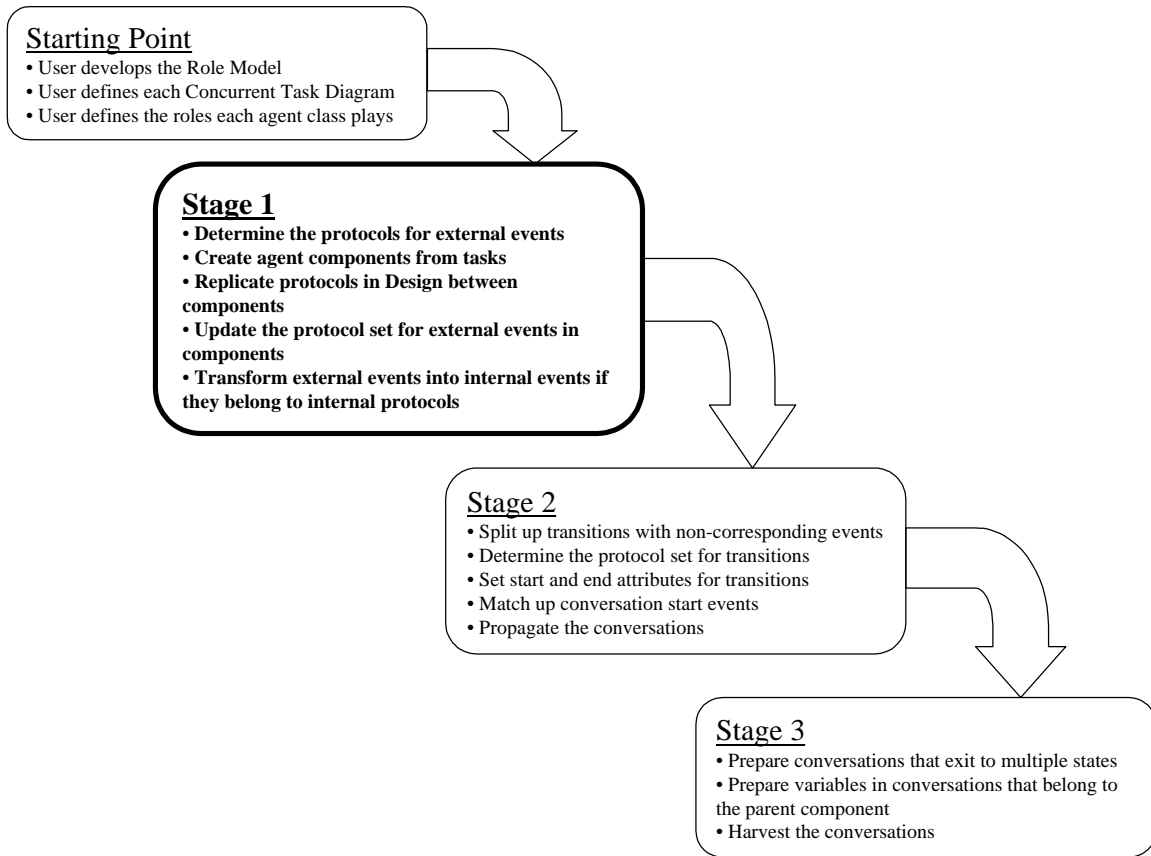


Figure 32 – Stage 1 in the Transformation Process

3.2.1 Determining Protocols for External Events

In order to transform external events into internal events by declaring the protocol as internal, the protocols that each external event belongs to must first be determined. The protocols that events belong to are also the primary factor in the second stage of the transformation process when transitions are labeled to denote the start and end of conversations. Events, or the messages they represent, may belong to multiple protocols. This may seem a little confusing at first, but the concept is simple. Figure 33 shows an example of how a SendEvent could belong to multiple protocols. The sets above each event are shown for purposes of the example and represent the set of protocols for that event. Each of the transitions into State3 have ReceiveEvents that belong to different protocols. The transition leaving State3 has a SendEvent that is a message sent in both protocols. This scenario seems logical and states that it doesn't matter which protocol

is currently being carried out when State3 is reached. The variable y will be computed and then sent, regardless of the current protocol.

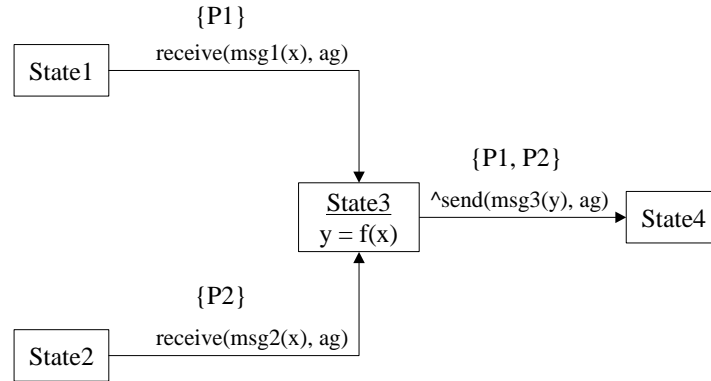


Figure 33 – SendEvent with Multiple Protocols

The next three transformations are applied to the Role Model in the analysis phase and automatically determine for some cases if external events belong to a specific protocol. However, there are cases where it is impossible to automatically determine if the analyst meant for an event to belong to a protocol or not. In these ambiguous cases, the analyst will need to make that determination. Transformation 1 covers the case shown in Figure 34 that illustrates the condition that there are two tasks that have at least one set of corresponding events¹ and those tasks have a protocol between them. Additionally, neither task has a protocol with another task that also has a corresponding event in its state table (denoted by an arrow with an X over it). If these conditions hold, then the events must be part of the protocol, since there are no other protocols to which the events could belong. Figure 34 also shows that the events within the tasks do not need to be unique. If Task1 has more than one correspond event, they will all be labeled as belonging to Protocol 1.

¹ A “corresponding events” refer to a SendEvent and a ReceiveEvent that have the same event (or message) parameter, e.g., $send(do(x), ag)$ and $receive(do(x), agent)$. The $do(x)$ parts represent the message being passed. The events only need to have the same number of parameters (with matching types). The names of the parameters do not need to match, nor do the identifiers (the recipient in the SendEvent and the sender in the ReceiveEvent).

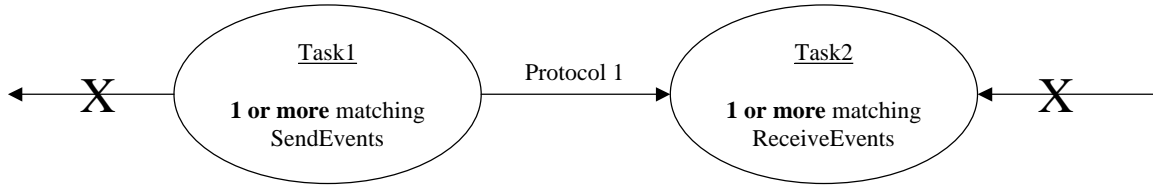


Figure 34 – Example of Transformation 1

Transformation 1

$\forall t, t2 : \mathbf{Task}, st, st2 : \mathbf{StateTable}, trans, trans2 : \mathbf{Transition}, se : \mathbf{SendEvent}, re : \mathbf{ReceiveEvent},$
 $p : \mathbf{Protocol} \bullet$
 $((p.initiator = t \wedge p.responder = t2) \vee (p.initiator = t2 \wedge p.responder = t)) \wedge st = t.stateTable$
 $\wedge st2 = t2.stateTable \wedge trans \in st.transitions \wedge trans2 \in st2.transitions \wedge se \in trans.sendEvents$
 $\wedge re = trans2.receiveEvent \wedge se.event = re.event$
 $\wedge \neg(\exists t3 : \mathbf{Task}, st3 : \mathbf{StateTable}, trans3 : \mathbf{Transition}, re2 : \mathbf{ReceiveEvent}, p2 : \mathbf{Protocol} \bullet$
 $p2 \neq p \wedge t3 \neq t \wedge st3 = t3.stateTable \wedge trans3 \in st3.transitions \wedge re2 = trans3.receiveEvent$
 $\wedge ((p2.initiator = t \wedge p2.responder = t3) \vee (p2.initiator = t3 \wedge p2.responder = t)) \wedge se.event = re2.event)$
 $\wedge \neg(\exists t4 : \mathbf{Task}, st4 : \mathbf{StateTable}, trans4 : \mathbf{Transition}, se2 : \mathbf{SendEvent}, p3 : \mathbf{Protocol} \bullet$
 $p3 \neq p \wedge t4 \neq t \wedge st4 = t4.stateTable \wedge trans4 \in st4.transitions \wedge se2 \in trans4.sendEvents$
 $\wedge ((p3.initiator = t2 \wedge p3.responder = t4) \vee (p3.initiator = t4 \wedge p3.responder = t2)) \wedge se2.event = re.event)$
 \Rightarrow
 $(re'.protocols = \{p\} \wedge se'.protocols = \{p\})$

Transformation 2 covers the case illustrated in Figure 35, where there are two tasks that have at least one set of corresponding events and those tasks have a protocol between them. This is no different than Transformation 1, except that now the ReceiveEvent must be unique within Task2, and it is acceptable for Task2 to have a protocol with another task that also has a corresponding SendEvent. The reason this is still correct is that there is no other protocol to which the SendEvent(s) in Task1 can belong. Furthermore, since the ReceiveEvent in Task2 is the only matching event for the SendEvent(s) in Task1, it must also belong to that protocol. However, the ReceiveEvent in Task2 is not limited to the protocol with Task1. If Task2 has another protocol with a different task that has a corresponding SendEvent, then the ReceiveEvent could also belong to that protocol.

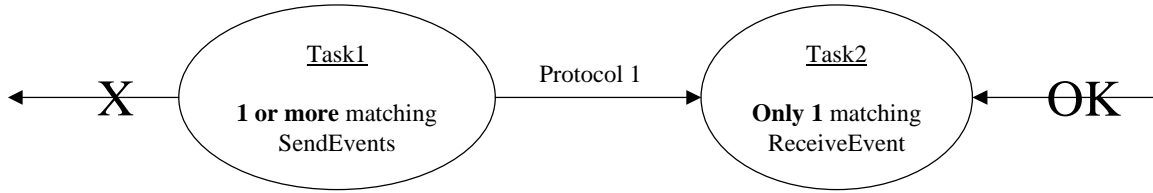


Figure 35 – Example of Transformation 2

Transformation 2

$\forall t, t2, t3 : \mathbf{Task}, st, st2, st3 : \mathbf{StateTable}, trans, trans2 : \mathbf{Transition}, se : \mathbf{SendEvent}, re : \mathbf{ReceiveEvent},$
 $p : \mathbf{Protocol} \bullet$
 $((p.initiator = t \wedge p.responder = t2) \vee (p.initiator = t2 \wedge p.responder = t)) \wedge st = t.stateTable$
 $\wedge st2 = t2.stateTable \wedge trans \in st.transitions \wedge trans2 \in st2.transitions \wedge se \in trans.sendEvents$
 $\wedge re = trans2.receiveEvent \wedge se.event = re.event$
 $\wedge \neg(\exists trans3 : \mathbf{Transition}, se2 : \mathbf{SendEvent} \bullet trans3 \in st.transitions \wedge se2 \in trans3.sendEvents$
 $\wedge se2 \neq se \wedge se2.event = se.event)$
 $\wedge \neg(\exists t4 : \mathbf{Task}, st4 : \mathbf{StateTable}, trans4 : \mathbf{Transition}, se3 : \mathbf{SendEvent}, p3 : \mathbf{Protocol} \bullet$
 $p3 \neq p \wedge t4 \neq t \wedge st4 = t4.stateTable \wedge trans4 \in st4.transitions \wedge se3 \in trans4.sendEvents$
 $\wedge ((p3.initiator = t3 \wedge p3.responder = t4) \vee (p3.initiator = t4 \wedge p3.responder = t3)) \wedge se3.event = re.event)$
 \Rightarrow
 $(\{p\} \subseteq re'.protocols \wedge \{p\} \subseteq se'.protocols)$

Transformation 3 covers the case depicted in Figure 36 that is essentially the mirror image of Transformation 2. There are two tasks that have at least one set of corresponding events and those tasks have a protocol between them. Now the SendEvent must be unique within Task1, and it is acceptable for Task1 to have a protocol with another task that also has a corresponding ReceiveEvent.

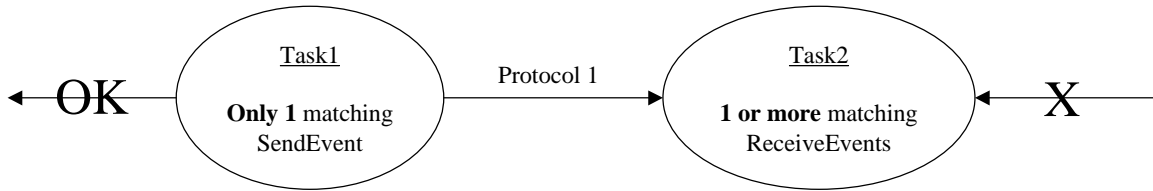


Figure 36 – Example of Transformation 3

Transformation 3

$\forall t, t2, t3 : \mathbf{Task}, st, st2, st3 : \mathbf{StateTable}, trans, trans2 : \mathbf{Transition}, se : \mathbf{SendEvent}, re : \mathbf{ReceiveEvent},$
 $p : \mathbf{Protocol} \bullet$
 $((p.initiator = t \wedge p.responder = t2) \vee (p.initiator = t2 \wedge p.responder = t)) \wedge st = t.stateTable$
 $\wedge st2 = t2.stateTable \wedge trans \in st.transitions \wedge trans2 \in st2.transitions \wedge se \in trans.sendEvents$
 $\wedge re = trans2.receiveEvent \wedge se.event = re.event$
 $\wedge \neg(\exists trans3 : \mathbf{Transition}, re2 : \mathbf{ReceiveEvent} \bullet trans3 \in st.transitions$
 $\wedge re2 = trans3.receiveEvent \wedge re2 \neq re \wedge re2.event = re.event)$
 $\wedge \neg(\exists t4 : \mathbf{Task}, st4 : \mathbf{StateTable}, trans4 : \mathbf{Transition}, re3 : \mathbf{ReceiveEvent}, p3 : \mathbf{Protocol} \bullet$
 $p3 \neq p \wedge t4 \neq t \wedge st4 = t4.stateTable \wedge trans4 \in st4.transitions \wedge re3 = trans4.receiveEvent$
 $\wedge ((p3.initiator = t3 \wedge p3.responder = t4) \vee (p3.initiator = t4 \wedge p3.responder = t3)) \wedge re3.event = re.event)$
 \Rightarrow
 $(\{p\} \subseteq se'.protocols \wedge \{p\} \subseteq re'.protocols)$

After the first three transformations have been applied, there may still be some cases where, due to ambiguity, the transformations were unable to automatically determine that an event is intended to belong to a protocol. In each case, the developer must determine whether or not the event belongs to the protocol. The ambiguous cases can be identified by an external protocol between two tasks with corresponding external events that were not automatically determined to belong to the protocol.

Figure 37 illustrates the case where it is impossible to automatically determine to which protocols a SendEvent belongs. Task1 has more than one SendEvent and participates in more than one protocol with other tasks that have corresponding ReceiveEvents. Some of the SendEvents may belong to only one protocol and not the other, while some SendEvents could belong to both. The developer has to make the determination.

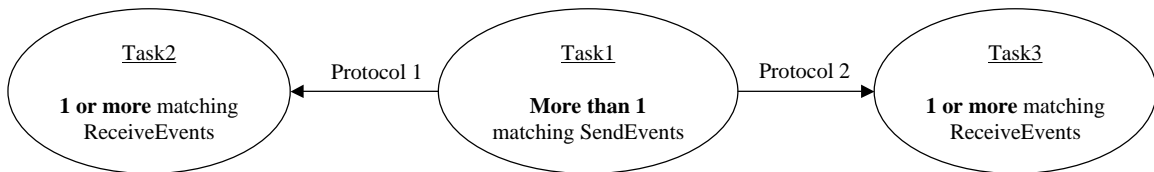


Figure 37 – Ambiguous Protocols for SendEvents

Figure 38 is similar to Figure 37, and illustrates the case when it is impossible to automatically determine to which protocols a ReceiveEvent belongs. The ReceiveEvents in Task1 may belong to Protocol1, Protocol2, or both.

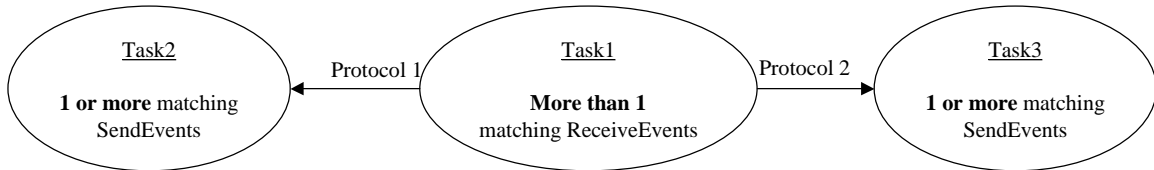


Figure 38 – Ambiguous Protocols for ReceiveEvents

3.2.2 Creating Components for Agents from Tasks

At this point, the designer must have already determined the set of roles each agent class will play. Transformation 4 states that for every task of every role that an agent plays, a component is created for that task. The component’s state table is initially the same as the state table of the task for which it was generated, and the component’s name is the name of the task. The rest of the transformation process is centered around these component state tables.

As an example of how Transformation 4 creates components for agent classes, consider Figure 39 as the Role Model created in the analysis phase. If the developer decides in the design phase to create the agent classes with the roles shown in Figure 40, then Transformation 4 creates the components shown for the agents. Since both agents play Role 2, there is a component created for each agent for Role 2’s Task 2. Figure 40 is not a MaSE diagram, but is presented to illustrate the internal agent components based on the initial Agent Class Diagram.

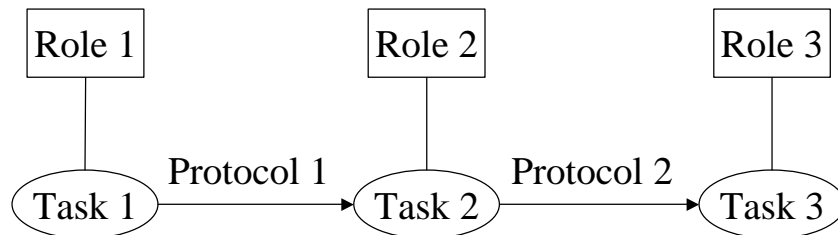


Figure 39 – Role Model Example

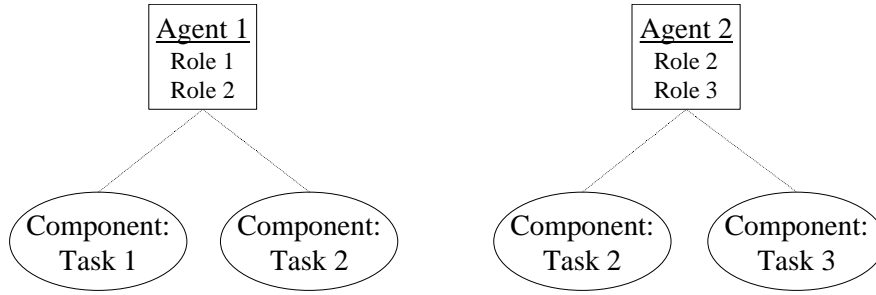


Figure 40 – Agent Components Created From the Roles' Tasks

Transformation 4

$\forall a : \mathbf{Agent}, r : \mathbf{Role}, t : \mathbf{Task} \bullet$

$(r \in a.roles \wedge t \in r.tasks)$

\Rightarrow

$(\exists c : \mathbf{Component} \bullet c \in a'.components \wedge c.stateTable = t.stateTable \wedge c.name = t.name)$

3.2.3 Replicating Protocols Between Components

Next, for each protocol in the analysis, Transformation 5 creates corresponding protocols in the design. Protocols in the analysis phase are defined between tasks. Transformation 4 created agent components based on the roles that each agent plays and the tasks that those roles have, protocols in the design phase must be between the components created for those tasks. Also, a role may be played by many different agent classes, and the tasks of that role are duplicated as components of all agents that play that role. Therefore, the protocols between those tasks are also duplicated for every component that was created from the task. This is done so that the designer can define whether, for every protocol between roles that are combined together, that protocol is now internal instead of external.

As an example of how the protocols might be duplicated in the design based on the Role Model and the roles chosen for agent classes, consider again our example from Figure 39 and Figure 40. Figure 41 takes the example one step further and illustrates how the protocols from the analysis phase are replicated in the design. Since both Agent 1 and Agent 2 have a component created from Task 2, and both Protocol 1 and Protocol 2 involve Task 2, there are two instances of each protocol in the design between

each component. The protocols shown in Figure 41 are illustrated purely for the purposes of the example.

There is currently no model in MaSE that depicts protocols between agent components.

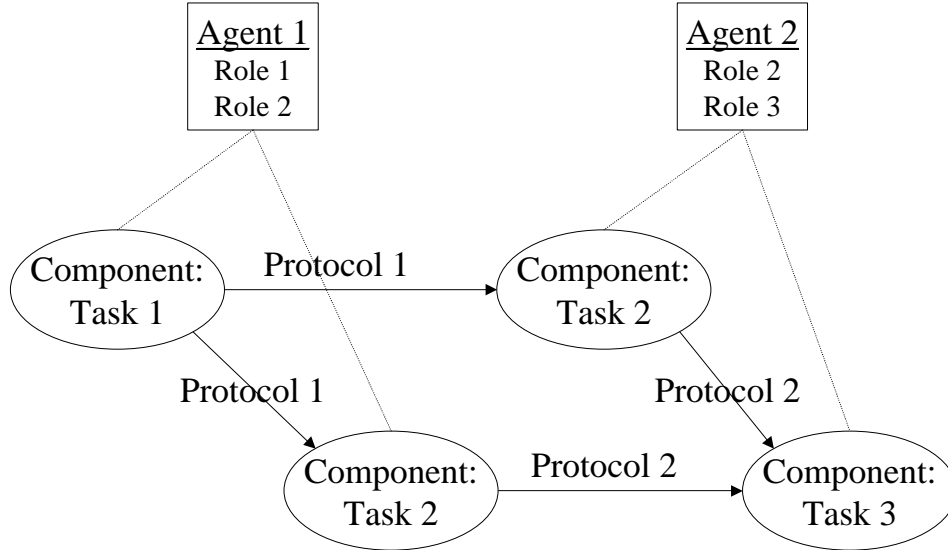


Figure 41 – Agent Diagram Example

Transformation 5

$\forall p : \mathbf{Protocol}, r, r2 : \mathbf{Role}, t, t2 : \mathbf{Task}, c, c2 : \mathbf{Component}, a, a2 : \mathbf{Agent} \bullet$

$(r \in a.roles \wedge t \in r.tasks \wedge c \in a.components \wedge c.name = t.name \wedge p.initiator = t \wedge r2 \in a2.roles$

$\wedge t2 \in r2.tasks \wedge c2 \in a2.components \wedge c2.name = t2.name \wedge p.responder = t2)$

\Rightarrow

$(\exists p2 : \mathbf{Protocol} \bullet p2.name = p.name \wedge p2.initiator = p.initiator \wedge p2.responder = p.responder$

$\wedge p2.initComp = c \wedge p2.respComp = c2 \wedge p2.mode = p.mode)$

At this point, the set of protocols for external events in the components are identical to the protocols defined between the tasks in the analysis phase. The next two transformations update those protocols to denote the protocols between the components. Transformation 6 converts the set of protocols for ReceiveEvents and Transformation 7 for SendEvents. The previous examples are used to illustrate the importance of these transformations. Assume that in Figure 39 there is an external event E in Task 2 that belongs to Protocol 1. Figure 41 illustrates the agents, component, and components after Transformation 4 and Transformation 5. In Agent 2's Task 2 component, the external event E must belong to the instance of

Protocol 1 between Agent 1's Task 1 component and Agent 2's Task 2 component, not to the other instance of Protocol 1 between Agent 1's Task 1 and Task 2 components.

Transformation 6

$$\begin{aligned} & \forall p, p2 : \mathbf{Protocol}, t, t2 : \mathbf{Task}, c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, re : \mathbf{ReceiveEvent} \bullet \\ & (st = c.stateTable \wedge t \in st.transitions \wedge re = t.receiveEvent \wedge p \in re.protocols \wedge p.initiator = p2.initiator \\ & \wedge p.responder = p2.responder \wedge (p2.initComp = c \vee p2.respComp = c)) \\ & \Rightarrow \\ & (p2 \in re'.protocols \wedge p \notin re'.protocols) \end{aligned}$$

Transformation 7

$$\begin{aligned} & \forall p, p2 : \mathbf{Protocol}, t, t2 : \mathbf{Task}, c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, se : \mathbf{SendEvent} \bullet \\ & (st = c.stateTable \wedge t \in st.transitions \wedge se \in t.sendEvents \wedge p \in se.protocols \wedge p.initiator = p2.initiator \\ & \wedge p.responder = p2.responder \wedge (p2.initComp = c \vee p2.respComp = c)) \\ & \Rightarrow \\ & (p2 \in se'.protocols \wedge p \notin se'.protocols) \end{aligned}$$

3.2.4 Transforming External Events into Internal Events

For each pair of roles that are combined into an agent class, the designer must determine whether each protocol that exists between components of that agent is either internal or external. This was also done for protocols between tasks of the same role in the analysis phase. If a protocol is defined as internal, all external ReceiveEvents and SendEvents that belong to the protocol are converted into internal receive and send Events. Transformation 8 describes how external SendEvents are converted to an internal Event in the *sends* clause, and Transformation 9 describes how an external ReceiveEvent is converted into an internal Event in the *receive* clause.

For example, if the ReceiveEvent, *receive(msg(x, y), agent)*, is part of a protocol that is determined to be internal, the event is changed to *msg(x, y)*. It should be noted that in order for an external event to be transformed into an internal event, every protocol that the event belongs to must be designated as internal.

If an event belongs to both internal and external protocols, an error has been made and it must be corrected before the transformation process can continue.

Transformation 8

$$\begin{aligned} & \forall c : \mathbf{Component}, p : \mathbf{Protocol}, st : \mathbf{StateTable}, t : \mathbf{Transition}, se : \mathbf{SendEvent} \bullet \\ & ((p.\text{initComp} = c \vee p.\text{respComp} = c) \wedge st = c.\text{stateTable} \wedge t \in st.\text{transitions} \wedge se \in t.\text{sendEvents} \\ & \wedge se.\text{protocol} = p \wedge p.\text{mode} = \text{"internal"}) \\ & \Rightarrow \\ & (se.\text{event} \in t'.\text{sends} \wedge se \notin t'.\text{sendEvents}) \end{aligned}$$

Transformation 9

$$\begin{aligned} & \forall c : \mathbf{Component}, p : \mathbf{Protocol}, st : \mathbf{StateTable}, t : \mathbf{Transition}, re : \mathbf{ReceiveEvent} \bullet \\ & ((p.\text{initComp} = c \vee p.\text{respComp} = c) \wedge st = c.\text{stateTable} \wedge t \in st.\text{transitions} \wedge re = t.\text{receiveEvent} \\ & \wedge re \neq \text{null} \wedge re.\text{protocol} = p \wedge p.\text{mode} = \text{"internal"}) \\ & \Rightarrow \\ & (re.\text{event} = t'.\text{receive} \wedge t'.\text{receiveEvent} = \text{null}) \end{aligned}$$

3.3 Annotating Component State Diagrams

Now that components have been created for the agent classes that represent the concurrent tasks from the analysis phase, the next stage of the transformation process (highlighted in Figure 42) is centered around annotating the component state tables for the removal of the conversations. There are many different cases in which tasks can be defined in the analysis phase that make removing conversations problematic, such as events being received or sent on transitions that do not belong to the same conversation. The transformations in this section first convert the component's state tables into a canonical form to simplify harvesting the conversations from them. Then the state tables are annotated to indicate where each conversation begins and ends. Finally, the starting points for the conversations in the different component state diagrams are matched.

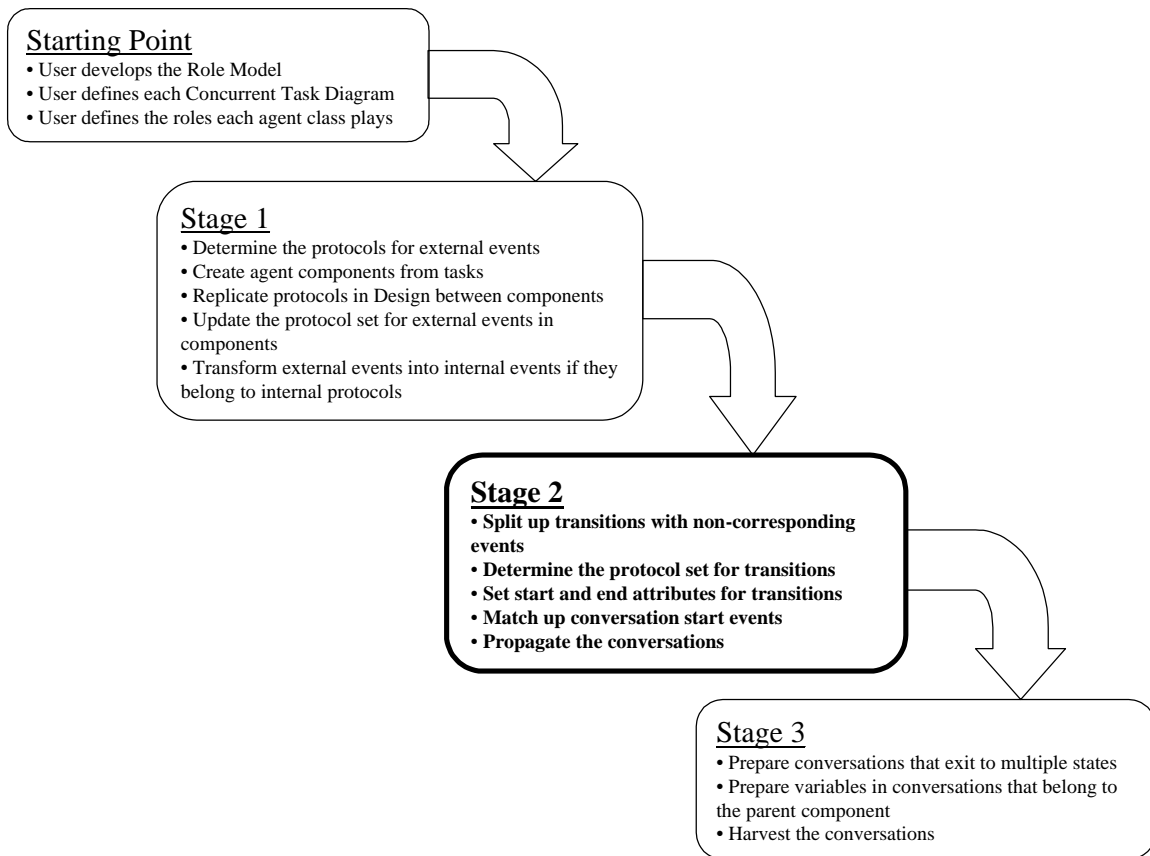


Figure 42 – Stage 2 in the Transformation Process

3.3.1 Splitting Transitions

Transitions in a component state table that have either multiple events that represent communication in different protocols, or some external and some internal communication, make it difficult to remove the conversations from the state table. Since a transition can only have either an external ReceiveEvent or an internal Receive Event, there is a transformation that handles each case. Transformation 10 covers transitions that have an external ReceiveEvent. The requirement for the transformation is that there is 1) an external ReceiveEvent and 2) either an internal send Event or an external SendEvent that belongs to a different set of protocols. All external SendEvents that have the same protocols as the ReceiveEvent are placed on the first transition along with the receiveEvent, the guard condition, and the actions, all of which are defined to take place before any transmitted events. The

internal send Events and the remaining external SendEvents with protocols that are different than the ReceiveEvent's protocols are placed on the second transition.

Transformation 10

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, re : \mathbf{ReceiveEvent} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge re = t.receiveEvent \wedge re \neq null \\
& \wedge ((\exists se : \mathbf{SendEvent} \bullet se \in t.sendEvents \wedge re.protocols \neq se.protocols) \vee (\exists e : \mathbf{Event} \bullet e \in t.sends))) \\
& \Rightarrow \\
& (\exists s : \mathbf{State}, t1, t2 : \mathbf{Transition}, n : \mathbf{String} \bullet s \in st'.states \wedge s \notin st.states \wedge t1 \in st'.transitions \\
& \wedge t2 \in st'.transitions \wedge s.name = ("Null" + n) \wedge \neg(\exists s2 : \mathbf{State} \bullet s \neq s2 \wedge s.name = s2.name) \wedge s.actions = \{ \} \\
& \wedge t1.receive = null \wedge t1.receiveEvent = t.receiveEvent \wedge t1.guard = t.guard \wedge t1.actions = t.actions \\
& \wedge t1.sends = \{ \} \wedge t1.from = t.from \wedge t1.to = s \wedge \neg(\exists t3 : \mathbf{Transition} \bullet t3.to = s \wedge t1 \neq t3) \\
& \wedge (\forall se1 : \mathbf{SendEvent} \bullet (se1 \in t.sendEvents \wedge re.protocols = se1.protocols) \Leftrightarrow se1 \in t1.sendEvents) \\
& \wedge t2.receive = null \wedge t2.receiveEvent = null \wedge t2.guard = null \wedge t2.actions = [] \wedge t2.sends = t.sends \\
& \wedge (\forall se2 : \mathbf{SendEvent} \bullet (se2 \in t.sendEvents \wedge re.protocols \neq se2.protocols) \Leftrightarrow se2 \in t2.sendEvents) \\
& \wedge t2.from = s \wedge t2.to = t.to \wedge t \notin st'.transitions \wedge \neg(\exists t3 : \mathbf{Transition} \bullet t3.from = s \wedge t2 \neq t3))
\end{aligned}$$

Figure 43 shows how Transformation 10 would split a transition. The sets above the events represent the protocols to which the events belong. The original transition has a ReceiveEvent that is part of protocol P1 and one SendEvent for protocol P1 and one SendEvent for protocol P2. After the transformation, the SendEvent for P1 is placed on the first transition with the ReceiveEvent and the SendEvent for P2 is placed on the second transition. The resulting transitions and null state are consistent with the semantics of the original transition. In order for the original transition to take place, both the guard condition must be met and the *do(a)* message is received from ag1. When the transition occurs the *ack* message is sent back to ag1, and the *do(a)* message is sent to ag2. After the transformation, the guard must be true and the *do(a)* message must be received from ag1 for the first transition to take place, sending the *ack* message back to ag1. There are no new actions that are done within the null state, and the second transition is automatically enabled, sending the *do(a)* message to ag2.

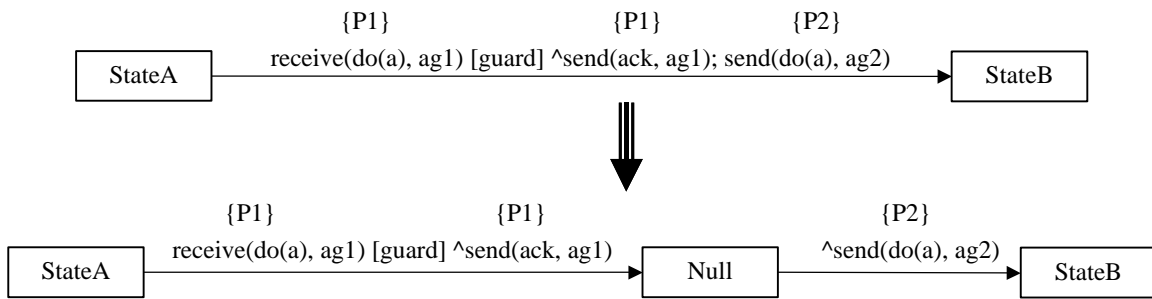


Figure 43 – Example of Splitting a Transition

As seen in the example shown in Figure 43, there was an ordering applied to the SendEvents. This is a design decision that is consistent with the original specification defined by the concurrent task diagrams. Since the SendEvents belong to different protocols (see Section 2.3.3.2.1), they are received by different components. Therefore, it makes no difference what order is chosen to send them. The first event is sent to one component, followed by the next event to the other component. Even if the different components belong to the same agent, they should both be waiting to receive the events, regardless of the order in which they are received.

Transformation 11 covers transitions that have either an internal receive or send event. The requirement for this transformation to take place is that there is 1) either an internally received or sent Event and 2) at least one external SendEvent in the sendEvents clause. Any internal receive or send Events are placed on the first transition along with the original transition's guard and actions. The second transition simply contains the set of external SendEvents.

Transformation 11

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge (t.receive \neq null \vee t.sends \neq \{\}) \wedge t.sendEvents \neq \{\})$

\Rightarrow

$(\exists s : \mathbf{State}, t1, t2 : \mathbf{Transition}, n : \mathbf{String} \bullet s \in st'.states \wedge s \notin st.states \wedge t1 \in st'.transitions$

$\wedge t2 \in st'.transitions \wedge s.name = ("Null" + n) \wedge \neg(\exists s2 : \mathbf{State} \bullet s2 \neq s \wedge s2.name = s.name) \wedge s.actions = \{\}$

$\wedge t1.receive = t.receive \wedge t1.receiveEvent = null \wedge t1.guard = t.guard \wedge t1.actions = t.actions$

$\wedge t1.sends = t.sends \wedge t1.sendEvents = \{\} \wedge t1.from = t.from \wedge t1.to = s \wedge t2.receive = null \wedge t2.sends = \{\}$

$\wedge t2.receiveEvent = null \wedge t2.guard = null \wedge t2.actions = [] \wedge t2.sendEvents = t.sendEvents \wedge t2.from = s$

$\wedge t2.to = t.to \wedge t \notin st'.transitions \wedge \neg(\exists t3 : \mathbf{Transition} \bullet (t3.to = s \wedge t1 \neq t3) \vee (t3.from = s \wedge t2 \neq t3))$)

Figure 44 illustrates how Transformation 11 would split a transition that has both internal events and external SendEvents. The original transition has both an internal receive and send Event, as well as an external SendEvent. After the transformation only the internal events are placed on the first transition and the external SendEvents (in this case only one) are placed on the second transition. Again, the state diagram after the transformation is consistent with the semantics of the original state diagram. In both cases, $do(a)$ must be internally received, the guard condition must be true, and the internal *acknowledge* event is sent, as well as the external SendEvent belonging to protocol P1. The transmissions in the resulting state diagram have been ordered, but since the events are being sent to different components, they are still consistent with the original state diagram.

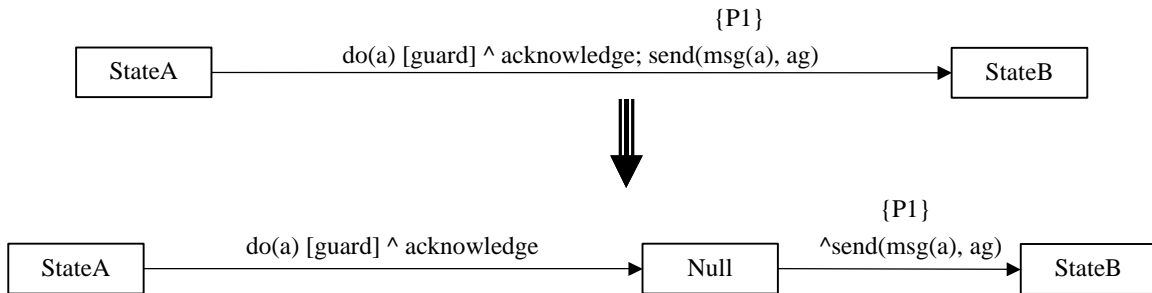


Figure 44 – Example 2 of Splitting a Transition

3.3.2 Determining the Protocols for Transitions

The next step of the transformation process is to annotate the component state tables to show where each conversation begins and ends. In order to simplify this process, each transition is labeled with a set of protocols that represents the external protocols in which the transition may participate. If a transition has a non-empty set of protocols, then the communication that takes place on that transition at any given time will be with only one of the protocols in the set, not all of them. If a transition has an empty set of protocols, then either there is no external communication taking place, or there is communication with more than one agent that takes place. Later, the set of protocols is the primary factor for determining where conversations start and end.

Table 1 shows the rules for determining the set of protocols for a transition. The first five columns show the properties for the transition being labeled. Transformation 10 and Transformation 11 split up transitions with events that do not correspond to each other, so Table 1 shows the only possible combinations for the transition being labeled. There will be no transitions with a) an internal receive Event and an external ReceiveEvent, b) an internal receive Event and external SendEvents, c) an external ReceiveEvent and internal send Events, d) a ReceiveEvent and SendEvents that don't correspond (i.e. different protocols), or e) internal send Events and external SendEvents.

An "x" in the table represents a "don't care" in a traditional logic table. Under the SendEvents column, "same protocols" means that every SendEvent on the transition has the same set of protocols, and "different protocols" means that not all SendEvents have the same set. The "Union" label means that the set of protocols is the union of all protocol sets on transitions into the *from* state.

Table 1 – Rules for Determining a Transition’s Set of Protocols

Transition being labeled						Protocols for Transitions into the <i>from</i> state	Protocols for Transitions out of the <i>from</i> state	Resulting Set of Protocols	Transformation
receive	ReceiveEvent	guard	sends	SendEvents	Actions				
no	yes	x	no	x	x	x	x	ReceiveEvent's	12
no	no	x	no	same protocols	x	x	x	SendEvent's	13
no	no	x	no	different protocols	x	x	x	{}	14
no	no	x	no	to <list>	x	x	x	{}	15
yes	no	x	x	no	x	x	x	{}	16
x	no	x	yes	no	x	x	x	{}	16
no	no	x	no	no	x	!= { }	Union	Union	17
no	no	x	no	no	x	{ }	x	{ }	18
no	no	x	no	no	x	x	{ }	{ }	18
no	no	x	no	no	x	x	!= Union	{ }	18

Transformation 12 sets the protocols for all transitions that have a non-null receiveEvent attribute. Transformation 10 ensured that transitions with an external ReceiveEvent only have SendEvents that have the same set of protocols. Therefore, it is certain that the set of protocols for the transition can be the same as that of the ReceiveEvent.

Transformation 12

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, re : \mathbf{ReceiveEvent} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge re = t.receiveEvent \wedge re \neq null \wedge t.receive = null \wedge t.sends = \{ \})$

$\wedge \neg(\exists se : \mathbf{SendEvent} \bullet se \in t.sendEvents \wedge re.protocols \neq se.protocols)$

\Rightarrow

$t'.protocols = re.protocols$

If there is a transition with no internal events, no ReceiveEvent, and all external SendEvents have the same protocols, and the none of the recipients of the SendEvents is a list, then Transformation 13 sets the protocols of the transition to the SendEvents’ set of protocols. While the set of protocols for the SendEvents may contain more than one protocol, it is assumed that the events that are sent belong to only one protocol at a time. If there is at least one SendEvent with different protocols, then Transformation 14 sets the protocols to the empty set, since that transition contains communication that may belong to two different protocols at the same time.

Transformation 13

$$\begin{aligned} & \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, se : \mathbf{SendEvent} \bullet \\ & (st = c.stateTable \wedge t \in st.transitions \wedge t.receiveEvent = null \wedge t.receive = null \wedge t.sends = \{ \} \\ & \wedge se \in t.sendEvents \wedge \neg isList(se.recipient) \\ & \wedge \neg (\exists se2 : \mathbf{SendEvent} \bullet se2 \in t.sendEvents \wedge se2 \neq se \wedge se2.protocols \neq se.protocols)) \\ & \Rightarrow \\ & t'.protocols = se.protocols \end{aligned}$$

Transformation 14

$$\begin{aligned} & \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet \\ & (st = c.stateTable \wedge t \in st.transitions \wedge t.receive = null \wedge t.sends = \{ \} \\ & \wedge (\exists se1, se2 : \mathbf{SendEvent} \bullet se1 \neq se2 \wedge se1 \in t.sendEvents \wedge se2 \in t.sendEvents \\ & \quad \wedge se1.protocols \neq se2.protocols)) \\ & \Rightarrow \\ & t'.protocols = \{ \} \end{aligned}$$

If a transition has an external SendEvent to a list (a multicast), then Transformation 15 sets the transition's protocols to the empty set, not because the transition contains communication to different protocols at the same time, but because a multicast implies simultaneous communication with a different instance of the protocol for each agent represented in the list. The isList(String) function returns true if the string representing the recipient is of the form <list-name>.

Transformation 15

$$\begin{aligned} & \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, se : \mathbf{SendEvent} \bullet \\ & (st = c.stateTable \wedge t \in st.transitions \wedge se \in t.sendEvents \wedge isList(se.recipient)) \\ & \Rightarrow \\ & t'.protocols = \{ \} \end{aligned}$$

Transformation 16 states that if a transition has an internal event, then the protocols set must be the empty set, denoting that no communication with external protocols takes place on the transition. Transformation 11 split transitions that had both internal and external events, so at this point any transitions

that have at least one internal event are assured to have no external events, and therefore belong to no external protocols.

Transformation 16

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge (t.receive \neq null \vee t.sends \neq \{\})) \\
& \Rightarrow \\
& t'.protocols = \{\}
\end{aligned}$$

If there is a transition with no internal or external events that are received or sent, then the transition by itself gives no information as to what the protocols set should be. It is not necessarily empty since the set of protocols represents the current communication that is taking place and is used in determining if a transition is the start or end of a conversation. Other factors are used to determine the protocols of these transitions. If every transition to or from the transition's *from* state have non-empty protocols, and every transition leaving the *from* state contains the union of the protocols for all transitions into the *from* state, then Transformation 17 also makes the set of protocols for the transition in question the union of all protocols of the transitions into the *from* state. Otherwise, there has been a change in the active protocol, and Transformation 18 gives the transition the empty set of protocols.

Transformation 17

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge t.receive = null \wedge t.receiveEvent = null \wedge t.sends = \{\} \\
& \wedge t.sendEvents = \{\} \\
& \wedge \neg(\exists t2 : \mathbf{Transition} \bullet t2 \in st.transitions \wedge t2 \neq t \wedge t2.to = t.from \wedge t2.protocols = \{\}) \\
& \wedge \neg(\exists t3 : \mathbf{Transition} \bullet t3 \in st.transitions \wedge t2 \neq t \wedge t3.from = t.from \wedge t3.protocols = \{\}) \\
& \wedge (\forall t4 : \mathbf{Transition}, p : \mathbf{Protocol} \bullet t4 \in st.transitions \wedge t4 \neq t \wedge t4.from = t.from \wedge p \in t4.protocols \\
& \quad \wedge p \neq null \Leftrightarrow (\exists t5 : \mathbf{Transition} \bullet t5 \in st.transitions \wedge t5.to = t4.from \wedge p \in t5.protocols)) \\
& \Rightarrow \\
& (\forall t6 : \mathbf{Transition} \bullet (t6 \neq t \wedge t6.to = t.from) \Rightarrow t'.protocols = (t.protocols \cup t6.protocols))
\end{aligned}$$

Transformation 18

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet$
 $(st = c.stateTable \wedge t \in st.transitions \wedge t.receive = null \wedge t.receiveEvent = null \wedge t.sends = \{ \}$
 $\wedge t.sendEvents = \{ \}$
 $\wedge ((\exists t2 : \mathbf{Transition} \bullet t2 \in st.transitions \wedge t2 \neq t \wedge t2.to = t.from \wedge t2.protocols = \{ \})$
 $\vee (\exists t3 : \mathbf{Transition} \bullet t3 \in st.transitions \wedge t3 \neq t \wedge t3.from = t.from \wedge t3.protocols = \{ \})$
 $\vee \neg (\forall t4 : \mathbf{Transition}, p : \mathbf{Protocol} \bullet t4 \in st.transitions \wedge t4.from = t.from \wedge p \in t4.protocols \wedge p \neq null$
 $\Leftrightarrow (\exists t5 : \mathbf{Transition} \bullet t5 \in st.transitions \wedge t5.to = t4.from \wedge p \in t5.protocols)))$
 \Rightarrow
 $t'.protocols = \{ \}$

The next example illustrates how Transformation 17 determines the protocols for transitions that have no events. Figure 45 shows a state diagram with three different transitions with no events. The sets in the figure show the protocols for the transitions. The transition leaving State1 is an automatic transition and has no events. However, the only transition into State1 has {P1} as its set of protocols. Since there is no indication that the active protocol has changed, Transformation 17 sets the protocols for the transition leaving State1 to {P1}. In the same way, the transition leaving State2 receives the protocol set {P2}. The resulting state diagram is shown in Figure 46.

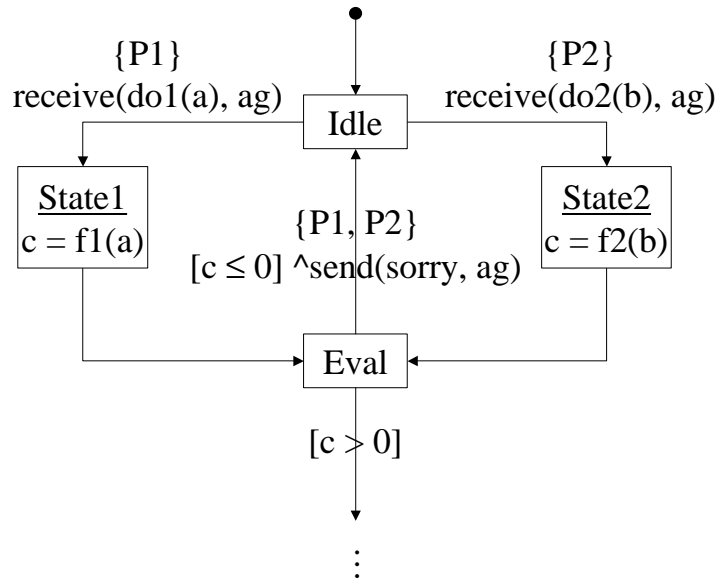


Figure 45 – Transitions With No Events

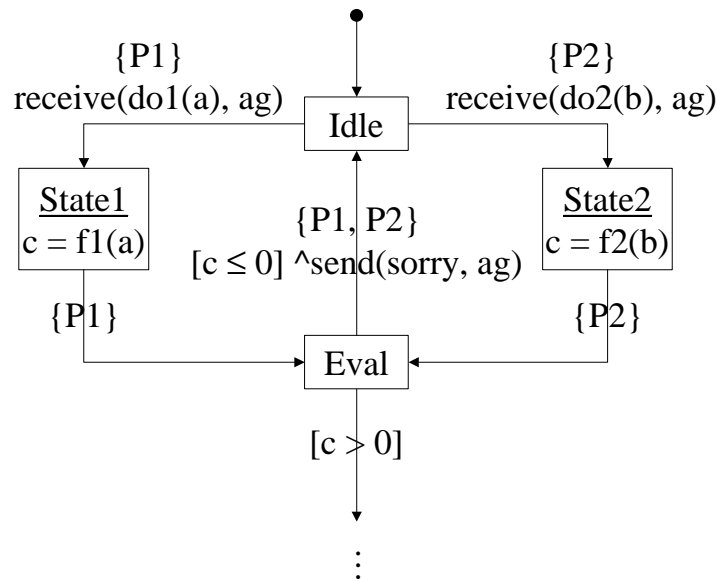


Figure 46 – Protocols Determined for Two Transitions

The more interesting case is the transition leaving the Eval state. It only has a guard condition and no events, yet Transformation 17 determines that the set of protocols should be $\{P1, P2\}$, the union of the protocols of the transitions into the Eval state. This is because the protocols of the other transition leaving the Eval state is also the union of the transitions into the Eval state (determined either by the first three transformations or by the designer), and there are no transitions into or out of the Eval state with an empty set of protocols. The resulting state diagram is shown in Figure 47. As you can see from this example, in order for these transformations to be executed correctly, all other transitions into or out of its *from* state must already be determined. For example, the protocols for the transition out of the Eval state with only the guard condition could not be determined correctly if the protocols for the transitions into the Eval state had not already been determined.

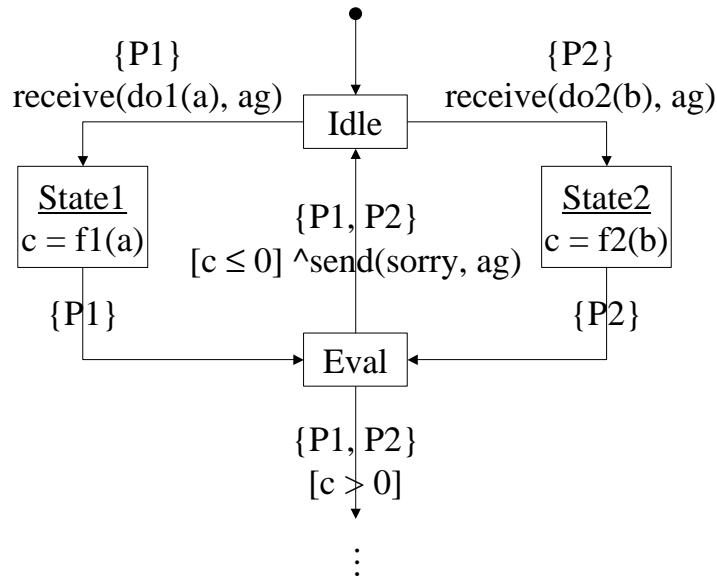


Figure 47 – Protocols Determined for All Transitions

3.3.3 Start Label for Transitions

Now that all transitions have a set of protocols, the next step is to determine where conversations begin and end. There are many reasons to label a transition as the start of a new conversation. However, since every transition already has its protocols set, the rules are greatly simplified. The protocols indicate with whom the communication takes place. The following six conditions indicate the start of a conversation by a change in who the agent is communicating with, which in most cases is due to a change in the protocols.

1. A transition has a protocol not found in at least one transition into its *from* state (Transformation 19).
2. A transition has a non-empty set of protocols that is different than another transition leaving the same state (Transformation 20).
3. A transition has a non-empty set of protocols, yet lacks a protocol of another transition into its *from* state (Transformation 21).
4. A transition has a non-empty set of protocols, and there is another transition into or out of its *from* state with an empty set of protocols (Transformation 22).

5. A transition has an empty set of protocols and at least one SendEvent (Transformation 23).
6. A transition has a SendEvent whose recipient was previously determine by an action (Transformation 24).

Transformation 19 states that when a transition has a protocol in its set of protocols that is not found in at least one transition into its *from* state, then the transition must be the start of a new conversation. In this case, the transition has communication that belongs to a protocol not previously active, so the communication to the newly active protocol will be a new conversation. The most obvious example of this is when there is a complete change in the set of protocols from one transition to the next. Figure 48 illustrates another example of when Transformation 19 would label a transition as the start of a conversation. The sets indicate the protocols set for the transitions, and the letter S over the transition indicates it has been labeled as the start of a conversation. The attributes of the transitions are not shown in these examples, because it is only the set of protocols that matters in these transformations. In the portion of the state diagram shown in Figure 48, the transition leaving State1 has both P1 and P2 as protocols, but it also has P3 and there is no transition into State1 with P3 as a protocol, so the transition becomes the start of a conversation.

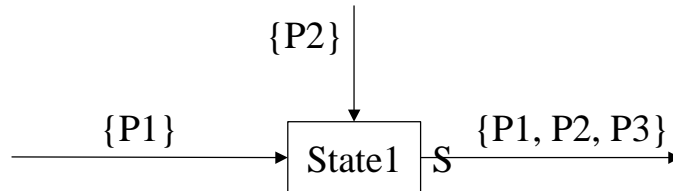


Figure 48 – Example of Transformation 19

Transformation 19

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, p : \mathbf{Protocol} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge t.protocols \neq \{ \} \wedge p \in t.protocols$

$\wedge \neg(\exists t2 : \mathbf{Transition} \bullet t2 \in st.transitions \wedge t2.to = t.from \wedge p \in t2.protocols)$

\Rightarrow

$t'.start = true$

Transformation 20 states that if there is a transition with a non-empty set of protocols leaving a state and there is another transition with different protocols leaving the same state, then the transition must be the start of a conversation. These transitions cannot be the continuation of a previous conversation, because they have different protocols that may be active when leaving the *from* state. Figure 49 illustrates one example of how Transformation 20 would label a transition as the start of a conversation. In the example, there are two transitions with different protocols both leaving State1. These transitions must be the start of conversations because it is unclear which transition would be enabled from State1 and therefore which protocol would be active in communication.

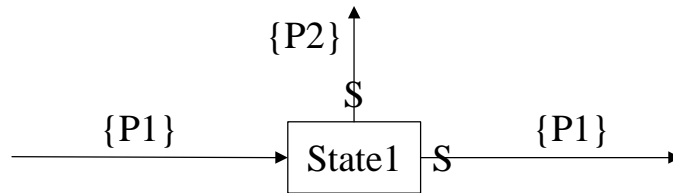


Figure 49 – Example of Transformation 20

Transformation 20

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge t.protocols \neq \{ \})$

$\wedge (\exists t2 : \mathbf{Transition} \cdot t2 \in st.transitions \wedge t2.from = t.from \wedge t2.protocols \neq t.protocols)$

\Rightarrow

$t.start = true$

Transformation 21 states that if there is a transition with at least one protocol leaving a state and that transition lacks a protocol that another transition into the *from* state has, then the transition is the start of a conversation. The protocol that is missing for the transition leaving the *from* state may be the active protocol for the transition into the *from* state, so that conversation cannot continue and another one must begin. In the previous example, Figure 49 illustrates an instance where Transformation 21 would be applied. Since the transition with protocols {P2} leaving State1 does not have P1 as a protocol, and the transition into State1 does, the transition leaving State1 is the start of a conversation, regardless of the fact that there is another transition leaving State1 with different protocols.

Transformation 21

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet$
 $(st = c.stateTable \wedge t \in st.transitions \wedge t.protocols \neq \{\})$
 $\wedge (\exists t2 : \mathbf{Transition}, p : \mathbf{Protocol} \bullet t2 \in st.transitions \wedge t2.to = t.from \wedge p \in t2.protocols \wedge p \notin t.protocols))$
 \Rightarrow
 $t'.start = true$

Transformation 22 states that when a transition is labeled with at least one protocol and there is another transition either from or into its *from* state that has no protocols, then the transition is the start of a conversation. If there is a transition into the *from* state with an empty set of protocols, then it is possible that the transition was the one taken, and the protocol activated on the transition leaving the state must represent communication to a new agent. If there is a transition out of the *from* state with an empty set of protocols, then no transition leaving the from state can continue any previous conversation because the transition with no protocols may be the one taken.

As an example, consider Figure 50, where the transition leaving State1 is labeled with only protocol P1 and there is another transition with an empty set of protocols that is also leaving State1. Transformation 22 labels the transition with protocols {P1} as the start of a conversation. If the transition with no protocols was not present, then the transition leaving State1 with protocols {P1} would be guaranteed to continue the conversation.

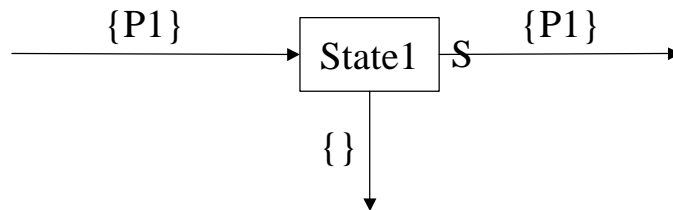


Figure 50 – Example of Transformation 22

Transformation 22

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet$
($st = c.stateTable \wedge t \in st.transitions \wedge t.protocols \neq \{\}$)
 $\wedge (\exists t2 : \mathbf{Transition} \bullet t2 \in st.transitions \wedge (t2.to = t.from \vee t2.from = t.from) \wedge t2.protocols = \{\})$
 \Rightarrow
 $t'.start = true$

Transformation 23 states that if a transition labeled with no protocols has a SendEvent, then the transition starts a new conversation. This covers the following two possibilities:

1. The transition has more than one SendEvents and they have different recipients.
2. The transition has a SendEvent to a list.

In each case, there is communication with more than one agent. Therefore, there are multiple instances of conversations that take place on the transition, and the conversations result in simple “single-transition” conversations. Later, Transformation 26 also labels these transitions as the end of the conversation.

Transformation 23

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition} \bullet$
($st = c.stateTable \wedge t \in st.transitions \wedge t.sendEvents \neq \{\} \wedge t.protocols = \{\}$)
 \Rightarrow
 $t'.start = true$

Transformation 24 simply states that if there is an action, either in a state or on a transition, that determines the recipient of a SendEvent on a subsequent transition, then the transition starts a new conversation. In most cases, that action will occur in the transition’s *from* state, but there could be states and transitions between the setting of that variable and its use in the SendEvent. The *isAssigned(SendEvent, Transition, StateTable)* function is defined in Appendix B, and takes care of these cases by recursively searching back from the transition to determine if there is an action that sets the recipient of the SendEvent. Figure 51 shows one example where Transformation 24 would apply. There

is an action in State1 that sets the recipient of the SendEvent on the transition leaving that state. Since the action just determined who the communication in the SendEvent would be with, the transition is the start of a new conversation.

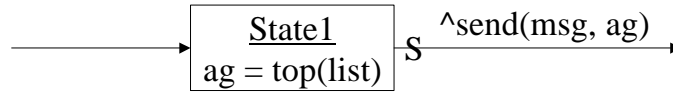


Figure 51 – Example of Transformation 24

Transformation 24

$\forall c : \text{Component}, st : \text{StateTable}, t : \text{Transition}, se : \text{SendEvent} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge se \in t.sendEvents \wedge isAssigned(se, t, st) \wedge t.protocols \neq \{\})$

\Rightarrow

$t.start = true$

3.3.4 End Label for Transitions

In the same way that the set of protocols for transitions are used to determine the start of conversations, they are also used to determine where conversations end. The following four conditions indicate the end of a conversation.

1. A transition has a protocol not found in a transition leaving its *to* state (Transformation 25).
2. A transition has an empty set of protocols and at least one SendEvent (Transformation 26).
3. A transition has a non-empty set of protocols and there is a start transition leaving its *to* state (Transformation 27).
4. A transition to the end state has a non-empty set of protocols (Transformation 28).

Transformation 25 states that if there is a transition with a protocol and there is another transition leaving its *to* state that does not also have that protocol, then the transition must be the end of a conversation because that protocol might not continue to have active communication. Figure 52 shows an example of how Transformation 25 would apply. The transition into State1 has P1 as a protocol, but the

transition leaving State1 has the empty set as its set of protocols. Thus, the transition with protocols {P1} is labeled as the end of a conversation.



Figure 52 – Example of Transformation 25

Transformation 25

$\forall c : \text{Component}, st : \text{StateTable}, t : \text{Transition}, p : \text{Protocol} \bullet$
 $(st = c.stateTable \wedge t \in st.transitions \wedge p \in t.protocols \wedge p \neq \text{null}$
 $\wedge \neg(\forall t2 : \text{Transition} \bullet t2 \in st.transitions \wedge t.to = t2.from \wedge p \in t2.protocols)$
 \Rightarrow
 $t'.end = \text{true}$

Transformation 26 is the corresponding transformation to Transformation 20, and designates any transition with an empty set of protocols and at least one external SendEvent as the end of a conversation. As stated earlier this occurs when either 1) there are external SendEvents with different recipients or 2) there is a SendEvent to a list. Each case represents multiple conversation instances that take place on the transition, so the transition is both the start and end of the conversation(s).

Transformation 26

$\forall c : \text{Component}, st : \text{StateTable}, t : \text{Transition} \bullet$
 $(st = c.stateTable \wedge t \in st.transitions \wedge t.sendEvents \neq \{\} \wedge t.protocols = \{\})$
 \Rightarrow
 $t'.end = \text{true}$

Transformation 27 is straightforward, and states that if a transition has a non-empty set of protocols and its *to* state is the *from* state of a transition that is marked as the start of a new conversation, then that transition is the end of the conversation. This must be the case so that the next conversation can start.

Transformation 27

$\forall c : \text{Component}, st : \text{StateTable}, t, t2 : \text{Transition} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge t.protocols \neq \{\} \wedge t.to = t2.from \wedge t2.start = true)$

\Rightarrow

$t'.end = true$

Transformation 28 describes the last reason that a transition can be labeled the end of a conversation, which is when a transition has a non-empty set of protocols and its *to* state is the end state. It is obvious in this situation that the conversation must end because the state table ends.

Transformation 28

$\forall c : \text{Component}, st : \text{StateTable}, t : \text{Transition}, s : \text{State} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge t.protocols \neq null \wedge s \in st.states \wedge s = t.to \wedge s.name = "end")$

\Rightarrow

$t'.end = true$

3.3.5 Matching Conversation Halves

After all of the components' state diagrams have been annotated, the different conversation halves, as annotated, must be matched. As the events are matched, they are given the same convName. Events may be matched with more than one corresponding event, so in practice, *every* matching set of events would receive the same conversation name, which may ripple through as new matches are made. Once all conversation halves have been matched, Conversations can be created to represent the communication. The protocols between the components provide a way to determine to which component the corresponding halves belong. The next two transformations define how, in some cases, the different conversation halves can be automatically matched. The transformations are very similar to the transformations used to determine the set of protocols for external events. Again, not all matches can be made automatically. In some cases, the developer must determine whether a message in a SendEvent is actually meant to be received by a ReceiveEvent in another component.

Transformation 29 is essentially the same as Transformation 2. It covers the conditions illustrated in Figure 53, where there is a protocol between two components that have corresponding events. One component has at least one corresponding SendEvent, and the other component has only one corresponding ReceiveEvent. The component with the SendEvent cannot have a protocol with another component that has a corresponding ReceiveEvent, while the component with the unique ReceiveEvent may have a protocol with another component with a corresponding SendEvent.

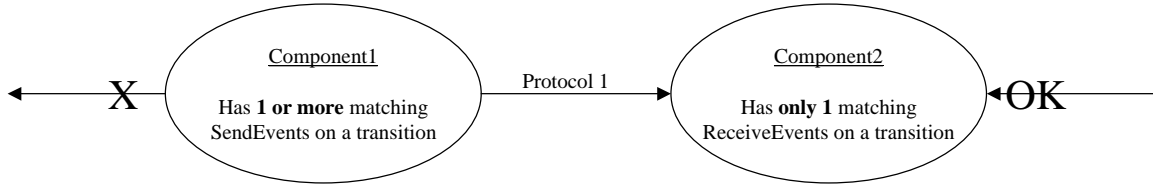


Figure 53 – Example of Transformation 29

Transformation 29

$\forall c1, c2 : \mathbf{Component}, p : \mathbf{Protocol}, st1, st2 : \mathbf{StateTable}, t1, t2 : \mathbf{Transition}, se : \mathbf{SendEvent},$
 $re : \mathbf{ReceiveEvent} \bullet$
 $(st1 = c1.stateTable \wedge st2 = c2.stateTable \wedge t1 \in st1.transitions \wedge t2 \in st2.transitions$
 $\wedge ((c1 = p.initComp \wedge c2 = p.respComp) \vee (c1 = p.respComp \wedge c2 = p.initComp))$
 $\wedge t1.start = true \wedge t2.start = true \wedge se \in t1.sendEvents \wedge re = t2.receiveEvent \wedge se.event = re.event$
 $\wedge \neg(\exists p2 : \mathbf{Protocol}, c3 : \mathbf{Component}, st3 : \mathbf{StateTable}, t3 : \mathbf{Transition}, se2 : \mathbf{SendEvent} \bullet$
 $p2 \neq p \wedge ((p2.initComp = c3 \wedge p2.respComp = c2) \vee (p2.initComp = c2 \wedge p2.respComp = c3))$
 $\wedge c3 \neq c1 \wedge st3 = c3.stateTable \wedge t3 \in st3.transitions \wedge se2 \in t3.sendEvents$
 $\wedge se2.event = re.event)$
 $\wedge \neg(\exists t3 : \mathbf{Transition}, se2 : \mathbf{SendEvent} \bullet t3 \neq t1 \wedge t3 \in st1.transitions \wedge se2 \in t3.sendEvents$
 $\wedge se2.event = se.event))$
 \Rightarrow
 $(\exists newName : \mathbf{String} \bullet se'.convName = newName \wedge re'.convName = newName)$

Transformation 30 is essentially the same as Transformation 3 and the mirror image to Transformation 29. The conditions for the transformation to apply are illustrated in Figure 54, where the SendEvent must be unique and its component is allowed to have a protocol with another component that has a corresponding ReceiveEvent. The ReceiveEvent in the other component is not required to be unique, but that component cannot have a protocol with another component with a corresponding SendEvent.

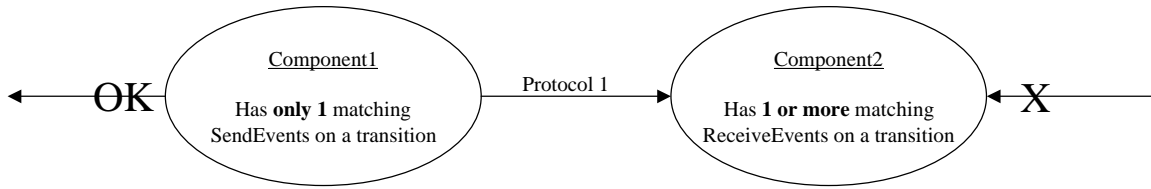


Figure 54 – Example of Transformation 30

Transformation 30

$\forall c1, c2 : \mathbf{Component}, p : \mathbf{Protocol}, st1, st2 : \mathbf{StateTable}, t1, t2 : \mathbf{Transition}, se : \mathbf{SendEvent},$
 $re : \mathbf{ReceiveEvent} \bullet$
 $(st1 = c1.stateTable \wedge st2 = c2.stateTable \wedge t1 \in st1.transitions \wedge t2 \in st2.transitions$
 $\wedge ((c1 = p.initComp \wedge c2 = p.respComp) \vee (c1 = p.respComp \wedge c2 = p.initComp))$
 $\wedge t1.start = true \wedge t2.start = true \wedge se \in t1.sendEvents \wedge re = t2.receiveEvent \wedge se.event = re.event$
 $\wedge \neg(\exists p2 : \mathbf{Protocol}, c3 : \mathbf{Component}, st3 : \mathbf{StateTable}, t3 : \mathbf{Transition}, re2 : \mathbf{ReceiveEvent} \bullet$
 $p2 \neq p \wedge ((p2.initComp = c3 \wedge p2.respComp = c1) \vee (p2.initComp = c1 \wedge p2.respComp = c3))$
 $\wedge c3 \neq c2 \wedge st3 = c3.stateTable \wedge t3 \in st3.transitions \wedge re2 = t3.receiveEvent \wedge re2.event = se.event)$
 $\wedge \neg(\exists t3 : \mathbf{Transition}, re2 : \mathbf{ReceiveEvent} \bullet t3 \neq t2 \wedge t3 \in st2.transitions \wedge re2 = t3.receiveEvent$
 $\wedge re2.event = re.event))$
 \Rightarrow
 $(\exists newName : \mathbf{String} \bullet se'.convName = newName \wedge re'.convName = newName)$

In many cases this transformation will not be sufficient. As mentioned earlier, the user will have to match up many events that cannot be determined automatically. However, this is not the only problem that may arise after matching up the conversation halves. One problem that may exist is that there may be two events matched as the beginning of a conversation, but the rest of the events are out of order or do not correspond. In this case, an error has been made, either because the definitions of the state tables for the tasks in the analysis phase were incorrect, or because the user decided to match up a SendEvent with a ReceiveEvent that was not really intended to correspond as a message passing between them.

Another problem that may result after the component state tables have been annotated is that the corresponding state tables might have been annotated differently so that the conversation halves do not match. This will be evident when there is a start transition in a component with either a SendEvent or ReceiveEvent, and there is no start transition in the state table of the component that participates in the

protocol with the corresponding event. This will happen when one of the components has coordination with other components or other agents that causes different start and end transitions. In this case, the appropriate start and end labels will need to be added to the state tables so that they match up.

Figure 55 shows one example where two state diagrams have been annotated differently. The dashed arrows show the two events that should match up as the beginning of conversations. However, only the first event will be matched. In the top state diagram, there is no start label for the second transition with the ReceiveEvent even though in the bottom state table the transition with the corresponding SendEvent is already labeled as a start transition. The annotations do not match up is because in the bottom state diagram there are internal events that take place in the middle of the transitions with external events, requiring two conversations instead of one. When the developer determines that second set of events match, a start label is added to the second transition in the top state diagram and a new conversation name is given to the transitions for the new match.

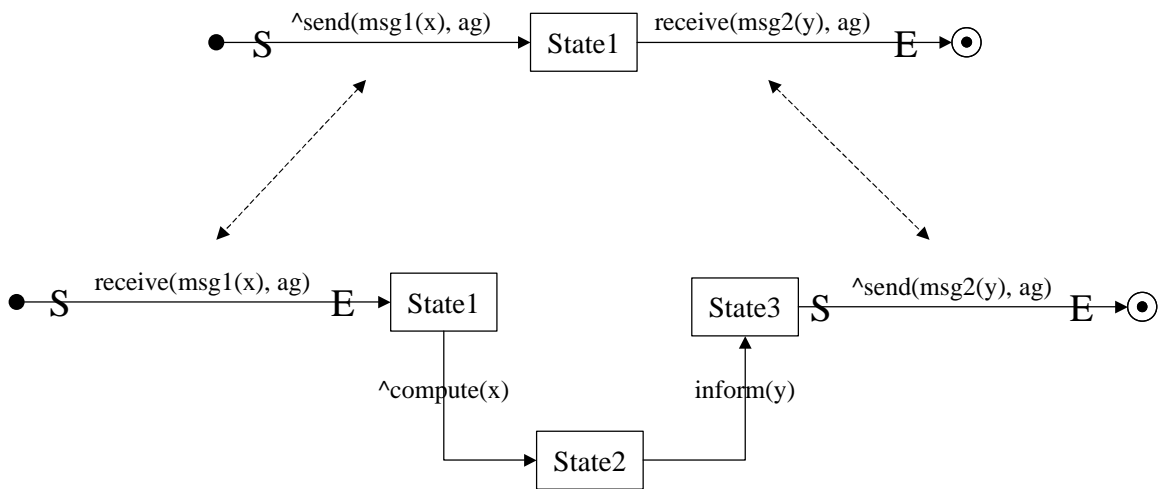


Figure 55 – Two State Diagrams Annotated Differently

Since this last process may involve adding new start labels to transitions, Transformation 27 would now be reapplied so that any needed end labels would also be added to the state tables on transitions in front of the new start transitions.

3.3.6 Splitting Transitions with a ReceiveEvent and Multiple Conversation Names

As conversations are matched, it may be the case that a transition that has a non-empty set of protocols can end up with ReceiveEvent that starts one conversation and a SendEvent that starts another conversation. Transformation 10 only split up transitions that had a ReceiveEvent and at least one SendEvent with different protocols. Consider the example in Figure 56. The sets above the transitions represent the set of protocols and the *convName* for the events. Although both events in the top state diagram belong to the same protocol, they become the first messages in two different conversations because of the way that the bottom state diagram was annotated due to the *internalMsg(x)* event on the transition between State1 and State2.

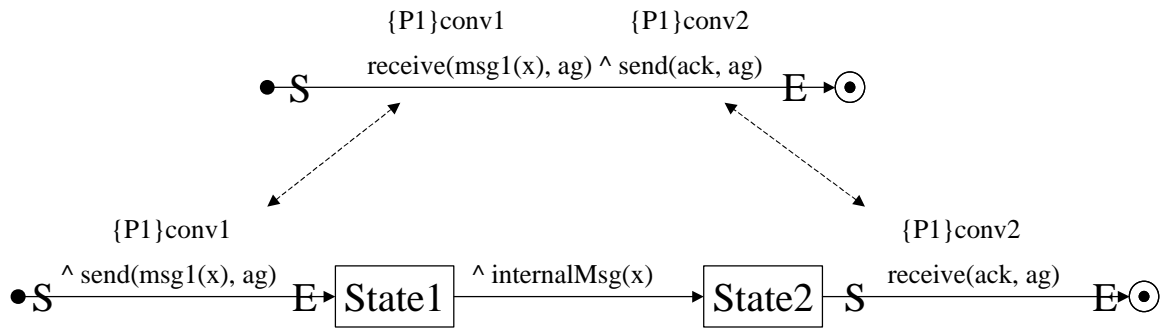


Figure 56 – Transition with a ReceiveEvent and Multiple Conversation Names

Transformation 31 splits transitions that are labeled as the start of a conversation but have a ReceiveEvent that starts one conversation and a SendEvent that starts another conversation. As mentioned, Transformation 10 made sure the only SendEvent on a transition with a ReceiveEvent has the same protocols. Furthermore, since a transition cannot have multiple SendEvents to the same entity (i.e. the protocols are the same), then we can also be certain at this point in the transformation process that there can only be *one* SendEvent on transitions that have a ReceiveEvent. When Transformation 31 splits up the transition, a new null state is created that becomes the *to* state of the original transition, and a new transition is added from the new null state to the original *to* state. The ReceiveEvent is left on the original transition with any guard and actions. The single SendEvent is placed on the new transition, and its set of protocols is the same as the original transition. The original transition is given the end label, and the new transition is

given the start label. Additionally, if the original transition had the end label, so will the new transition with the SendEvent.

Transformation 31

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, re : \mathbf{ReceiveEvent} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge t.start = true \wedge t.protocols \neq \{ \} \wedge t.receiveEvent = re \wedge re \neq null \\
& \wedge (\exists se : \mathbf{SendEvent} \bullet se \in t.sendEvents \wedge se \neq null \wedge se.convName \neq null \wedge se.convName \neq re.convName)) \\
& \Rightarrow \\
& (t'.end = true \wedge t'.sendEvents = \{ \} \\
& \wedge (\exists s : \mathbf{State}, t2 : \mathbf{Transition}, num : \mathbf{String} \bullet s \notin st.states \wedge s \in st'.states \wedge s.name = "Null" + num \\
& \wedge s.actions = [] \wedge t2 \notin st.transitions \wedge t2 \in st'.transitions \wedge t2.receive = null \wedge t2.receiveEvent = null \\
& \wedge t2.guard = null \wedge t2.actions = [] \wedge t2.sends = [] \wedge t2.sendEvents = [se] \wedge t2.protocols = t.protocols \\
& \wedge t2.start = true \wedge t2.from = s \wedge t2.to = t.to \wedge t'.to = s \wedge (t.end = true \Rightarrow t2.end = true) \\
& \wedge \neg(\exists s2 : \mathbf{State} \bullet s2 \in st'.states \wedge s2 \neq s \wedge s2.name = s.name)))
\end{aligned}$$

Continuing with our example from Figure 56, Transformation 31 changes the state diagram to that shown in Figure 57. Breaking up the transition was straightforward, and now the messages for the two conversations are on two different transitions, but still in the same order as that of the original transition.

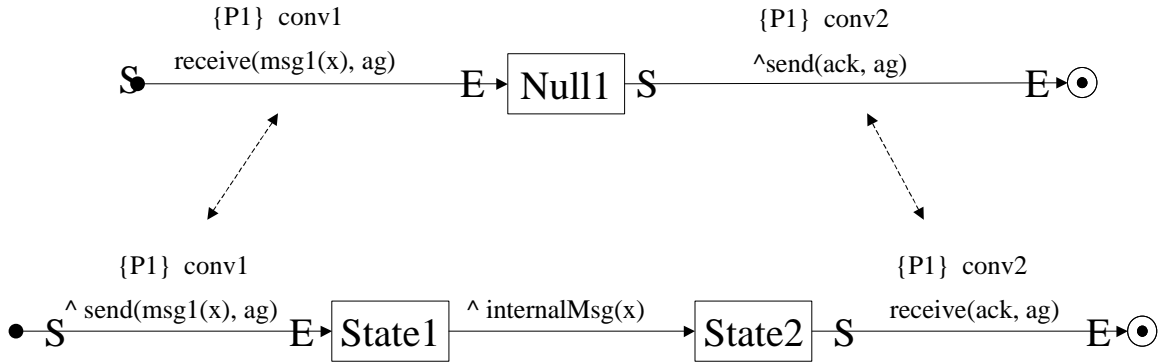


Figure 57 – State Diagrams After Transformation 31

3.3.7 Creating Conversations

Once the different halves of the conversations have been matched up in the component state tables, “empty” conversations can be created based on the conversation names given to the transitions. The conversations will initially be empty because no ConversationHalf objects exist yet that hold the state

tables for the initiator and responder parts of the conversation. The ConversationHalves will be created for the conversations during the third stage of the transformation process, when the conversations are harvested from the components.

As events within component state tables are matched as the beginning of conversations, an event may be matched to several other corresponding events. For every matching pair of events for a given conversation name, a conversation with a unique name is created between the agents of those components. As an example, Figure 58 shows four agents and three conversations between them. The conversations are given unique names because of the compound definition that MaSE uses for conversations². While each half of the conversations must send/receive the same messages in the same order, the state tables do not need to be equivalent. There may be different actions within the states or on the transitions, as well as different states, etc. In other words, the messages sent and received within the components of Agent2 and Agent3 must be the same and in the same order so that they both correspond to the messages within Agent1's component. However, the state tables may still be different and therefore require unique conversations.

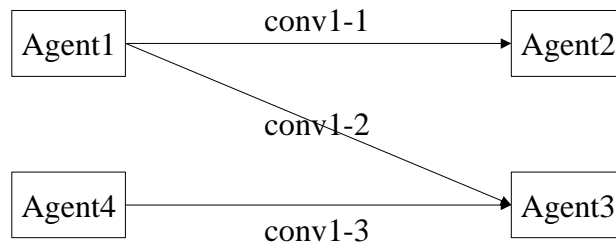


Figure 58 – Duplicate Conversations Between Agents

In order to further illustrate this point, Figure 59 shows the state diagrams for the components of Agent2 and Agent3 annotated for conv1-1 and conv1-2 respectively. The transitions and events in the two state diagrams are identical. However, the actions in the states used to compute y call different functions, so the state diagrams are not equivalent and therefore require different conversations.

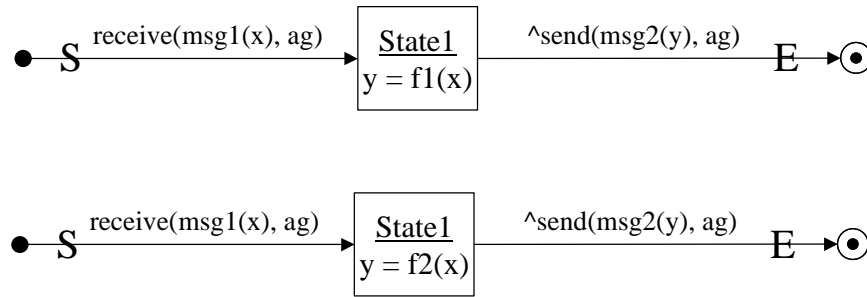


Figure 59 – State Diagrams with Different Actions in State1

3.3.8 Propagating the Set of Conversations

Once all component state tables have been annotated and conversations have been assigned to the start transitions, the set of conversations needs to be propagated to all of the states and transitions belonging to the conversations. Transformation 32 does just that and is intended to be applied iteratively, beginning with all of the start transitions. The transformation no longer needs to be applied when it reaches all transitions that are labeled as the end of a conversation.

Figure 60 continues with an earlier example to demonstrate how Transformation 32 propagates the set of conversations from the start transitions until an end transition is reached. The sets above the transitions represent the set of conversations that the transitions belong to. The S and E labels on the transitions represent the start and end of conversations respectively. In the example, the two transitions leaving the Idle state are the start of two different conversations named conv1 and conv2. The transition leaving State1 receives the set of conversations from the transition into State1, which is {conv1}. The transition leaving State2 likewise receives the set {conv2}. The two conversations merge at the Eval state, and the transitions leaving the Eval state receive {conv1, conv2} as conversations, the union of the two sets.

² In MaSE, conversations are defined by two state tables, one for the initiator and one for the responder. Therefore, the conversations are defined not only by the messages that pass between the agents, but also by the actions that take place in the states and on the transitions to perform the necessary processing.

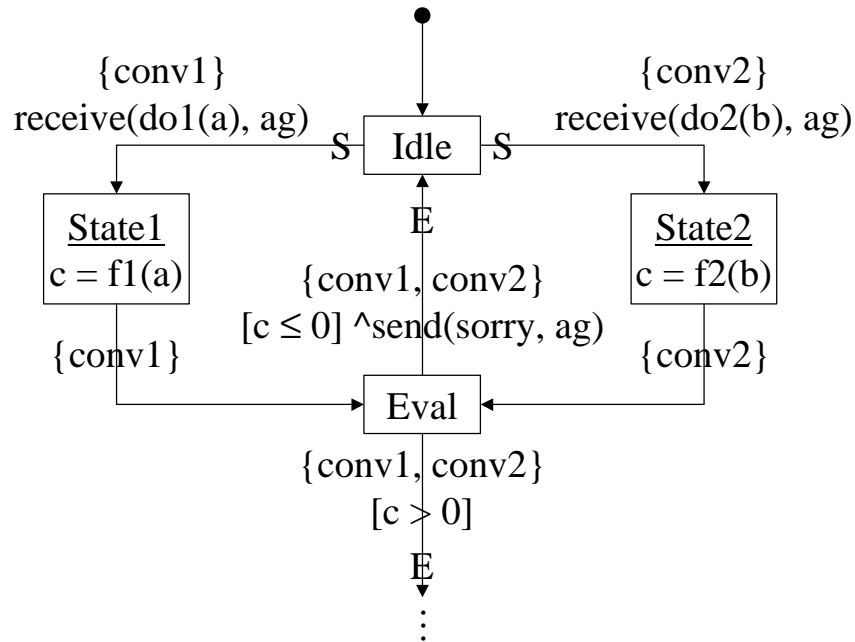


Figure 60 – Example of Propagating the Set of Conversations

Transformation 32

$\forall c : \text{Component}, st : \text{StateTable}, t, t2 : \text{Transition}, s : \text{State} \bullet$
 $(st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge s \in st.states \wedge s = t.to \wedge s = t2.from$
 $\wedge t.end = false)$
 \Rightarrow
 $(s'.conversations = s.conversations ? t.conversations$
 $\wedge t2'.conversations = t2.conversations ? t.conversations)$

3.4 Harvesting the Conversations

Once the component state tables have been fully annotated and the different conversation halves have been matched, the next stage in the transformation process, highlighted in Figure 61, is to first prepare the conversations to be removed and then to actually remove them and replace them with an action on a transition that performs that conversation.

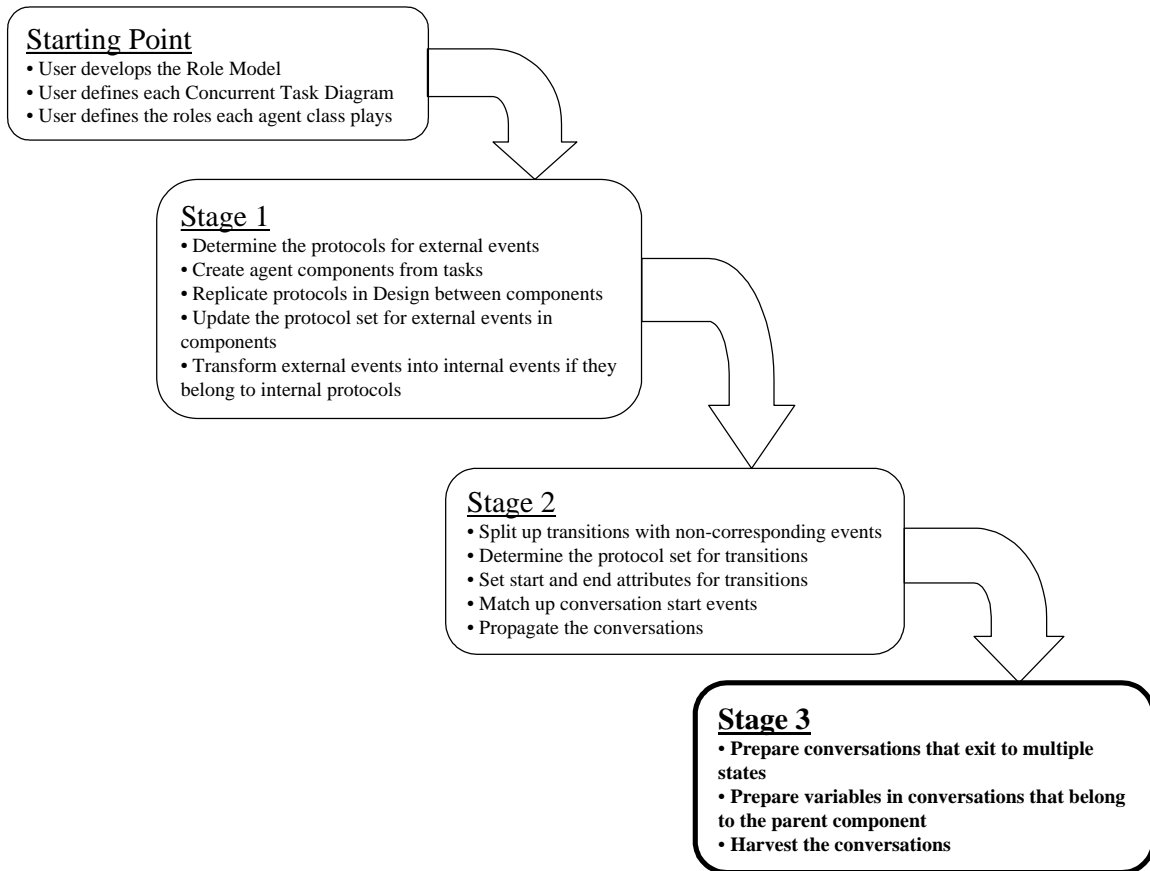


Figure 61 – Stage 3 in the Transformation Process

3.4.1 Combining Conversation End States

The approach for harvesting the conversations from the component state tables is to replace the states and transitions that belong to the conversation with a transition that has an action to perform the conversation. However, if a conversation can end in more than one state, replacing the conversation with a single transition is impossible without first modifying the state table so that the conversation will always exit to a single state. This section describes how this modification is done while preserving the semantics of the model. Before the individual transformations are presented, consider the following example. Figure 62 illustrates a portion of a state diagram annotated as a conversation with multiple states that the end transitions exit to (State 2 and State 3).

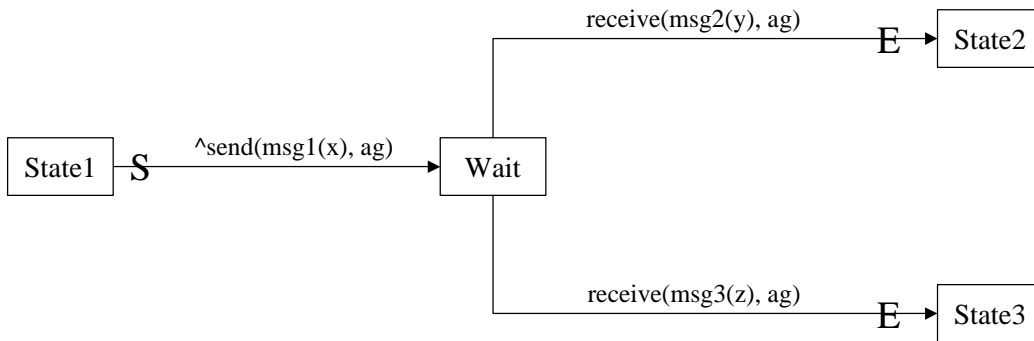


Figure 62 – Conversation with Multiple Exit States

Figure 63 shows the state diagram after the transformations execute. All end transitions now have the same *to* state, which is a newly created null state. Additionally, there is an action on each end transition that sets a BRANCH variable unique to each transition, and for each end transition there is a corresponding transition to the original *to* state with a guard testing the value of the BRANCH variable. The reason “parent.BRANCH” is used in the action will be explained later. This change in the state diagram maintains the semantics of the original state diagram. For example, if the *receive(msg2(y),ag)* ReceiveEvent is received while in the Wait state, the original state diagram will transition to State2. In the state diagram after the transformations, if the same *receive(msg2(y), ag)* is received while in the Wait state, the state diagram will transition to the new Null1 state. However, the BRANCH variable is set to 1 and there is an automatic transition from state Null1 to State2 with a guard condition “BRANCH == 1”.

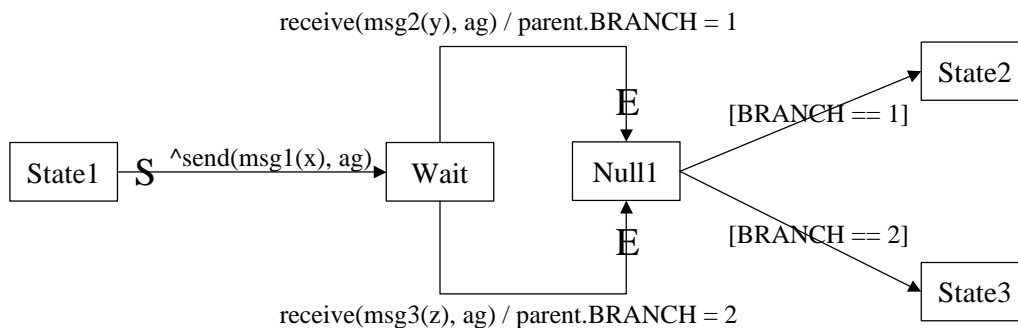


Figure 63 – State Diagram After Transformations

In order to simplify the transformations, this process is broken into three different transformations. If the transitions out of an annotated conversation are to different states, then the first step is to create a new

null state for each exiting transition and set the *to* state to the new null state. Also a new automatic transition is created from the null state to the original *to* state. This change is consistent with the semantics of the original state table. There is nothing new being done and the flow of actions and events remains the same. These modifications are found in Transformation 33. Using the example for this section and beginning with the state diagram in Figure 62, Transformation 33 alters the state table into the form shown in Figure 64

Transformation 33

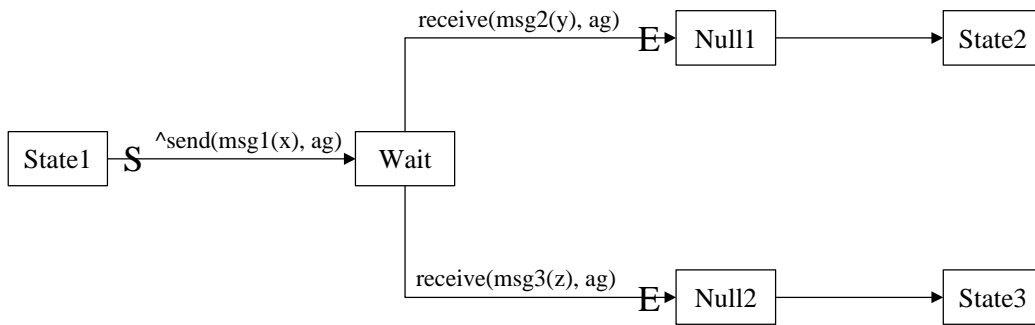
$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, s, s2 : \mathbf{State} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge s \in st.states \wedge t.end = true \\
& \wedge t.to = s \wedge s2 \in st.states \wedge t2.end = true \wedge t2.to = s2 \wedge t2 \neq t \wedge s2 \neq s \\
& \wedge t.conversations \subseteq t2.conversations) \\
& \Rightarrow \\
& (\exists s3 : \mathbf{State}, t3 : \mathbf{Transition}, n : \mathbf{String} \bullet s3 \in st'.states \wedge s3 \notin st.states \wedge s3.name = ("Null" + n) \\
& \wedge s3.conversations = \{ \} \wedge \neg(\exists s4 : \mathbf{States} \bullet s4 \neq s3 \wedge s4 \in st'.states \wedge s4.name = s3.name) \\
& \wedge s3.assignments = \{ \} \wedge t'.to = s3 \wedge t3 \in st'.transitions \wedge t3.from = s3 \wedge t3.to = t.to \\
& \wedge t3.receive = null \wedge t3.receiveEvent = null \wedge t3.guard = null \wedge t3.sends = \{ \} \wedge t3.sendEvents = \{ \} \\
& \wedge t3.actions = [] \wedge t3.protocols = \{ \} \wedge t3.conversations = \{ \} \\
& \wedge \neg(\exists t4 : \mathbf{Transition} \bullet t4 \in st'.transitions \wedge t4.to = s3 \wedge t4 \neq t') \\
& \wedge \neg(\exists t5 : \mathbf{Transition} \bullet t5 \in st'.transitions \wedge t5.from = s3 \wedge t5 \neq t3))
\end{aligned}$$


Figure 64 – State Diagram After Transformation 33

Now that all transitions that end the conversation exit to null state, the null states need to be combined so that the conversation exits to a single state. Before this is done, there must be some way to determine which transition was taken as the conversation completed. Transformation 34 does this by

adding an action to each exiting transition of the form “parent.BRANCH = x”, where x is a unique integer for each transition. The reason “parent.BRANCH” is used in the left hand side of the action is because the variable being set will be checked within the component after the conversation is removed from it. Additionally, for each transition out of the null states the guard condition “BRANCH == x” is added, where x corresponds to the x in the action on the transition into the state. Here, “parent.” does not need to be prepended to the BRANCH variable because this transition will remain in the component’s state table and the BRANCH variable belongs to the component. Continuing with the current example, Transformation 34 would alter the state diagram in Figure 64 into the state diagram shown in Figure 65.

Transformation 34

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, s, s2 : \mathbf{State} \bullet$
 $(st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge s \in st.states \wedge t.end = true \wedge t.to = s$
 $\wedge s2 \in st.states \wedge t2.end = true \wedge t2.to = s2 \wedge t \neq t2 \wedge s2 \neq s \wedge t.conversations \subseteq t2.conversations)$
 \Rightarrow
 $(\exists a : \mathbf{Action}, num : \mathbf{String} \bullet t'.actions = (t.actions \cap a) \wedge a.lhs = [“parent.BRANCH”] \wedge a.rhs = [num]$
 $\wedge \neg(\exists t3 : \mathbf{Transition}, a2 : \mathbf{Action} \bullet t3 \in st'.transitions \wedge t' \neq t3 \wedge a \in t3.actions$
 $\wedge (t.conversations \cap t2.conversations \cap t3.conversations) \neq \{ \})$
 $\wedge (\forall t4 : \mathbf{Transition} \bullet t4 \in st.transitions \wedge t4.from = t.to \Rightarrow t4'.guard = “BRANCH == “ + num))$

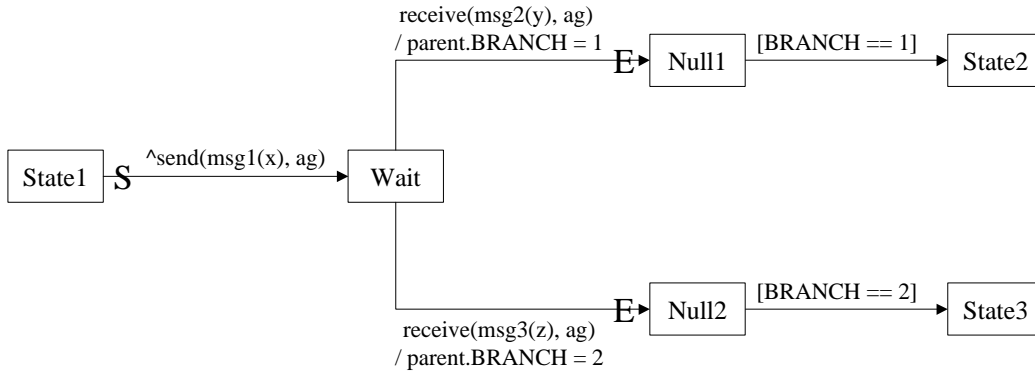


Figure 65 – State Diagram After Transformation 34

Now that each exiting transition has been uniquely labeled with an action and there is a guard condition on the transition out of the null state, Transformation 35 merges all of the null states that the conversation exits to into a single null state, that becomes the *to* state of all of the exiting transitions and the

from state of all transitions out of the null states. The set of null states that the conversation once exited to are removed. In the current example, Transformation 35 changes the state diagram in Figure 65 into its final state, as shown in Figure 66.

Transformation 35

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, s, s2 : \mathbf{State} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge s \in st.states \wedge t.end = true \wedge t.to = s$
 $\wedge s2 \in st.states \wedge t2.end = true \wedge t2.to = s2 \wedge t2 \neq t \wedge s2 \neq s \wedge t.conversations \subseteq t2.conversations)$

\Rightarrow

$(\exists s3 : \mathbf{State}, n : \mathbf{String} \bullet s3 \in st'.states \wedge s3 \notin st.states \wedge s3.name = ("Null" + n) \wedge s3.conversations = \{ \}$
 $\wedge s3.actions = [] \wedge t'.to = s3 \wedge t2'.to = s3 \wedge \neg(\exists s4 : \mathbf{State} \bullet s4 \in st'.states \wedge s4 \neq s3 \wedge s4.name = s3.name)$
 $\wedge (\forall t3 : \mathbf{Transition} \bullet (t3 \in st.transitions \wedge (t3.from = s \vee t3.from = s2)) \Rightarrow t3'.from = s3))$

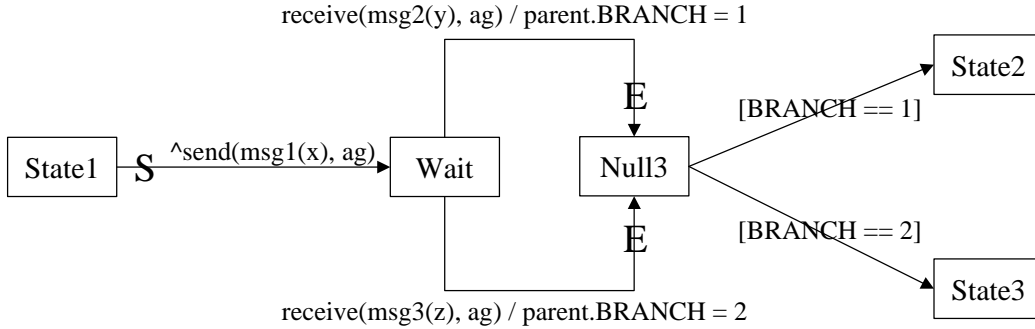


Figure 66 – State Diagram After Transformation 35

3.4.2 Preparing Variables and Parameters

The next step in preparing the state tables for removal of conversations deals with variables and parameters used within a conversation that are also used outside of that conversation. The semantics of variables in a conversation are that they are local to the conversation. Therefore, any variable that is accessed within a conversation and is also used elsewhere in the state table must belong to the “parent” component.

If a transition that belongs to a conversation has a receive event with parameters that are used anywhere else besides locally to the conversation, then there must be an action to set each parameter in the

parent component. Otherwise, the event will be received in the conversation, but the component and the other conversations that belong to the component that also must know about the parameters in the event will not have visibility to it. Transformation 36 covers the case when there is a state that does not belong to the conversation and has an action that uses the parameter, and Transformation 37 takes care of cases where there is another transition that does not belong to the conversation and uses that parameter.

All of the transformations in this section use one of two functions defined in Appendix B. The `usedInAction(Parameter, Action)` function returns true if the parameter is used in the action's left hand side, right hand side, or as a parameter of its right hand side function. The `usedInTransition(Parameter, Transition)` function returns true if the parameter is a parameter of any of the events on the transition, used in the guard condition, or used in an action on the transition.

Transformation 36

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, re : \mathbf{ReceiveEvent}, e : \mathbf{Event}, s : \mathbf{State}, \\
& a : \mathbf{Action}, p : \mathbf{Parameter}, param : \mathbf{String} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge re = t.receiveEvent \wedge e = re.event \wedge p \in e.parameters \\
& \wedge param = p.name \wedge s \in st.states \wedge a \in s.actions \wedge (t.conversations \cap s.conversations = \{\})) \\
& \wedge usedInAction(p, a)) \\
& \Rightarrow \\
& (\exists a2 : \mathbf{Action} \bullet t'.actions = t.actions \cap a2 \wedge a2.lhs = ["parent." + param] \wedge a2.rhs = [param])
\end{aligned}$$

Transformation 37

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, re : \mathbf{ReceiveEvent}, e : \mathbf{Event}, p : \mathbf{Parameter}, \\
& param : \mathbf{String} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge re = t.receiveEvent \wedge e = re.event \wedge p \in e.parameters \\
& \wedge param = p.name \wedge t2 \in st.transitions \wedge (t.conversations \cap s.conversations = \{\})) \\
& \wedge usedInTransition(p, t2)) \\
& \Rightarrow \\
& (\exists a2 : \mathbf{Action} \bullet t'.actions = t.actions \cap a2 \wedge a2.lhs = ["parent." + param] \wedge a2.rhs = [param])
\end{aligned}$$

As an example, consider Figure 67 that shows a state table with an annotated conversation. The start transition for the conversation receives the message $msgI(x)$. There is also another transition in the

state table that is not part of the conversation with an internal event that has x as one of its parameters. Therefore, the parameter x must belong to the component, not just to the conversation, so an action is added to the start transition and the resulting state diagram is shown in Figure 68.

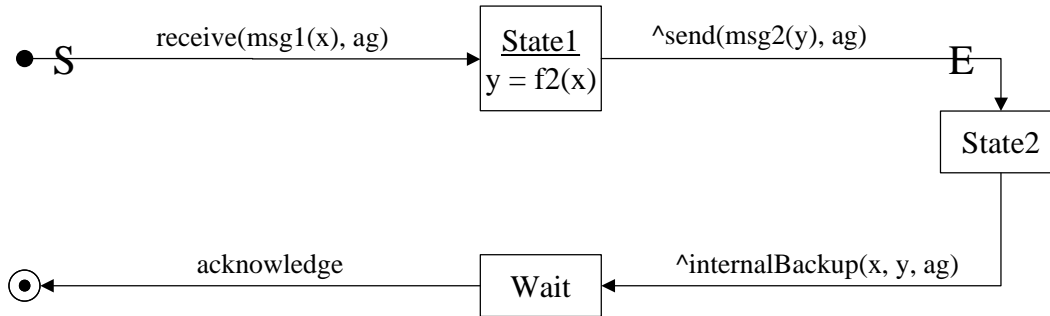


Figure 67 – State Diagram Before Transformation 37

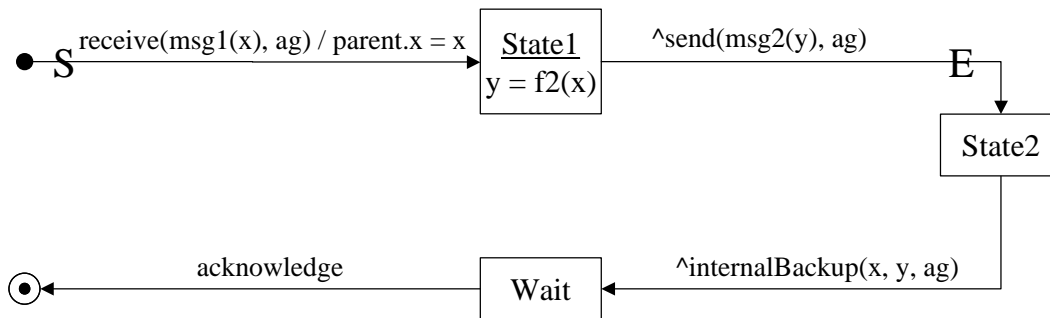


Figure 68 – State Diagram After Transformation 37

In addition to parameters in received events in conversations, if a states that belongs to a conversation has an action that uses a variable and that variable is also used or set anywhere else in the component, then the variable must be prepended with “parent.” to indicate that it is a variable that belongs to the parent component. Transformation 38 makes sure this is done when the variable is used in another state not in the conversation, and Transformation 39 covers the case when the variable is used in a transition that does not belong to the conversation.

Transformation 38

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, s, s2 : \mathbf{State}, a, a2 : \mathbf{Action}, p : \mathbf{Parameter}, param : \mathbf{String} \bullet$
 $(st = c.stateTable \wedge s \in st.states \wedge a \in s.actions \wedge param = p.name \wedge s2 \in st.states \wedge a2 \in s2.actions$
 $\wedge (param \in a.lhs \vee param \in a.rhs \vee p \in a.function.parameters) \wedge (s.conversations \cap s2.conversations = \{\})$
 $\wedge usedInAction(p, a2))$
 \Rightarrow
 $(p'.name = "parent." + param \wedge param' = p.name)$

Transformation 39

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, s : \mathbf{State}, a : \mathbf{Action}, p : \mathbf{Parameter}, param : \mathbf{String}, t : \mathbf{Transition} \bullet$
 $(st = c.stateTable \wedge s \in st.states \wedge a \in s.actions \wedge param = p.name \wedge t \in st.transitions$
 $\wedge (param \in a.lhs \vee param \in a.rhs \vee p \in a.function.parameters) \wedge (s.conversations \cap s2.conversations = \{\})$
 $\wedge usedInTransition(p, t))$
 \Rightarrow
 $(p'.name = "p." + param \wedge param' = p.name)$

Continuing with the previous example, State1 in Figure 68 has an action that computes y based on the parameter x . Since x and y are used as parameters in the *internalBackup*(x, y, ag) event, that variable must belong to the parent component. The resulting state diagram is shown in Figure 69

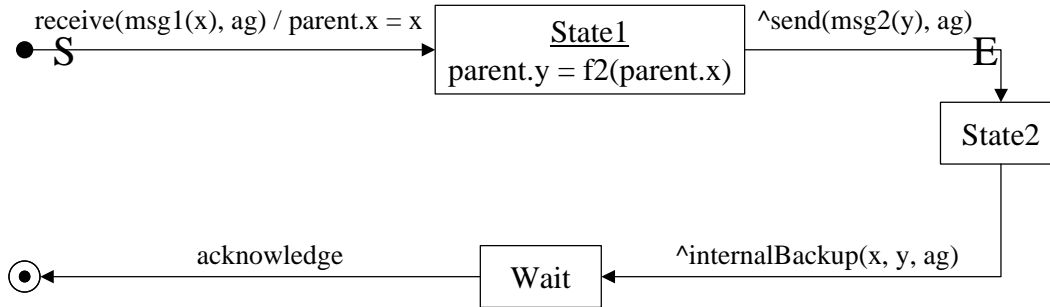


Figure 69 – State Diagram After Transformation 39

The next two transformations are essentially the same as Transformation 38 and Transformation 39, except that the actions with the variables to be prepended with “parent.” are on transitions within a conversation, not states.

Transformation 40

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, s : \mathbf{State}, a, a2 : \mathbf{Action}, p : \mathbf{Parameter}, param : \mathbf{String} \bullet$
($st = c.stateTable \wedge t \in st.transitions \wedge a \in t.actions \wedge param = p.name \wedge s \in st.states \wedge a2 \in s.actions$
 $\wedge (param \in a.lhs \vee param \in a.rhs \vee p \in a.function.parameters) \wedge (t.conversations \neq s.conversations = \{\})$
 $\wedge usedInAction(p, a2)$)
 \Rightarrow
($p'.name = "parent." + param \wedge param' = p.name$)

Transformation 41

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, a : \mathbf{Action}, p : \mathbf{Parameter}, param : \mathbf{String}, t, t2 : \mathbf{Transition} \bullet$
($st = c.stateTable \wedge t \in st.transitions \wedge a \in t.actions \wedge param = p.name \wedge t2 \in st.transitions$
 $\wedge (param \in a.lhs \vee param \in a.rhs \vee p \in a.function.parameters) \wedge (t.conversations \neq s.conversations = \{\})$
 $\wedge usedInTransition(p, t2)$)
 \Rightarrow
($p'.name = "parent." + param \wedge param' = p.name$)

The last condition where special preparations must be made for variables is when there is a transition that belongs to a conversation that has a SendEvent with a parameter that is used outside of the annotated conversation. Transformation 42 covers the case when the parameter is also used in an action within a state that does not belong to the conversation, while Transformation 43 covers the case when the variable is used in another transition that does not belong to the same conversation. The result of the transformations is that the parameter is prepended with “parent.” to indicate that it belongs to the parent component.

Transformation 42

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, se : \mathbf{SendEvent}, e : \mathbf{Event}, s : \mathbf{State}, a : \mathbf{Action},$
 $p : \mathbf{Parameter} \bullet$
($st = c.stateTable \wedge t \in st.transitions \wedge se \in t.sendEvents \wedge e = se.event \wedge p \in e.parameters \wedge s \in st.states$
 $\wedge a \in s.actions \wedge (t.conversations \neq s.conversations = \{\}) \wedge usedInAction(p, a)$)
 \Rightarrow
 $p'.name = "parent." + p.name$

Transformation 43

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, se : \mathbf{SendEvent}, e : \mathbf{Event}, p : \mathbf{Parameter} \cdot$
 $(st = c.stateTable \wedge t \in st.transitions \wedge se \in t.sendEvents \wedge e = se.event \wedge p \in e.parameters$
 $\wedge t2 \in st.transitions \wedge (t.conversations \neq t2.conversations = \{ \}) \wedge usedInTransition(p, t2))$
 \Rightarrow
 $p'.name = "parent." + p.name$

Continuing with our example, the parameter *y* is used in the SendEvent *send(msg2(y), ag)* on the transition leaving State1. However, *y* is also used in the *internalBackup(x, y, ag)* event on a transition that does not belong to the conversation. Transformation 43 changes *y* in the SendEvent to *parent.y* to indicate that it belongs to the parent component. The resulting state diagram is shown in Figure 70.

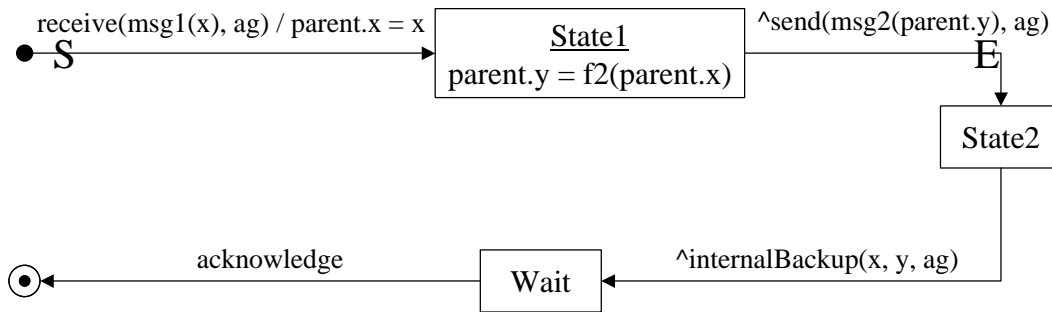


Figure 70 – State Diagram After Transformation 43

3.4.3 Initiator Conversation Halves

In a component state table, the initiator half of a conversation is indicated by a start transition with no ReceiveEvent, but that does have at least one SendEvent. There are two possible cases that must be considered when dealing with the initiator sides of the conversations. The first case is that the transition could have a non-empty set of protocols, which also implies there is a single conversation name. Transformation 44 deals with this case, and creates a transition with a single action that represents the execution of the conversation. The other case is when a start transition has an empty set of protocols. This happens when a transition has SendEvents to different recipients or there is a SendEvent that is a multicast. Transformation 45 handles this case and creates an action on the transition for each conversation that is

indicated by the set of conversation names. The following steps will be taken when performing these transformations:

- A new transition is added to component's state diagram. The transition's *from* state is the start transition's *from* state, and the transition's *to* state is the end transition's *to* state.
- The guard condition from the initial transition of the conversation is added to the transition and removed from the conversation's transition. This is done so that the conversation is only instantiated if the guard condition is true.
- An action is added to the transition for each conversation that is started on the transition. The action instantiates each conversation, and when the conversation completes, the action is done, thus preserving the original semantics of the state table.
- The recipient in the first SendEvent in the conversation is added as the first parameter to the action's function call, and all variables used in the conversation before they are set, as defined by `isNeeded()` in Appendix B, are added as parameters to the action.

Transformation 44

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, s, s2 : \mathbf{State} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge s \in st.states \wedge s2 \in st.states \\
& \wedge t.protocols \neq \{\} \wedge t.start = true \wedge t2.end = true \wedge t.from = s \wedge t2.to = s2 \wedge t.receiveEvent = null \\
& \wedge t.conversations \subseteq t2.conversations \wedge (\exists se : \mathbf{SendEvent} \bullet se \in t.sendEvents \wedge se \neq null)) \\
& \Rightarrow \\
& (\exists! t3 : \mathbf{Transition}, a : \mathbf{Action}, f : \mathbf{FunctionCall}, num : \mathbf{String} \bullet t3 \in st'.transitions \wedge t3 \notin st.transitions \\
& \wedge t3.from = t.from \wedge t3.to = t2.to \wedge t3.guard = t.guard \wedge t'.guard = null \wedge t3.conversations = \{\} \\
& \wedge t3.actions = [a] \wedge a.lhs = null \wedge a.rhs = f \wedge (\#(t.conversations) > 1) \Rightarrow f.name = se.convName + num \\
& \wedge \neg(\exists t4 : \mathbf{Transition}, a2 : \mathbf{Action}, f2 : \mathbf{FunctionCall} \bullet t4 \in st'.transitions \wedge t4 \neq t3 \wedge a2 \in t4.actions \\
& \quad \wedge a2.rhs = f2 \wedge f2.name = f.name) \\
& \wedge (\#(t.conversations) = 1) \Rightarrow f.name = t.conversations[1].name \wedge f.parameters[1] = se.recipient \\
& \wedge (\forall p : \mathbf{Parameter} \bullet isNeeded(p, t.conversations, st) \Rightarrow f'.parameters = f.parameters \cap p))
\end{aligned}$$

In order to more fully describe Transformation 44, consider the state diagram shown in Figure 71. The set above the transitions indicate the conversations to which the transitions belong. The transition from the start state to State1 is indicated as the start of the conversation, and since there is only a SendEvent and no ReceiveEvent, it must be the initiator half of the conversation. Figure 52 shows the state diagram after

Transformation 44 creates the new transition from the start state to the end state. The action was given the name of the conversation because there was a single conversation being started. The variable x is passed as a parameter for the action to perform the conversation because x is sent in a message before its value has been determined by an action in the conversation. Similarly, variables used in an action, either in a state or on a transition, before they have been set within the conversation will also be provided as parameters to the function for that conversation. The exception to this rule is when a variable is used before it is set, but was prepended with “parent.” by one of the earlier transformations. In this case, the variable does not need to be provided when the conversation is instantiated, because the variable is already referencing the parent component.

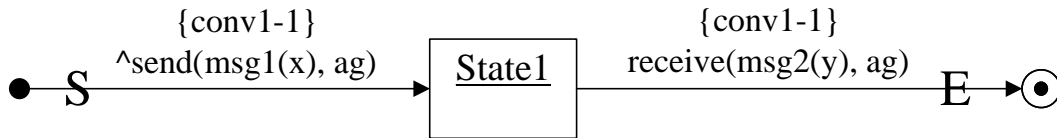


Figure 71 – State Diagram Before Transformation 44

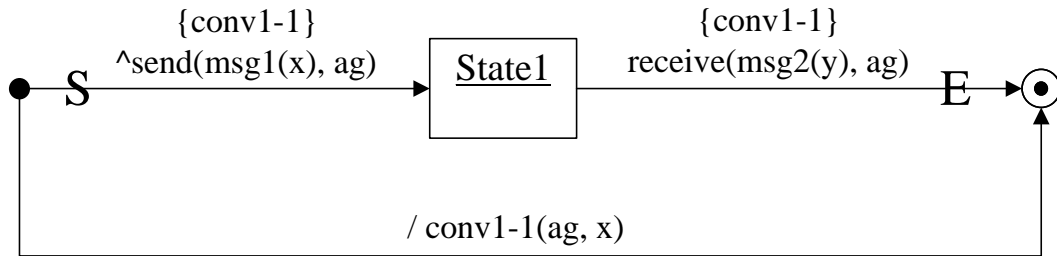


Figure 72 – State Diagram After Transformation 44

The next example demonstrates how Transformation 44 creates a transition and an action if the start transition has a non-empty protocols set, but there are multiple conversations in its conversations set. This means that the SendEvent matched up with multiple ReceiveEvents as the start of the conversation. Figure 74 shows the state diagram after Transformation 44 has added the transition from the start state to the end state with the action named conv1-1_2. There is a single action placed on the transition even though there are two possibilities for which conversation actually takes place when the action is executed. While the responder halves’ state diagrams may not be identical, from the initiator’s point of view the messages they pass are the same so only one action is necessary. The recipient of the first SendEvent (ag)

is supplied as a parameter to the function call and will be the parameter that actually determines which conversation is started.

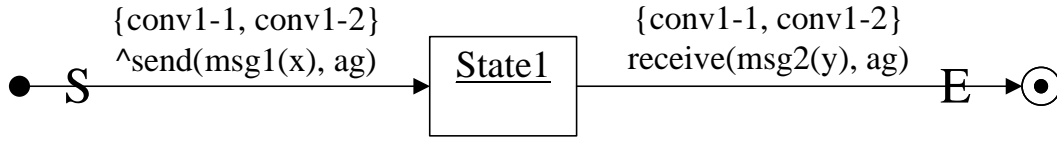


Figure 73 – State Diagram Before Transformation 44

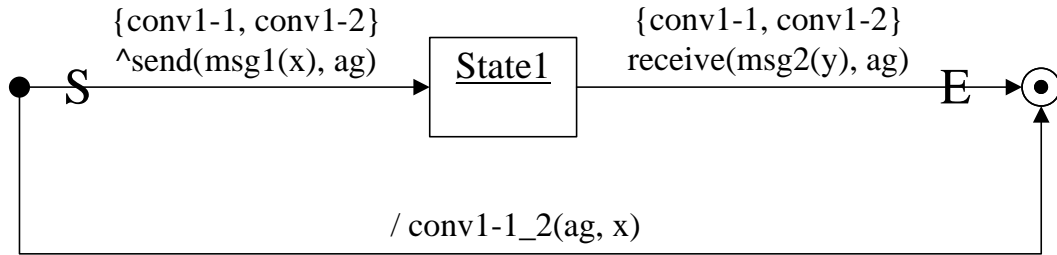


Figure 74 – State Diagram After Transformation 44

Transformation 45

$\forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, s, s2 : \mathbf{State} \bullet$

$(st = c.stateTable \wedge t \in st.transitions \wedge s \in st.states \wedge s2 \in st.states \wedge t.protocols = \{ \} \wedge t.start = true$
 $\wedge t.end = true \wedge t.from = s \wedge t.to = s2 \wedge t.receiveEvent = null \wedge t.sendEvents \neq \{ \})$

\Rightarrow

$(\exists! t3 : \mathbf{Transition} \bullet t3 \in st'.transitions \wedge t3 \notin st.transitions \wedge t3.from = s \wedge t3.to = s2$

$\wedge t3.guard = t.guard \wedge t'.guard = null \wedge t3.conversations = \{ \}$

$\wedge (\forall cname : \mathbf{String} \bullet (cname \in t.convNames$

$\wedge (\exists se : \mathbf{SendEvent} \bullet se \in t.sendEvents \wedge se.convName = cname))$

\Rightarrow

$(\exists a : \mathbf{Action}, f : \mathbf{FunctionCall}, num : \mathbf{String} \bullet t3'.actions = t3.actions \cup a \wedge a.lhs = null \wedge a.rhs = f$

$\wedge (\#(se.conversations) > 1) \Rightarrow f.name = se.convName + num$

$\wedge \neg(\exists t4 : \mathbf{Transition}, a2 : \mathbf{Action}, f2 : \mathbf{FunctionCall} \bullet t4 \in st'.transitions \wedge t4 \neq t3 \wedge a2 \in t4.actions$
 $\wedge a2.rhs = f2 \wedge f2.name = f.name)$

$\wedge (\#(se.conversations) = 1) \Rightarrow f.name = se.conversations[1].name \wedge f.parameters[1] = se.recipient$

$\wedge (\forall p : \mathbf{Parameter} \bullet isNeeded(p, se.conversations, st) \Rightarrow f'.parameters = f.parameters \cup p)))$

Transformation 45 is also be described by way of an example. Figure 75 shows a simple state diagram where there is a transition with SendEvents that start different conversations. The sets above the

transitions are the conversations sets for the SendEvents, not the transitions. The first SendEvent starts conv1-1, while the second SendEvent starts either conv2-1 or conv2-2 based on ag2, the recipient. Figure 76 shows the state diagram after Transformation 45 adds the new transition from the start state to the end state. The new transition has two actions, one named conv1-1 that was added for the first SendEvent on the original transition, and the other named conv2-1_2 that was added for the second SendEvent on the original transition with conversations conv2-1 and conv2-2. Only one action was used for the latter for the same reasons previously described.

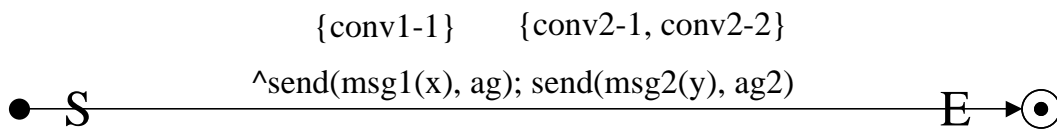


Figure 75 – State Diagram Before Transformation 45

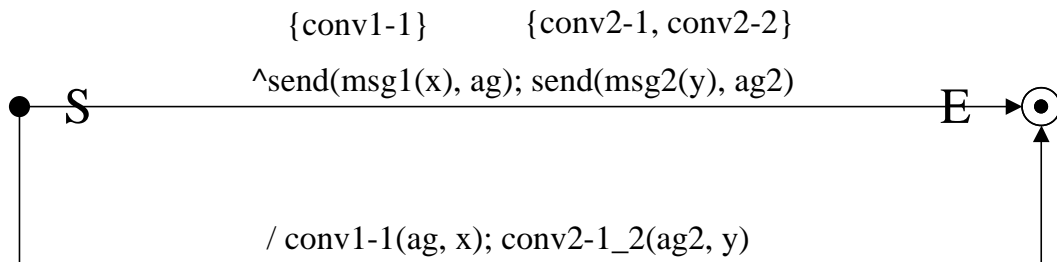


Figure 76 – State Diagram After Transformation 45

3.4.4 Responder Conversation Halves

In a component state table, the responder half of a conversation is indicated by a transition with the start label that also has a ReceiveEvent. For responder conversation halves, Transformation 46 creates a transition and an action to instantiate the conversation as follows:

- A new transition is added to component's state diagram. The transition's *from* state is the *from* state of the transition with the start label. The transition's *to* state is the *to* state of the end transition in the conversation.
- The guard condition from the initial transition of the conversation is added to the transition and removed from the original transition.

- The external ReceiveEvent from the initial transition of the conversation is added to the transition. This means that when the component receives this first message it will know to start the corresponding conversation.
- An action is added to the transition to create the conversation. Again, the conversation ends before the action is finished and the next state is entered.
- All parameters in the conversation that are used somewhere else, as defined by the *isNeeded()* function in Appendix B, are added as parameters to the action.

Transformation 46

$$\begin{aligned}
& \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t, t2 : \mathbf{Transition}, s, s2 : \mathbf{State}, re : \mathbf{ReceiveEvent}, cid : \mathbf{String} \bullet \\
& (st = c.stateTable \wedge t \in st.transitions \wedge t2 \in st.transitions \wedge s \in st.states \wedge s2 \in st.states \\
& \wedge t.conversations \subseteq t2.conversations \wedge t.start = true \wedge t2.end = true \wedge t.from = s \wedge t2.to = s2 \\
& \wedge re = t.receiveEvent \wedge re \neq null) \\
& \Rightarrow \\
& (\exists t3 : \mathbf{Transition}, a : \mathbf{Action}, f : \mathbf{FunctionCall}, num : \mathbf{String} \bullet t3 \in st'.transitions \wedge t3.from = t.from \\
& \wedge t3.to = t2.to \wedge t3.guard = t.guard \wedge t'.guard = null \wedge t3.convIDs = \{ \} \wedge t3.receiveEvent = t.receiveEvent \\
& \wedge t3'.actions = t3.actions \cap a \wedge a.lhs = null \wedge a.rhs = f \\
& \wedge (\#(t.conversations) > 1) \Rightarrow f.name = re.convName + num \\
& \wedge \neg(\exists t4 : \mathbf{Transition}, a2 : \mathbf{Action}, f2 : \mathbf{FunctionCall} \bullet t4 \in st'.transitions \wedge t4 \neq t3 \wedge a2 \in t4.actions \\
& \wedge a2.rhs = f2 \wedge f2.name = f.name) \\
& \wedge (\#(t.conversations) = 1) \Rightarrow f.name = t.conversations[1].name \wedge f.parameters[1] = re.sender \\
& \wedge (\forall p : \mathbf{Parameter} \bullet isNeeded(p, t.conversations, st) \Rightarrow f.parameters = f.parameters \cap p))
\end{aligned}$$

An example is used to more fully explain Transformation 46. Figure 77 show a state diagram that has a transition with a ReceiveEvent that is the start of conversation conv1-1. The sets above the transitions show the set of conversations to which the transitions belong. Figure 78 shows the state diagram after Transformation 46 adds the new transition from the start state to the end state with the original ReceiveEvent and the action to start conv1-1. The ReceiveEvent is added to the transition only to indicate that the external message has arrived. Without the ReceiveEvent on the new transition, there is no trigger that associates the receipt of the message to the transition being activated. The component itself does not handle the message, but instead calls the function that performs the conversation. The conversation will handle the message as the first message in the conversation.

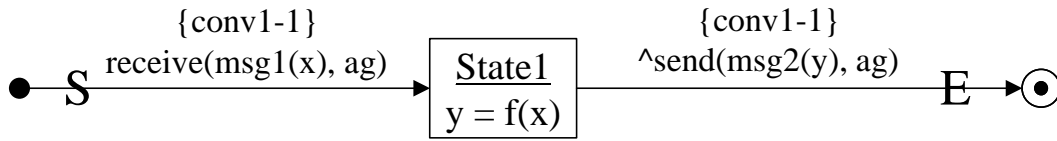


Figure 77 – State Diagram Before Transformation 46

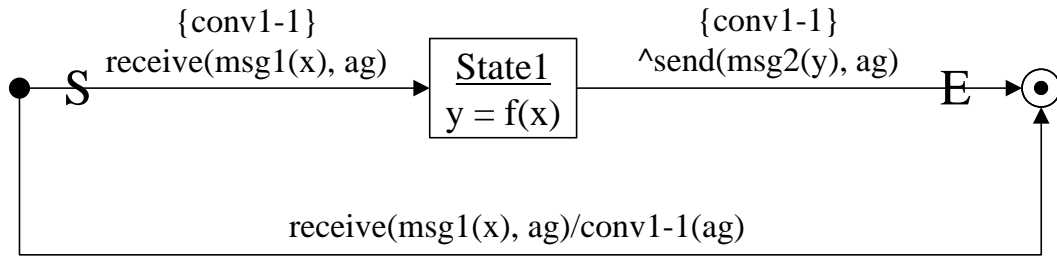


Figure 78 – State Diagram After Transformation 46

Figure 79 shows the same state diagram as in the previous example, but this time the ReceiveEvent has been matched to two different SendEvents and therefore there are two conversations (conv1-1 and conv1-2) that may be started by receiving the $msg1(x)$ message from the start state. As in the case with the initiator conversations, a single transition and single action are used because from this agent's point of view the conversations are the same. Figure 80 shows that state table after Transformation 46.

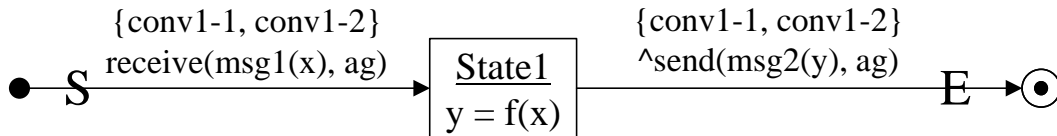


Figure 79 – State Diagram Before Transformation 46

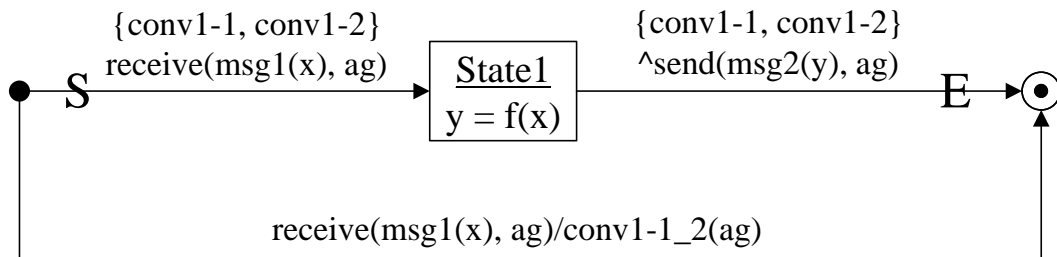


Figure 80 – State Diagram After Transformation 46

3.4.5 Moving States and Transitions From Components to Conversations

At this point every annotated conversation in the component state tables has a transition with an action to replace the states and transitions of the conversations. Transformation 47 creates the ConversationHalves that contain the state tables for the initiator and responder halves of the conversations. The states and transitions that belong to those ConversationHalves will then be removed from the component state tables and added to the ConversationHalf state tables by Transformation 48 through Transformation 50.

Transformation 47

$$\begin{aligned}
 & \forall c : \mathbf{Component}, st : \mathbf{StateTable}, t : \mathbf{Transition}, conv : \mathbf{Conversation} \bullet \\
 & (st = c.stateTable \wedge t \in st.transitions \wedge t.start = true \wedge conv \in t.conversations) \\
 & \Rightarrow \\
 & (\exists ch : \mathbf{ConversationHalf} \bullet ch \in c'.convs \wedge ch.convID = conv.name \\
 & \wedge ((t.receiveEvent = null) \Rightarrow conv.initiator = ch) \wedge ((t.receiveEvent \neq null) \Rightarrow conv.responder = ch))
 \end{aligned}$$

Transformation 48 duplicates states from component state tables for every conversation half to which they belong. Since states and transitions can belong to more than one conversation, as they are added to the conversation half's state table the conversation is removed from its set of conversations. Only when that set is empty (i.e., when it has been added to the state tables of all necessary conversation halves) are the state or transition removed from the component state table.

Transformation 48

$$\begin{aligned}
 & \forall c : \mathbf{Component}, conv : \mathbf{Conversation}, ch : \mathbf{ConversationHalf}, st, st2 : \mathbf{StateTable}, s : \mathbf{State} \bullet \\
 & (st = c.stateTable \wedge st2 = ch.stateTable \wedge ch \in c.convs \wedge s \in st.states \\
 & \wedge (ch = conv.initiator \vee ch = conv.responder) \wedge conv \in s.conversations) \\
 & \Rightarrow \\
 & (s \in st2'.states \wedge conv \notin s'.conversations \wedge ((\#(s'.conversations) = 0) \Rightarrow s' \notin st'.states))
 \end{aligned}$$

When a transition has SendEvents that belong to different conversations it is handled as a special case. Transformation 49 states that if the transition has SendEvents that have different conversations, then

a transition with only the SendEvents that belong to that conversation half will be added to the conversation half's state table. If there are no SendEvents with different conversations, then Transformation 50 adds the entire transition to the conversation half's state table.

Transformation 49

$$\begin{aligned}
& \forall c : \mathbf{Component}, \text{conv} : \mathbf{Conversation}, \text{ch} : \mathbf{ConversationHalf}, \text{st}, \text{st2} : \mathbf{StateTable}, t : \mathbf{Transition} \bullet \\
& (\text{st} = \text{c.stateTable} \wedge \text{st2} = \text{ch.stateTable} \wedge \text{ch} \in \text{c.convs} \wedge t \in \text{c.transitions} \\
& \wedge (\text{ch} = \text{conv.initiator} \vee \text{ch} = \text{conv.responder}) \wedge \text{conv} \in \text{t.conversations} \wedge \text{t.protocols} = \{\}) \\
& \Rightarrow \\
& (\exists t2 : \mathbf{Transition} \bullet t2 \in \text{st2'.transitions} \wedge t2.\text{guard} = \text{null} \wedge t2.\text{receive} = \text{null} \wedge t2.\text{receiveEvent} = \text{null} \\
& \wedge t2.\text{sends} = \{\} \wedge t2.\text{actions} = [] \wedge t2.\text{start} = \text{true} \wedge t2.\text{end} = \text{true} \\
& \wedge (\forall \text{se} : \mathbf{SendEvent} \bullet (\text{se} \in \text{t.sendEvents} \wedge \text{conv} \in \text{se3.conversations}) \Rightarrow \text{se} \in \text{t2.sendEvents}) \\
& \wedge \text{conv} \notin \text{t'.conversations} \wedge ((\#(\text{t'.conversations})) = 0) \Rightarrow \text{t}' \notin \text{st'.transitions})
\end{aligned}$$

Transformation 50

$$\begin{aligned}
& \forall c : \mathbf{Component}, \text{conv} : \mathbf{Conversation}, \text{ch} : \mathbf{ConversationHalf}, \text{st}, \text{st2} : \mathbf{StateTable}, t : \mathbf{Transition} \bullet \\
& (\text{st} = \text{c.stateTable} \wedge \text{st2} = \text{ch.stateTable} \wedge \text{ch} \in \text{c.convs} \wedge t \in \text{st.transitions} \\
& \wedge (\text{ch} = \text{conv.initiator} \vee \text{ch} = \text{conv.responder}) \wedge \text{conv} \in \text{t.conversations} \wedge \text{t.protocols} \neq \{\}) \\
& \Rightarrow \\
& (t \in \text{st2'.transitions} \wedge \text{cid} \notin \text{t'.conversations} \wedge ((\#(\text{t'.conversations})) = 0) \Rightarrow \text{t}' \notin \text{st'.transitions})
\end{aligned}$$

As the transitions are added to the state tables of the conversation halves, the events on the transitions are either ReceiveEvents or SendEvents. However, events in the Communication Class Diagrams that make up the conversations use events in the *receive* and *sends* clauses. Therefore, Transformation 51 changes any ReceiveEvent into an Event in the *receive* clause, and Transformation 52 changes SendEvents into Events in the *sends* clause.

Transformation 51

\forall ch : **ConversationHalf**, st : **StateTable**, t : **Transition**, re : **ReceiveEvent** •

(st = ch.stateTable \wedge t \in st.transitions \wedge re = t.receiveEvent \wedge re \neq null)

\Rightarrow

(t'.receive = re.event \wedge t'.receiveEvent = null)

Transformation 52

\forall ch : **ConversationHalf**, st : **StateTable**, t : **Transition**, se : **SendEvent** •

(st = ch.stateTable \wedge t \in st.transitions \wedge se \in t.sendEvents \wedge se \neq null)

\Rightarrow

(se.event \in t'.sends \wedge se \notin t'.sendEvents)

The last step in the transformation process is to add the start and end states to the state tables of the conversation halves. These are simple transformations. Whenever a transition is a start transition, then Transformation 53 creates a start state that is that transition's *from* state. Likewise, if a transition is has the end label, then Transformation 54 creates an end state that is that transition's *to* state.

Transformation 53

\forall ch : **ConversationHalf**, st : **StateTable**, t : **Transition** •

(st = ch.stateTable \wedge t \in st.transitions \wedge t.start = true)

\Rightarrow

(\exists s : **State** • s.name = "start" \wedge s \in st'.states \wedge t'.from = s)

Transformation 54

\forall ch : **ConversationHalf**, st : **StateTable**, t : **Transition** •

(st = ch.stateTable \wedge t \in st.transitions \wedge t.end = true)

\Rightarrow

(\exists s : **State** • s.name = "end" \wedge s \in st'.states \wedge t'.to = s)

3.5 Summary

This chapter used formal predicate logic equations to present the transformations that generate the MaSE design models from the analysis models. The transformation system was broken down into a three-stage process. The first stage created the agent components from the concurrent tasks based on the roles given to the agents in the Agent Class Diagram. Other activities in this stage include determining protocols for external events in the Concurrent Task Diagrams, replicating protocols in the design, and transforming external events into internal events in the components if the designer determines the protocol they belong to is internal. The second stage annotated component state diagrams for the start and end of conversations and matching the events in the different components that start the conversations. The last stage added the states and transitions from the components to their appropriate conversation halves, removing them from the components and replacing them with a transition and an action that performed the conversation. Chapter IV describes how the transformation system was demonstrated by implementing them in AFIT's agentTool.

IV. Demonstration

Chapter III used predicate logic equations to define a formal transformation system that creates the MaSE design models based on the analysis models. This chapter outlines how the transformations were implemented and integrated with AFIT's agentTool multiagent development environment. Section 4.1 provides an overview of the three-stage transformation process. Section 4.2 details how the transformations were implemented in agentTool. Finally, Section 4.3 steps through an example and illustrates how the transformations incrementally create the agent components and conversations from the Role Model and Concurrent Task Diagrams.

4.1 Transformation System Overview

Chapter III described, in detail, how the transformation system can be thought of as the three-stage process shown in Figure 81. Before the transformations can take place, the developer must analyze the system and develop a Role Model, which defines the roles that are present in the system, and a set of concurrent tasks, which the roles perform to accomplish their goals. The developer must also decide which agent classes will be in the system and the roles that each agent class will play. During the first stage of the transformation process, the components for the agent classes are created based on the roles assigned by the developer. The set of protocols for each external event is also determined. The second stage centers around annotating the component state diagrams and matching external events in the different components that become the initial messages of a conversation. During the last stage of the transformation process the component state diagrams are prepared for the removal of the states and transitions that belong to conversations. They are then removed and added to the state diagrams of the corresponding conversation halves. As they are removed from the components they are replaced with a single transition that has an action that starts the conversation.

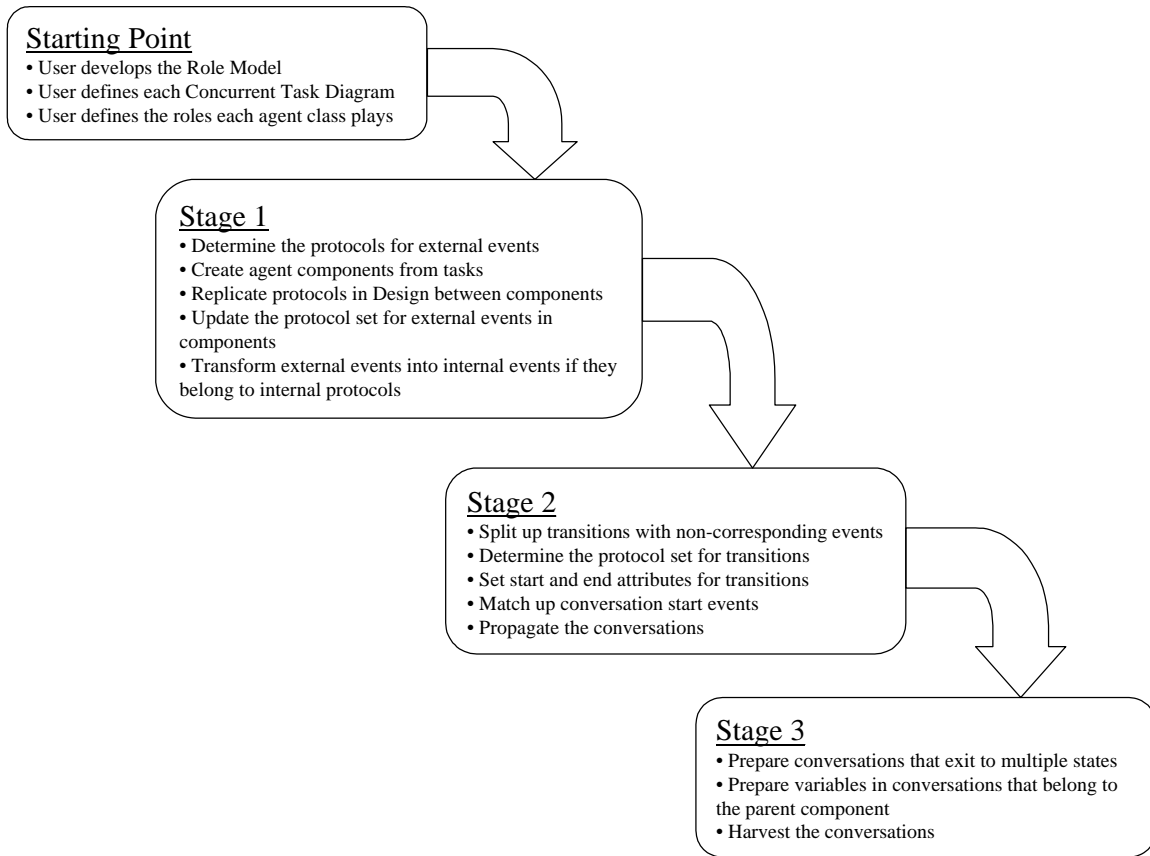


Figure 81 – Three Stages of the Transformation Process

4.2 Integration with agentTool

In order to demonstrate the transformations defined in Chapter III, a transformation system was implemented as part of the agentTool development environment using the Java programming language. The implementation maintained the three-stage approach. Figure 82 shows the menu that was added to agentTool’s menu bar. The menu item *Add Agent Components* corresponds to the first stage of the transformation process, *Annotate Component State Diagrams* corresponds to the second stage, and *Create Conversations* corresponds to the third stage. If the developer selects *Create Conversations* from the menu without having previously selected the first two, then they are done automatically before the conversations are created. As previously stated, before the transformations can take place the Role Model must already exist and there must be at least one agent class that plays each role.

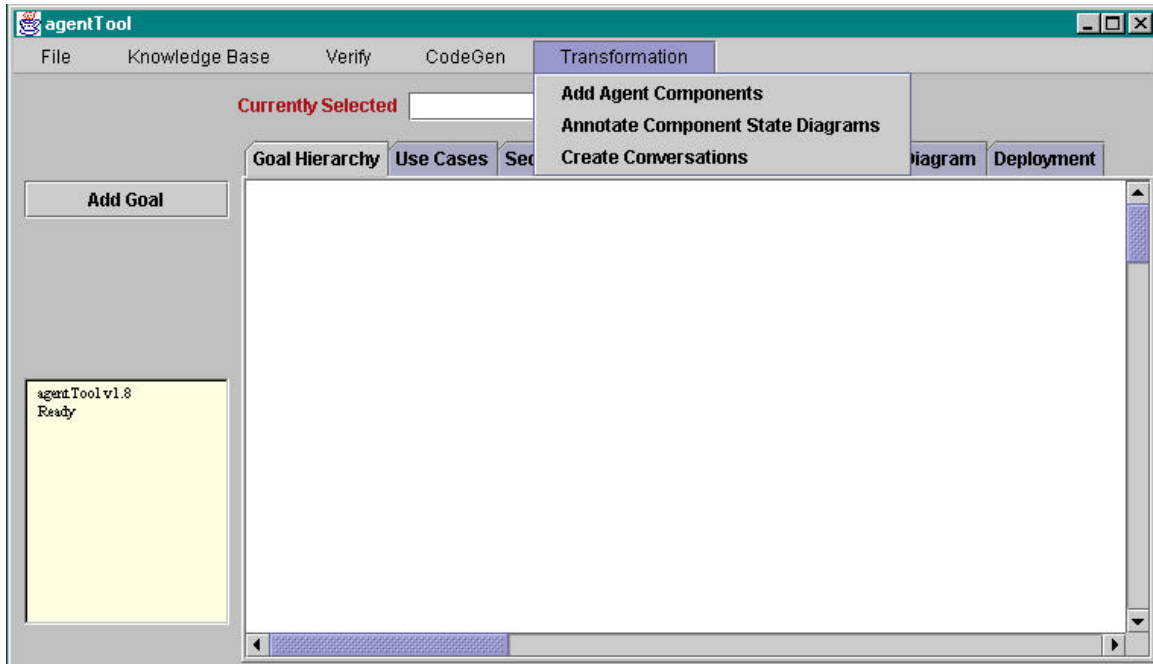


Figure 82 – Transformation Menu in agentTool

4.2.1 Transformation Classes

To implement the transformations, Java classes were defined for the transformations. In most instances each transformation class represents a single transformation from Chapter III. However, there were times when it was possible to combine several transformations into a single class. For example, Transformation 19 through Transformation 24 all add start labels to transitions. A class named *Transform19* was created that represented all of these transformations. When a transformation class is instantiated, the constructor calls its *execute* method, which is where the transformation is actually performed.

When the user makes a selection from the transformation menu, a class is instantiated that represents that stage of the transformation process, which in turn creates instances of the transformation classes that execute during that stage. For example, when the user selects *Create Components* from the transformation menu, a class named *Stage1* is instantiated. Upon creation, the *Stage1* object instantiates, in order, the classes implementing Transformation 1 through Transformation 9.

The transformations are formally defined in Chapter III using universal and existential quantification. In most cases, a loop is used to implement universal quantification over a variable, and a method call is used to implement existential quantification. Therefore, the transformations that use universal quantification over several variables have several nested loops that drill down through the tree to test each combination of the variables. An alternate approach would have been to use a visitor pattern [14] to walk the tree, but implementing the transformations would have been more difficult and harder to understand.

4.2.2 Model Classes

The architectural structure of agentTool already had classes for roles, tasks, state tables, etc. These are referred to as the *ATsystem* classes. However, a new package was created with Java classes for each of the types defined in Chapter II and used in the transformations in Chapter III. This was done for two reasons. First, this made implementing the transformations straightforward. Many of the transformations are non-trivial and translating the formal representations into code was much easier using classes that had the same names and attributes. Secondly, the *ATsystem* classes were created only to hold the information needed to visually represent the models and did not have the required granularity of detail required to perform the transformations. For example, the transmissions on a transition are represented by a single string. They do not distinguish between different events or whether the events are external or internal, much less the parameters of the events. Creating the new classes in a separate package kept the transformations loosely coupled to the existing architectural design for agentTool, whereas altering the *ATsystem* classes would have risked injecting errors into the existing code. Each new class created for the transformations held a pointer to the corresponding *ATsystem* class, and made updates to it whenever necessary.

4.3 Example

This section steps through an example to demonstrate how the transformation system implemented in agentTool creates agent components and conversations from the Role Model and the Concurrent Task

Diagrams. The example does not demonstrate every possible situation that may arise, but demonstrates many of the most common situations encountered while transforming a well-analyzed multiagent system. This section also describes the mechanism for prompting the user for design decisions necessary to complete the transformation process.

4.3.1 Starting Point – Role Model and Initial Agent Classes

The Role Model for a multiagent system is shown in Figure 83, which is the starting point for the transformation process. The Role Model is fairly simple, with only three roles, each with a single task. The Manager role is responsible for bidding out certain search tasks using the ContractNet protocol. The Bidder role is responsible for bidding on the different tasks and then requesting a search from the Searcher role via the SearchRequest protocol. The Searcher role uses mobility to search for the request from the Bidder.

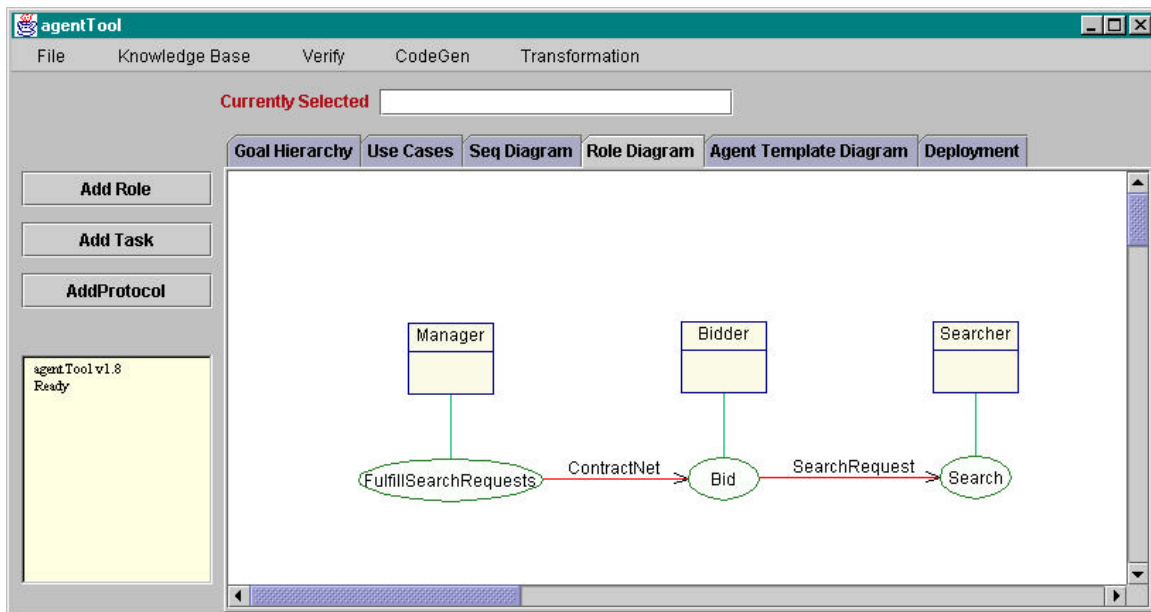


Figure 83 – Role Model

Figure 84 shows the state diagram that represents the Manager role's FulfillSearchRequests task. The task is basically the initiator half of the Contract Net protocol. When there is a task to bid out, a multicast announcement is sent to the list of bidders. The manager then accepts bids until a set time has

expired. The manager determines the winner, sending that agent a message to start the task. Every other role in the list is sent a *sorry* message. The manager then waits for the results from the bidder, displaying them when they are received.

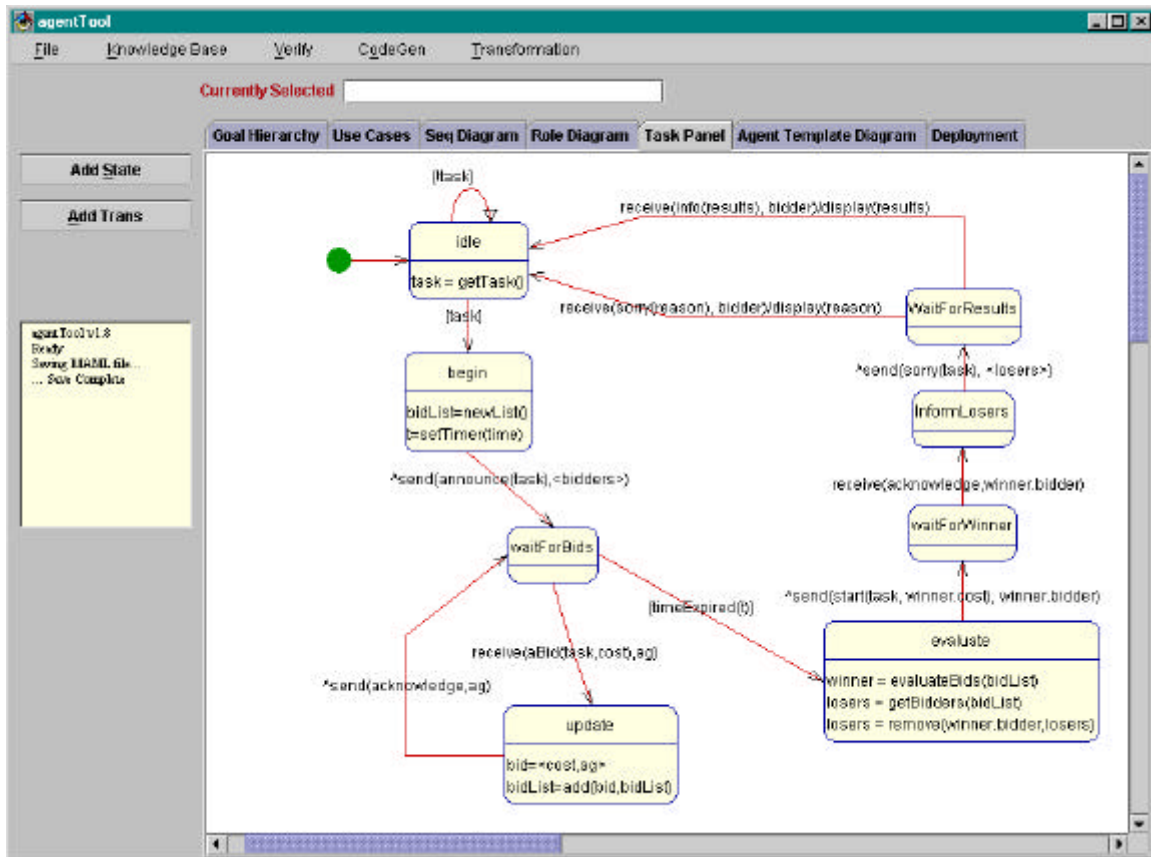


Figure 84 – FulfillSearchRequest Task for the Manager Role

The Bid task for the Bidder role is shown in Figure 85. This task is started automatically and enters an idle state until it receives announcement for a new task. The bidder then determines if it should submit a bid on the task, sending the bid if it is acceptable. Once the bid has been placed, the bidder waits for a message from the manager that indicates if it won the task or not. If it did not, then it transitions back to the idle state. However, if it receives a *start* message, then it sends a message to the searcher to do the search. When it receives the results from the searcher, they are forwarded back to the manager that requested them.

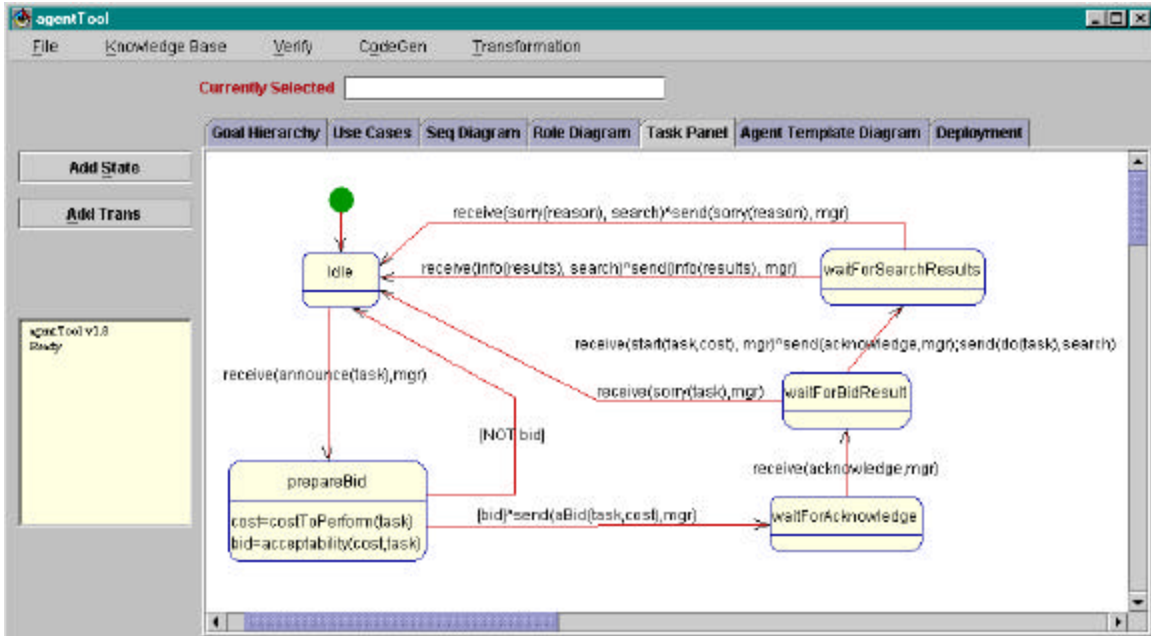


Figure 85 – Bid Task for the Bidder Role

Figure 86 defines the Search task for the Searcher role. The task is started upon receipt of a *do(task)* message from a bidder. The searcher then determines if it needs to move in order to do the task. If it does, then it attempts to move. If the move fails, then it sends a *sorry* message to the bidder. However, if the move is successful or the searcher doesn't need to move, then it searches based on the given task. The searcher then sends the results back to the bidder if there are any, or sends a *sorry* message if there are none.

Another important requirement for the transformation process to begin is that the developer must determine the initial agent classes and the roles they play. For this example, one SearchManager agent class plays the Manager role and the MobileSearcher agent class plays both the Bidder and Searcher roles. This will be important as the agent components are created during the first stage of the transformation process. The initial Agent Class Diagram is shown in Figure 87.

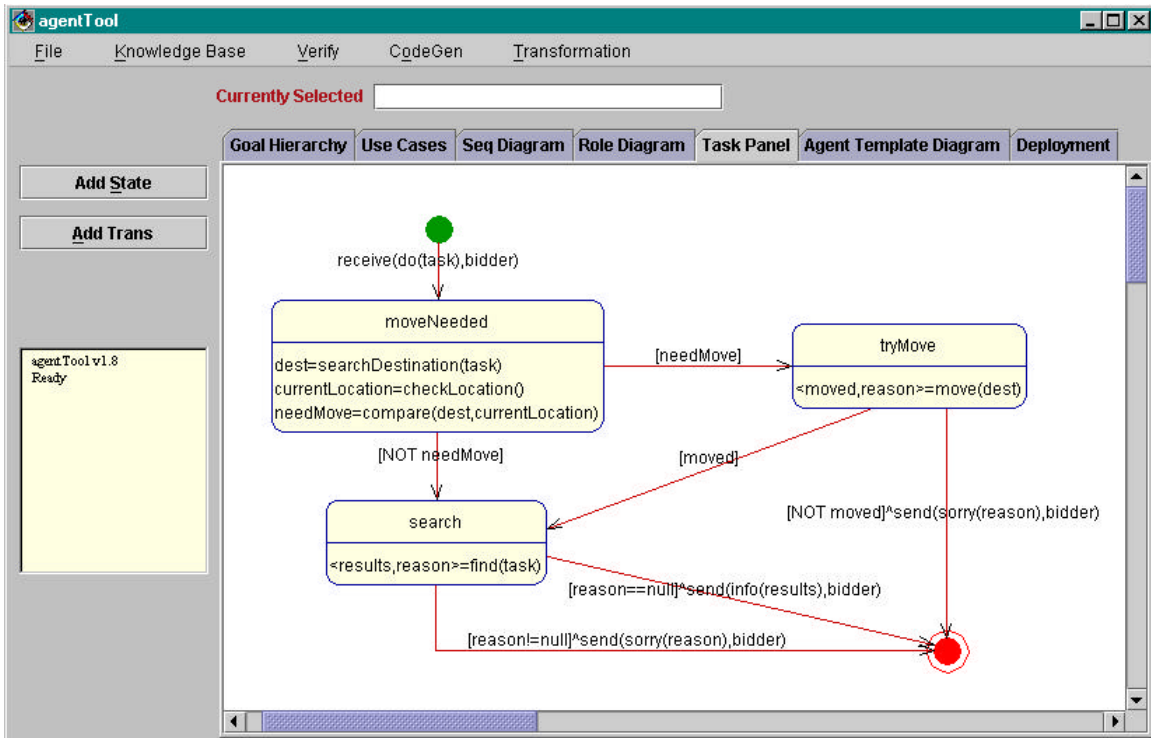


Figure 86 – Search Task for the Searcher Role

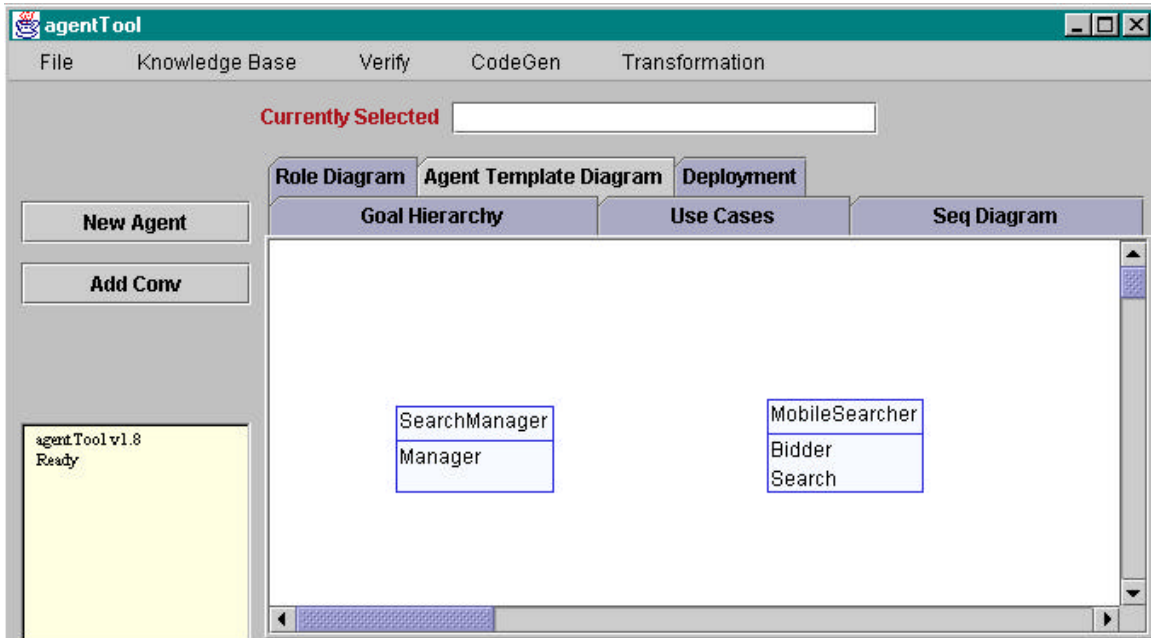


Figure 87 – Initial Agent Class Diagram

4.3.2 Stage One – Creating Agent Components

Now that the Role Model and the initial Agent Class Diagram have been defined, the first stage of the transformation process can begin. This stage will determine the protocols for external events, create agent components based on the roles they play, and allow the user to determine the mode of some protocols in the design.

4.3.2.1 Determining the Protocols for External Events

The first transformations in stage one try to identify the protocols for the external events. In most cases, this process requires no input from the user. However, in some instances it is impossible to automatically determine in which protocols the events were meant to belong. Before the developer is asked to make any decisions about protocols for events, the dialog in Figure 88 is displayed as information on what is about to happen next.

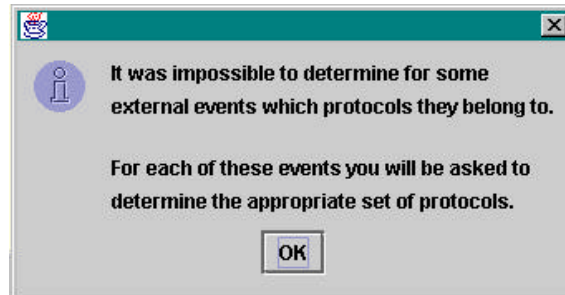


Figure 88 – Ambiguous Protocols Dialog

In the example, there are two events for which the transformations could not automatically determine the protocols. The Bid task for the Bidder role receives two different external *sorry* events, one from the Manager role and another from the Searcher role. The developer is asked to make the decision for the events one at a time. As shown in Figure 89, the transition with the event in question is highlighted in the Bid task and another window is displayed for the developer to select the protocols for that event. The first external event presented to the developer is the *receive(sorry(task), mgr)* event. This event belongs to the ContractNet protocol between the Manager role and the Bidder role, so that protocol is chosen.

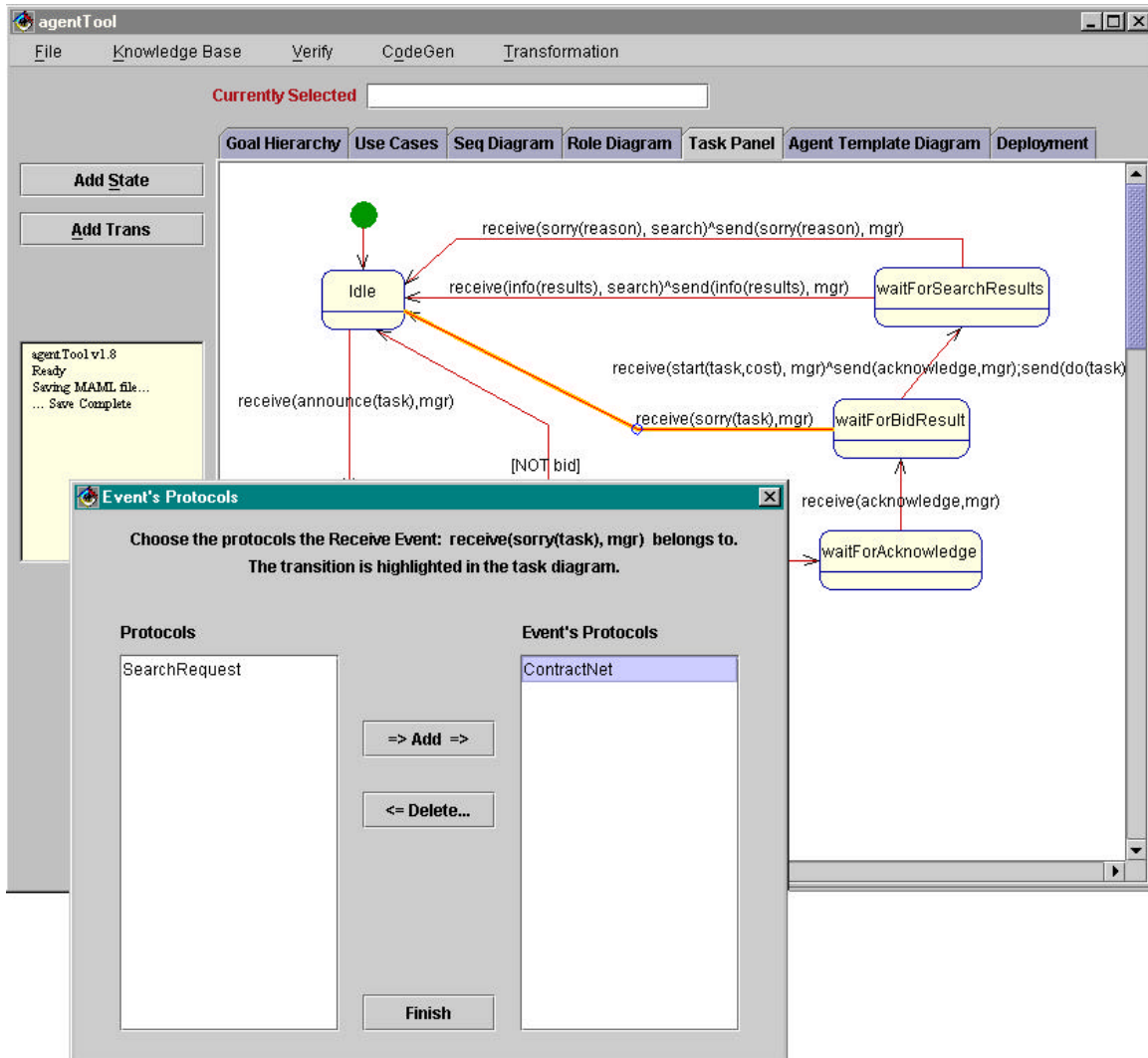


Figure 89 – First Protocol Decision

The next external event for which the developer must determine the protocols is the `receive(sorry(reason), search)` message, shown in Figure 90. Since that event belongs to the SearchRequest protocol between the Bidder and Searcher roles, that is the protocol selected.

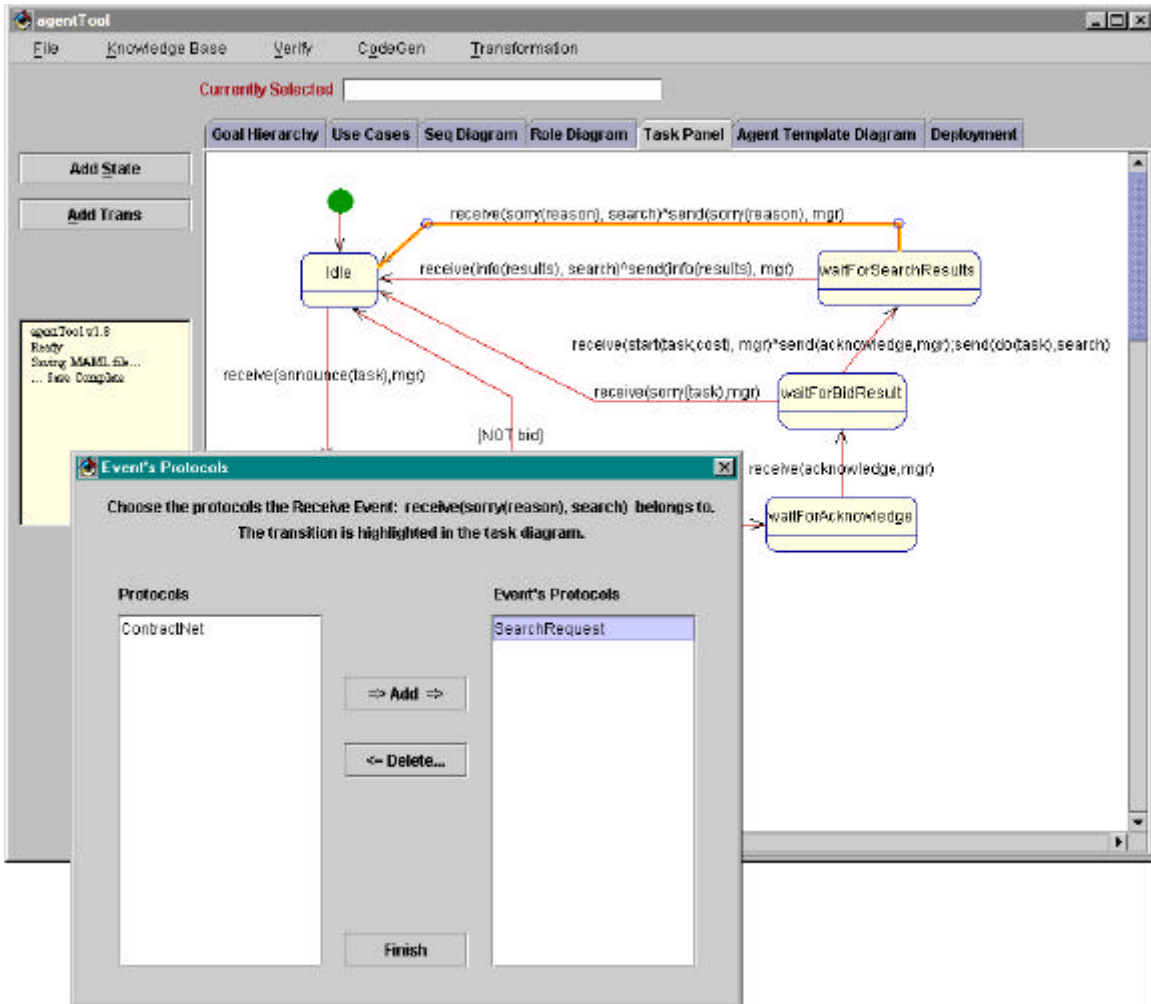


Figure 90 – Second Protocol Decision

4.3.2.2 Determining the Mode for the SearchRequest Protocol

Since the developer determined that a single agent class could play both the Bidder and Searcher roles, the developer must decide if the protocols between tasks of those roles are still external, or if they are now meant to be internal communication. The SearchRequest protocol is the only protocol that falls into this category, and is meant to be internal communication. When the dialog shown in Figure 91 is displayed, the “Internal” button is chosen, and every event that belongs to the SearchRequest protocol in the Bid and Search components are changed into internal events.

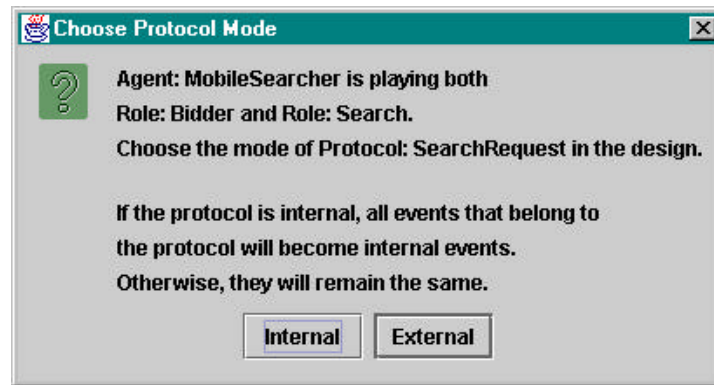


Figure 91 – Dialog to Choose a Protocol's Mode

When determining the mode for protocols in the design phase, it is possible for the user to make a mistake. Once the developer is finished determining the mode for protocols, if any event belongs to both an external and internal protocol, an error has been made. The developer will be notified, all of the protocols for that event will be reset to external, and the developer will be asked again to determine the mode for the protocols.

4.3.2.3 Agent Components

The result of the first stage of the transformation process is that components are created for the agent classes based on the roles they play. The state diagram for the component is initially the same as that of the task it implements. If there are any external events that belong to protocols that the developer determines to be internal communication, the events are transformed into internal events.

In the example, a component named FulfillSearchRequests was created for the SearchManager agent. The component's state diagram is the same as the Manager role's FulfillSearchRequests task, so it is not shown again. Figure 92 shows the state diagram for the Bid component created for the MobileSearcher agent. Every event that belongs to the SearchRequest protocol has been changed into an internal event. Every remaining external event belongs to the ContractNet protocol with the SearchManager agent. Figure 93 shows the Search task that was created for the MobileSearcher agent. Since every event was determined to belong to the SearchRequest protocol and that protocol was then determined to be an internal event, every event within the component was changed into an internal event.

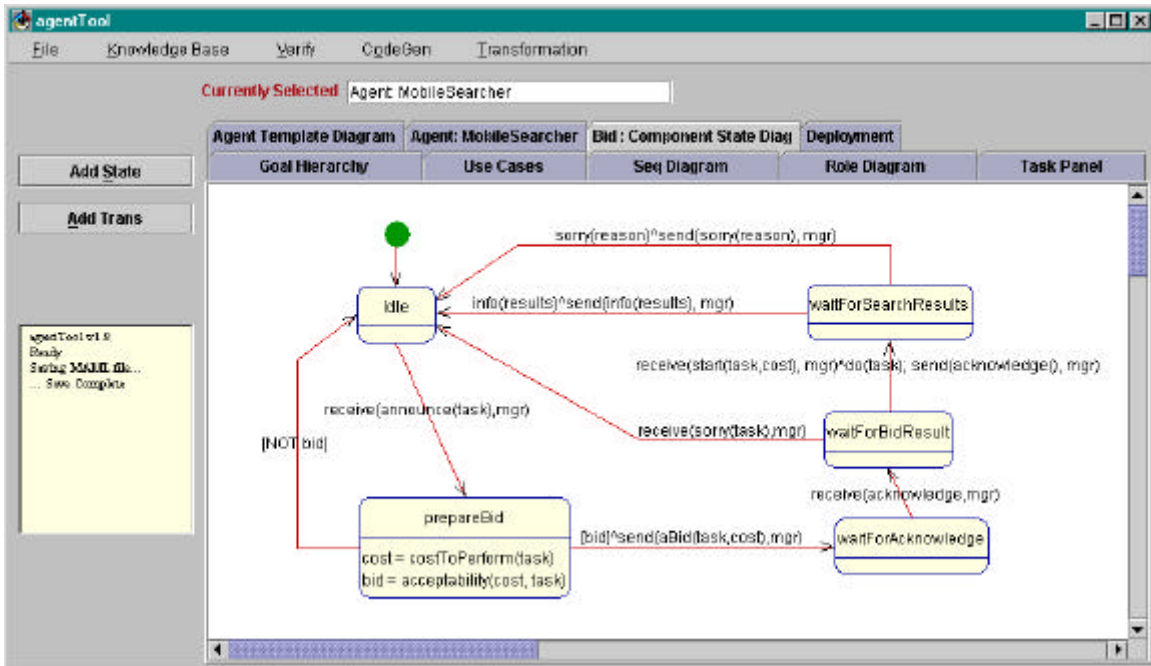


Figure 92 – MobileSearcher Agent's Bid Component

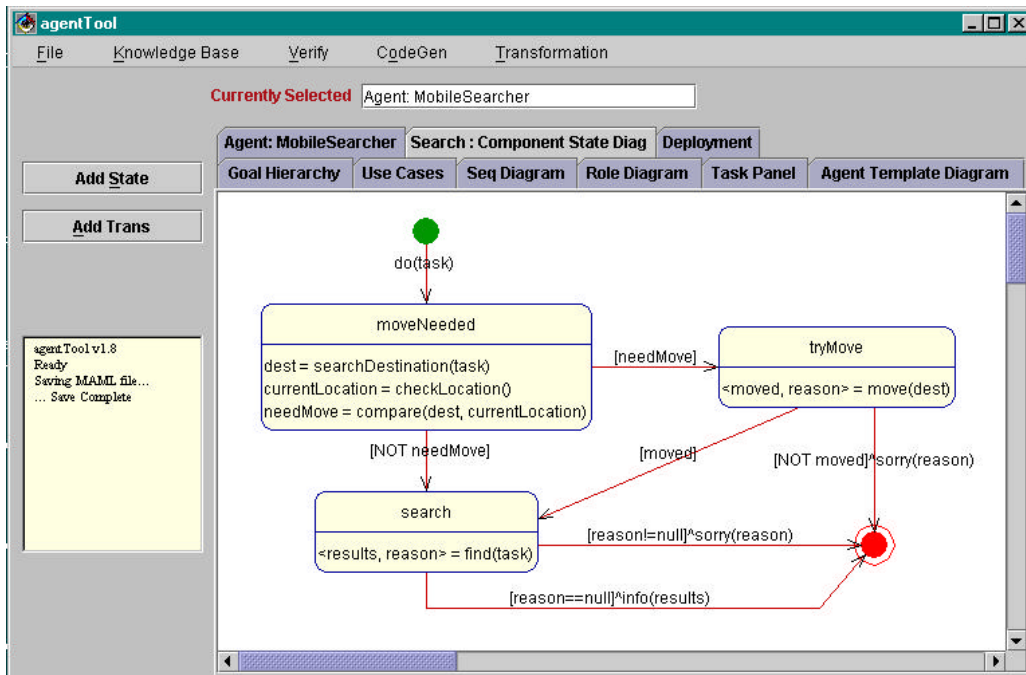


Figure 93 – MobileSearcher Agent's Search Component

4.3.3 Stage Two – Annotating Component State Diagrams

Now that every agent now has components with state diagrams, the second stage focuses on annotating the components to show where conversations will begin and end. This stage will also match up the events that become the initial messages of the conversations.

4.3.3.1 Matching up the First Messages of the Conversations

Although not the first transformations that take place during phase two of the transformation process, the first interaction requires the developer to determine if events in different components correspond. In most cases this can be done automatically, but as with determining the protocols for events, there are some cases where only the developer can make the determination. In these cases, a window is displayed with the state diagrams of both components that contain the events in question, as well as the Role Model from the analysis phase for reference. The transitions in the state diagrams that contain the events are highlighted, and developer is asked if the events correspond to each other.

In the example, there are three cases where the transformations can not automatically determine that the events corresponded. The first case, shown in Figure 94, involves the *aBid(task, cost)* message from the Bid component of the MobileSearcher agent to the FulfillSearchRequests component of the SearchManager agent. These events were intended to correspond, so the “YES” option is chosen.

The two other cases, shown in Figure 95 and Figure 96, involve the *start(task, cost)* and *sorry(task)* messages respectively, both from the FulfillSearchRequest component of the SearchManager agent to the Bid component of the MobileSearcher agent. As in the first case, these events were also meant to correspond, so the “YES” option was chosen for each case.

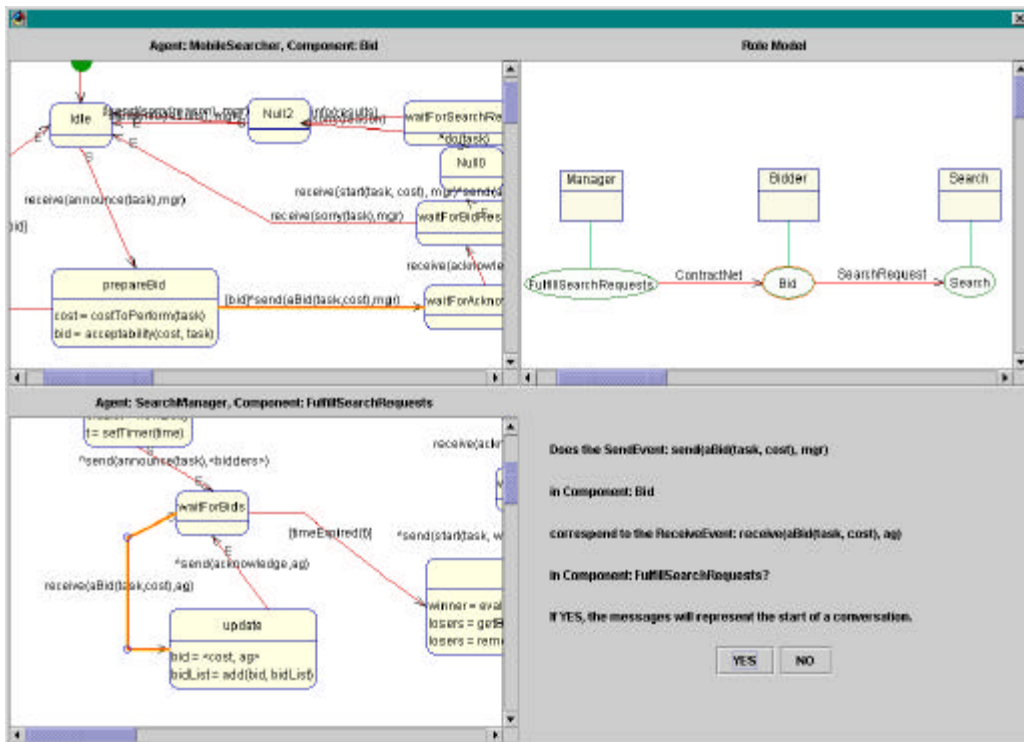


Figure 94 – First Event Match Decision

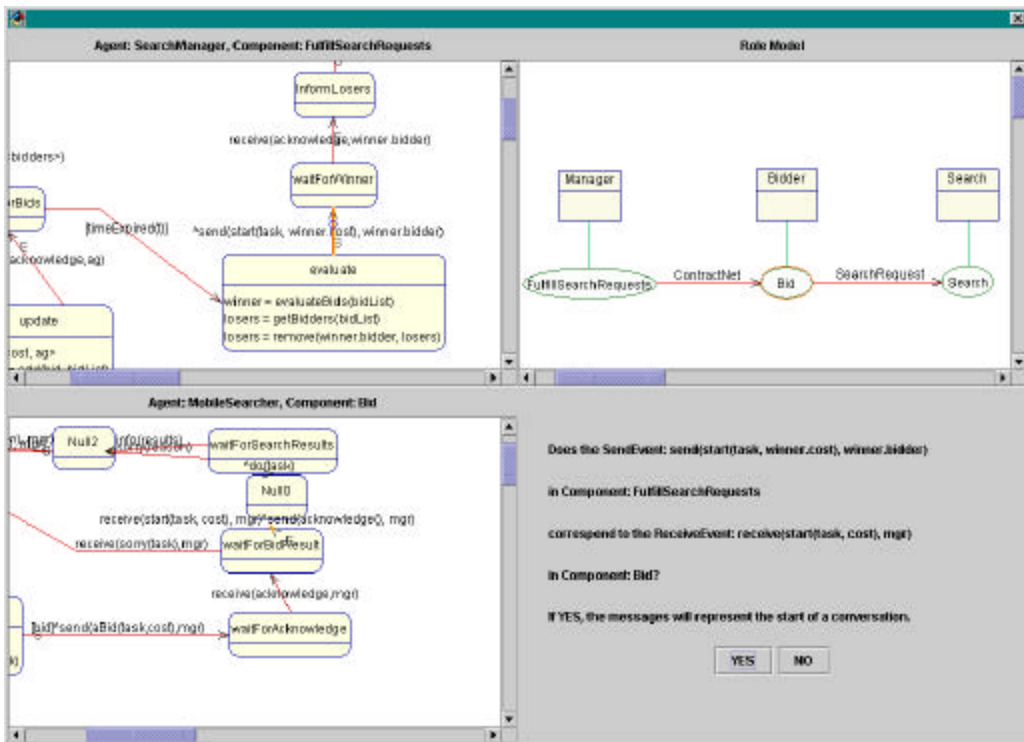


Figure 95 – Second Event Match Decision

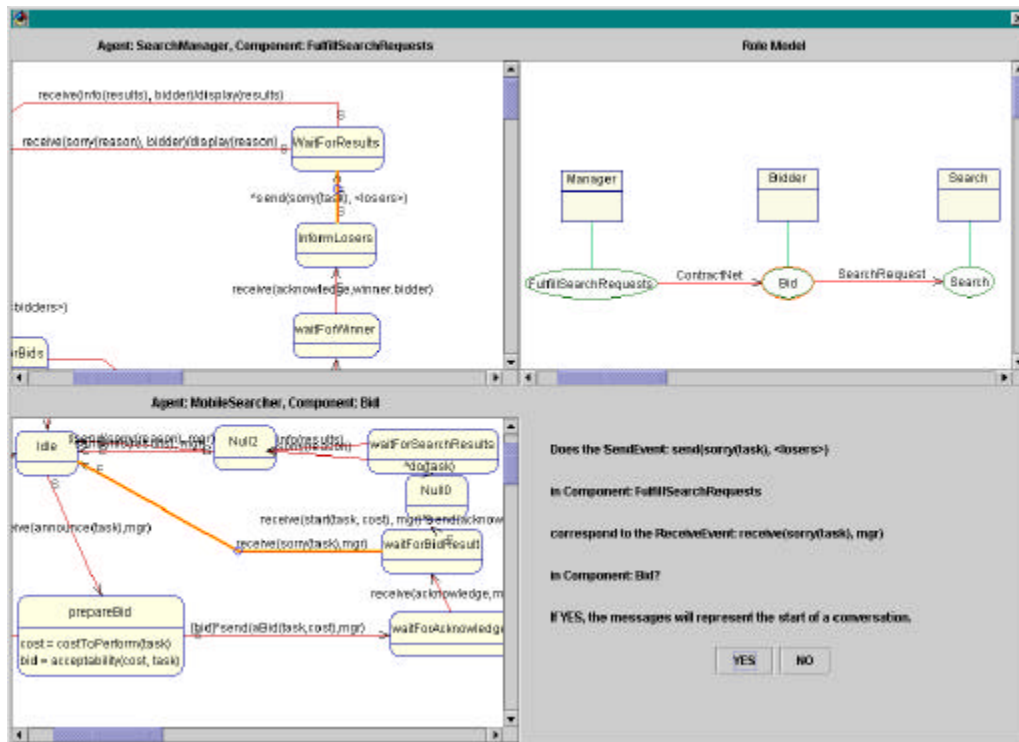


Figure 96 – Third Event Match Decision

4.3.3.2 Annotated Component State Diagrams

The result of the second stage of the transformation process is that all of the component state diagrams have been annotated, and the events that represent the beginning of conversations have been matched.

The annotated state diagram for the FulfillSearchRequests component is shown in Figure 97. The letter “S” at the beginning of a transition denotes where a conversation will begin, and the letter “E” at the end of a transition represents the end of a conversation. The states and transitions between the start and end labels will be removed from the component and placed in the conversation state diagrams in the next stage of the transformation process. There are many different conversations that emerge from this component, mainly because of the multicast messages that are sent in the ContractNet protocol.

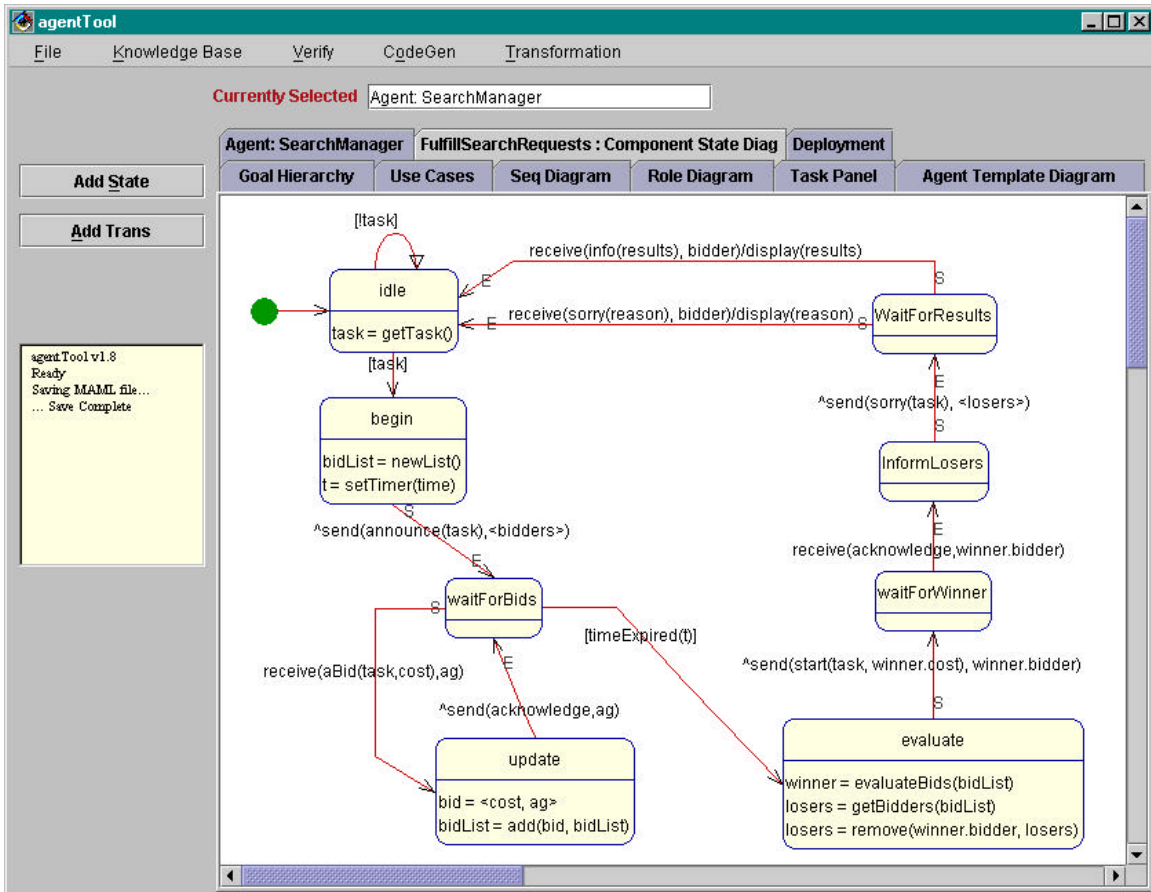


Figure 97 – Annotated FulfillSearchRequests Component

The annotated state diagram for the Bid component is shown in Figure 98. Again, the letters “S” and “E” denote the beginning and end of conversations that will be removed from the components during the next phase. There are also three new null states that were added during this stage of the transformation process. The new null states in the diagram are the result of splitting up the transitions that had both internal and external events, which allowed for a clear delineation of where the conversations begin and end.

Since the Search component for the MobileSearcher agent did not have any external events left after the first stage of transformations, the component remained unchanged during this stage.

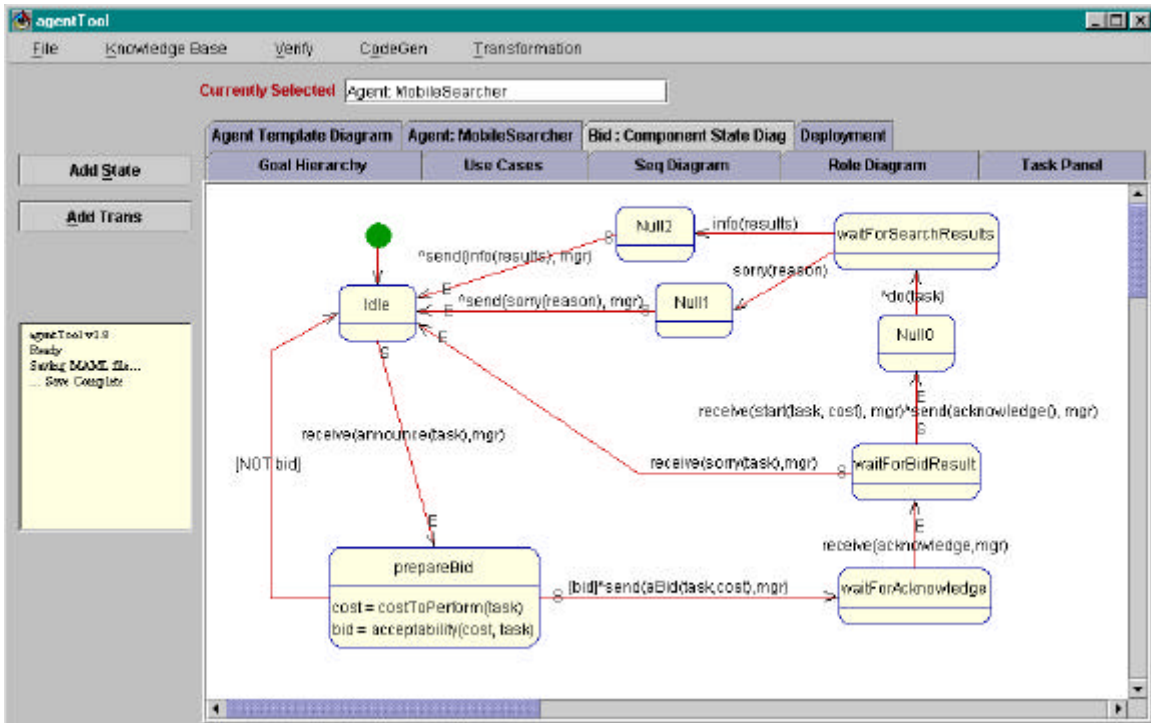


Figure 98 – Annotated Bid Component

4.3.4 Stage Three – Creating Conversations

At this point, all component state diagrams have been annotated and the initial messages of the annotated conversations have been matched. The last stage of the transformation process moves the states and transitions from the components to their appropriate Conversation Class Diagrams, replacing them with a transition that has an action to do the conversation. Figure 99 shows the Agent Class Diagram after the conversations have been added between the agents. The transformations gave the conversations generic but unique names.

The state diagram for the SearchManager agent’s FulfillSearchRequests component after harvesting the conversations is shown in Figure 100, and the MobileSearcher agent’s Bid component is shown in Figure 101. As shown in each diagram, the states and transitions that belong to the conversations are no longer in the component, but the state diagram that remains defines how the different conversations are coordinated together.

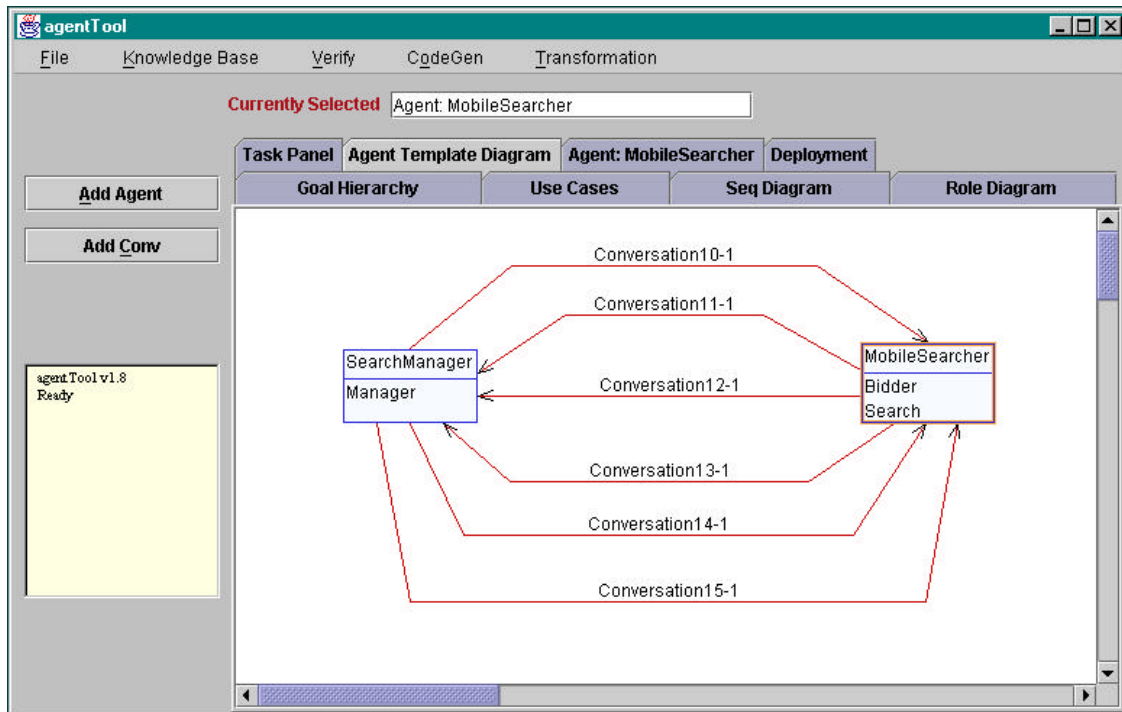


Figure 99 – Agent Class Diagram with Conversations

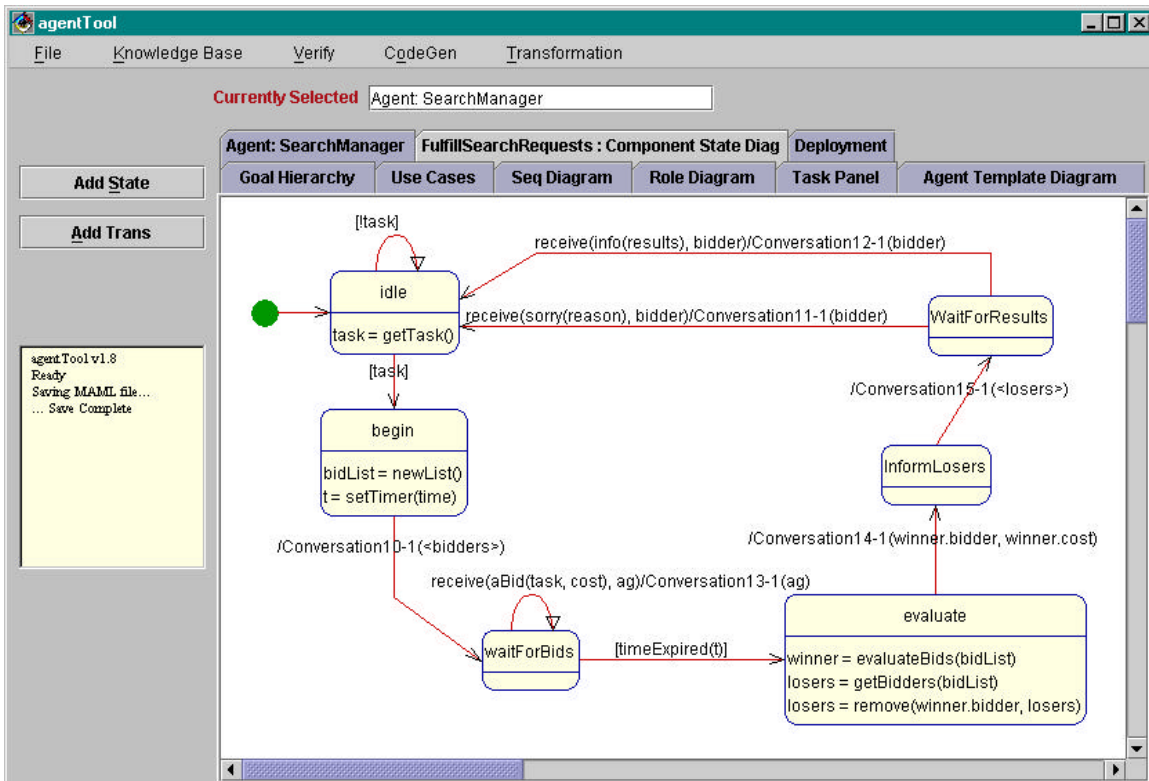


Figure 100 – FulfillSearchRequests Component After Stage Three

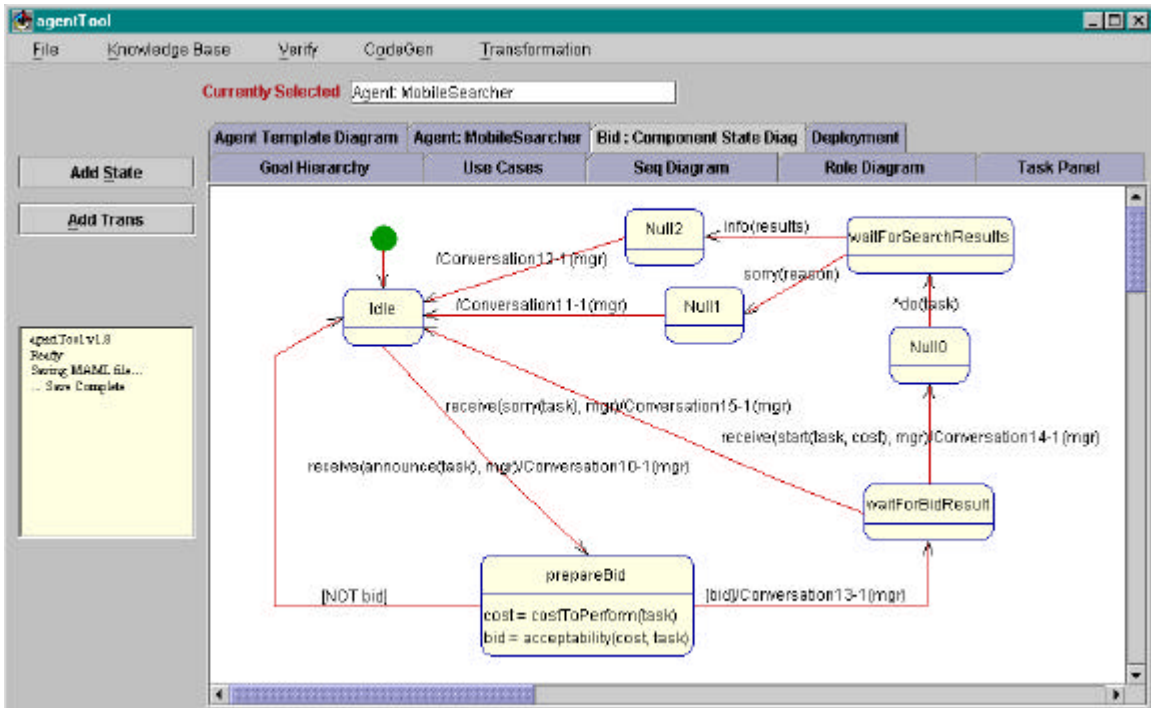


Figure 101 – Bid Component After Stage Three

Each conversation shown in the Agent Class Diagram now has the appropriate states and transitions in the initiator and responder halves, but only one conversation will be examined for the sake of brevity. Figure 102 shows the Communication Class Diagram for the initiator half of Conversation13-1. The state and transitions were added from the Bid component to create this is a very simple state diagram, where the MobileSearcher agent sends the *aBid(parent.task, parent.cost)* message and then receives the acknowledgement of the bid. The parameters *task* and *cost* for the *aBid* message were prepended with “parent.” to indicate that they belong to the parent (Bid) component, rather than the conversation.

Figure 103 shows the Communication Class Diagram for the responder half of Conversation13-1, harvested from the FulfillSearchRequests component. The *aBid(task, cost)* message is received and then an *acknowledge* message is sent in return. Also to note in the state diagram are the new actions and prepended variables. The *parent.task=task* action was added to the transition from the start state to the update state to set the variable named *task* in the parent component because it was received as a message parameter in this conversation and is either used within the parent component (FulfillSearchRequests), or within another

conversation that belongs to the parent component. Similarly, the *bidList* variables in the update state's second action were changed to *parent.bidList*, indicating that the *bidList* variable also belongs to the parent (FulfillSearchRequests) component.

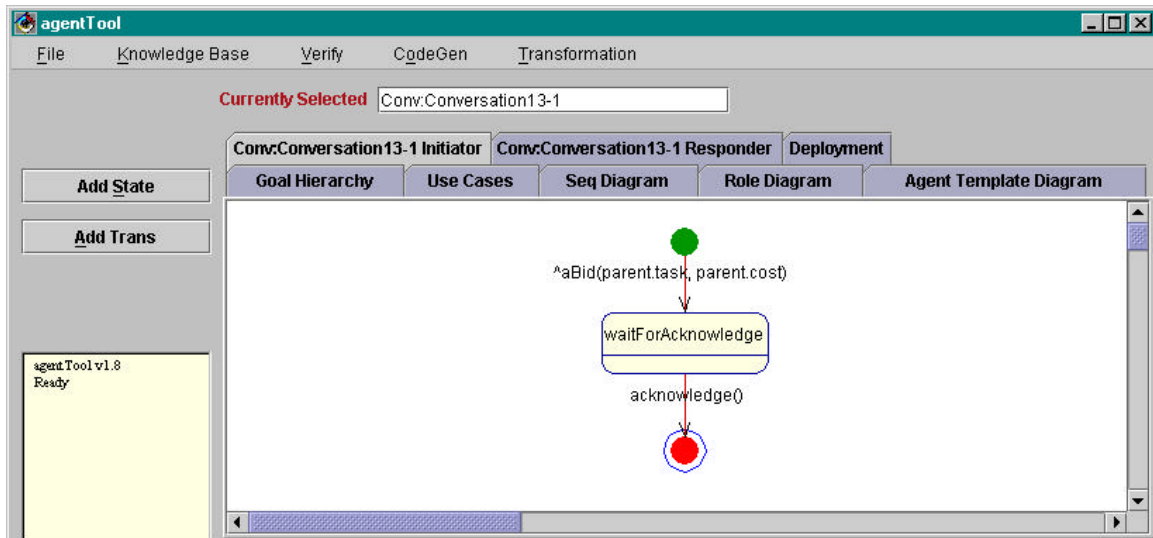


Figure 102 – Initiator Half of Conversation13-1

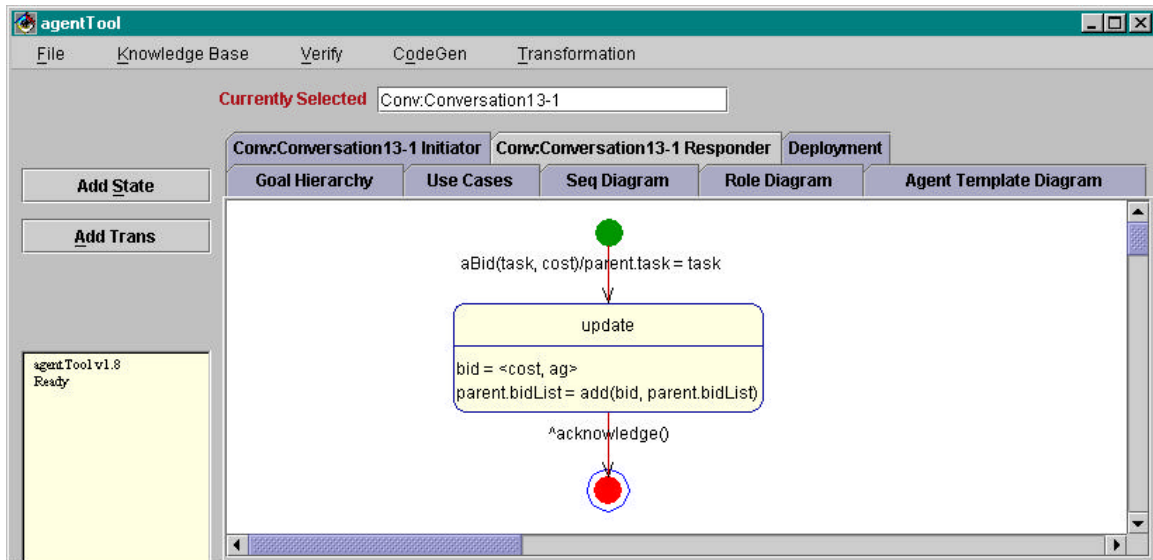


Figure 103 – Responder Half of Conversation13-1

4.4 Summary

This chapter described how the transformation system defined in Chapter III was successfully integrated with AFIT's agentTool multiagent development environment. An example was also presented to show the input required from the developer as design decisions, as well as the output from each stage of the transformation process.

V. Conclusions and Future Work

The previous chapters of this thesis described a semi-automatic formal transformation system for the MaSE methodology that generates agent components and conversations in the design phase from the Role Model and Concurrent Task Diagrams in the analysis phase. This chapter summarizes the conclusions from the previous chapters and suggests areas of future work that will enhance or extend this research.

5.1 Conclusions

The transformation system described in the previous chapters successfully accomplished the objectives established at the outset of this research. The transformations provide a correct and robust methodology for generating MaSE design models from the analysis models without losing any information from the analysis phase. A key contribution of the research in this thesis is that the MaSE methodology has necessarily matured and expanded. In order to develop formal transformations between the different models, the models had to be fully defined and the relationships between the models had to be identified.

The transformation system was developed as a three-stage process that incrementally forms the design models from the analysis models. The first stage creates the initial components for the agent classes based on the roles they play. Each agent component implements a task from the Role Model. Transformations in this stage also determine the protocols in which external events are passed. The second stage determines where conversations logically take place within the agent components, annotating the state tables accordingly. External events that constitute the first messages passed in the conversations are also matched, in some cases automatically and in others by the developer. The last stage of the transformation process creates the conversations between the agents based on the way the agent components are annotated. The states and transitions that belong to the conversations are removed from the component state tables and placed in state tables for the appropriate conversation halves. When the states

and transitions are removed from the component state tables, they are replaced with a transition that has an action that starts the conversation.

The transformation system is predominantly an automatic process, requiring only a few key design decisions from the system developer. There are many benefits from using an automated process that is known to preserve correctness from one model to the next. One key advantage offered by the transformation process is that it provides clear traceability between the analysis and design, simplifying the verification process. The developer also has much more confidence that no inconsistencies or errors occurred during the design process. Furthermore, when implemented in a development environment such as agentTool, the transformations allow the developer to maintain the system in the more abstract analysis models and regenerate the design when any changes are made. How many times during a software development project are the models in the analysis phase forgotten once the project enters the design phase? In many cases, there is simply not enough time or manpower to maintain the consistency between the models in the two phases. The transformation system presented in this thesis can eliminate that problem for system developers using the MaSE methodology.

5.2 Future Research Areas

The work done in this thesis brought to light many related areas where more work is still required. This section presents those areas of future work that would benefit not only the ongoing research being done at AFIT, but would have overarching impact on the development of multiagent systems and formal methods for software engineering as a whole.

5.2.1 Transformation Enhancements

While the transformation system defined in this thesis fully addressed the need to automate the transition between the analysis and design phases of the MaSE methodology, there are many other areas where the transformation system could be enhanced and expanded. The transformations were designed to be applied in the order they are presented. Throughout this process, the developer may be required to make

some decisions that affect the eventual system design. The transformations were implemented in agentTool accordingly, but the transformation system is a one-way process.

In order for the developer to be able to more effectively maintain the system at the analysis phase, the design decisions that the user makes should be maintained so they can be “undone”, “redone”, or “replayed” when applying the transformations. Most effective would be the ability to “step through” the decisions, similar to a web browser or program debugger. Currently, if the user needs to change the analysis of the system and desires to reapply the transformations, the developer has to make the same design decisions again during the transformation process. Being able to “replay” the previous design decisions would greatly enhance the interactive process. Furthermore, if a mistake is made while applying the transformations in agentTool, the developer is unable to stop, backup, and fix the mistake. The process must be started again from the very beginning. This is where “undo” and “redo” functionality would be of great benefit.

The transformation system could also be expanded by defining a set of transformations that automatically determine the attributes and methods for the agent components. These transformations should be straightforward, and would provide the user a more complete view of the internal design of the agent classes. The user could then supply information for the attribute types, function return types, function parameter types, and function pre- and post-conditions. Automated verification procedures could then be applied to the design to check for things like type-matching, etc. In addition, this would also provide the requisite formality for transforming the design into another formal language representation, and would improve the quality of the code that can be automatically generated from the design.

The last suggestion for further enhancements to the transformations deals with optimality. The scope of this thesis included defining transformations that preserved correctness, but in many cases optimality was forsaken for simplicity. One example is the way that conversations are created. After applying the transformations, there may be two conversations between two agents that do *exactly* the same thing. The current transformations do not even check for this, much less try to fix it. One possible approach to this problem is to add an additional set of transformations that optimize the design.

5.2.2 Formal Transformations for Mixed-Initiative Systems

The transformation process defined in this thesis can be thought of as a mix-initiative process because the developer is required to make various design decisions as the transformations are applied. However, none of the formal transformations capture the mixed-initiative aspect of the transformation process. They simply assume that the interaction takes place at the right time and the data is available when needed. Is there a way to formally capture the required user interaction and incorporate it into the rest of the transformation process?

Formalizing the mixed-initiative aspect of a system could have even greater implications for other systems that have more complicated user interaction patterns, especially in systems where user interaction is critically important. For example, a mixed-initiative strategic or tactical planning system should be able to provide the utmost confidence to the user that the system will *always* perform correctly. Part of that performance includes interaction with the user. If that interaction does not take place, or if it takes place in the wrong order, there could be dire consequences if the user is unaware of the error. The ability to formally capture the interaction and incorporate that with the rest of the system design could prove to be invaluable.

5.2.3 Formal Proof

While many example cases, simple and complex, were used to test the transformation system, there is no way to test *every* case to make sure that the transformations are absolutely correct and complete. The only way to ensure correctness and completeness is to develop a formal proof of the transformations, but in doing so would require even more rigorous formal definitions of the MaSE models and their properties. Developing a formal proof is no small task, and an automated tool that could identify any “missing” pieces in the formal representations of the models would be very useful, however developing such a tool may not be feasible. Even if the transformations are proved correct, there is still the matter of translating the formal representation of the transformation system into code, providing more than enough opportunity for error in the implementation. In that sense, unless there is also some automated method for

implementing verifiably correct transformations, the effort necessary to prove that the transformations are correct may have diminishing returns.

5.3 Summary

This research addresses the critical need for more reliable multiagent systems, which may be one way to provide information superiority for the Air Force and the Department of Defense during the 21st Century. Formal transformation systems reduce mistakes made during design and implementation of complex multiagent systems. No longer do system engineers have to hope that their design corresponds to the analysis, thus fulfilling the system requirements. Combining the work here with research done in the past, present, and future, provides the foundation necessary for developing multiagent systems that reliably operate in complex, distributed environments.

VI. Bibliography

- [1] Shalikhshvili, John M., Joint Vision 2010. Joint Staff: Pentagon, 1999.
- [2] Kelley, Jay W., Air Force 2025. 2025 Support Office, Air University, Air Education and Training Command: Air University Press, 1996.
- [3] DeLoach, Scott, "Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems," Proceedings of the Agent Oriented Information Systems '99 (AOIS'99), Seattle, WA, 1 May 1999, 1999.
- [4] Sycara, Katia, "Multiagent Systems," in AI Magazine, vol. 19[2], 1998, pp. 79-92.
- [5] DeLoach, Scott and Wood, Mark, "Multiagent Systems Engineering Methodology: the Analysis Phase," Air Force Institute of Technology, Technical Report AFIT/EN-TR-00-02, June 2000.
- [6] DeLoach, Scott, Wood, Mark, and Sparkman, Clint, "Multiagent Systems Engineering," submitted to International Journal on Software Engineering and Knowledge Engineering, 2000.
- [7] DeLoach, Scott and Wood, Mark, "Developing Multiagent Systems with agentTool," Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000), Boston, MA, July 7-9, 2000.
- [8] Wood, Mark, Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems. MS thesis, AFIT/GCS/ENG/00M-26. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 2000.
- [9] DeLoach, Scott, "Specifying Agent Behavior as Concurrent Tasks: defining the behavior of social agents," Air Force Institute of Technology, Technical Report AFIT/EN-TR-00-03, July 2000.
- [10] Robinson, David, A Component Based Approach to Agent Specification. MS thesis, AFIT/GCS/ENG/00M-22. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, 2000.
- [11] Hartrum, Thomas and Graham, Robert, "The AFIT Wide Spectrum Object Modeling Environment: An AWESOME Beginning," Proceedings of the National Aerospace and Electronics Conference (NAECON), Dayton, OH, October 10-12, 2000.
- [12] Saba, G. Mitchell and Santos, Eugene, "the Multi-Agent Distributed Goal Satisfaction System," Proceedings of the International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA '2000) 2000.
- [13] DeLoach, Scott, "Using agentMom," Air Force Institute of Technology, 2000.
- [14] Gamma, Erich, Heim, Richard, Johnson, Ralph, et al., Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley Pub. Co., 1995.
- [15] Wooldrige, Michael, Jennings, Nicholas, and Kinney, David, "The Gaia Methodology for Agent-Oriented Analysis and Design," Journal of Autonomous Agents and Multiagent Systems, 2000.
- [16] Iglesias, Carlos, Garijo, Mercedes, Gonzalez, Fose, et al., "Analysis and Design of Multiagent Systems using MAS-CommonKADS," Proceedings of the AAAI '97 Workshop on Agent Theories, Architectures and Languages, Providence, RI, July, 1997.
- [17] Clarke, Edmund and Wing, Jeannette, "Formal Methods: State of the Art and Future Directions," ACM Computing Surveys, vol. 28, No.4, 1996.
- [18] Hall, Anthony, "Seven Myths of Formal Methods," IEEE Software, 1990.

- [19] Green, C., Luckham, D., Balzer, R., et al., "Report on a Knowledge-Based Software Assistant," in Readings in Artificial Intelligence and Software Engineering, C. Rich and R. C. Waters, Eds. San Mateo, Calif.: Morgan Kaufmann, 1986, pp. 377-428.
- [20] d'Iverno, Mark, Fisher, Michael, Lomuscio, Alessio, et al., "Formalisms for Multi-Agent Systems," Proceedings of the First UK Workshop of Multi-Agent Systems 1996.
- [21] d'Iverno, Mark and Luck, Michael, "Development and Application of a Formal Agent Framework," Proceedings of the First IEEE International Conference on Formal Engineering Methods 1997.

Appendix A. Background

This appendix provides background information to assist the reader in understanding the concepts that are foundational to this thesis. The material is divided into two sections. In the first section (A.1), three different methodologies for developing multiagent system will be reviewed with respect to both the analysis and design phases. Particular attention will be paid to the guidance given for transitioning from analysis to design and the possibilities for automating this process for each methodology. In the second section (A.2), formal methods and transformation systems will be reviewed.

A.1 Multiagent System Methodologies

Agent technology has received a great deal of attention in the last few years, and as a result, the industry is beginning to develop methodologies for the development of multiagent systems. There are currently only a few complete and well defined methodologies for multiagent systems, and many of those lack guidance for transitioning from the analysis phase to the design phase.

The first phase of any software development is the analysis phase, which is the most crucial step to developing a system that meets the user's requirements and behaves in the desired manner. The objective of the analysis phase is to transform the requirements into some abstract representation of the system that can then be translated into a more concrete design. The analysis of a system should capture how the system will perform, i.e. *what* it does, not *how* it does it. Since multiagent systems have different characteristics than traditional software systems due to their distributive, cooperative nature, many of the analysis techniques attempt to capture those unique characteristics through the idea of roles, protocols, interactions, and organizations.

After the analysis phase, the design phase traditionally takes *what* the system has been modeled to do and define *how* the system will do it. The output of the design phase should be a set of models at a sufficiently low level of abstraction that they can be easily implemented. The step of transitioning from

analysis to design is critical because without clearly defined methods of doing so, a design could be developed that is inconsistent with the analysis, therefore introducing errors into the system.

A.1.1 Multiagent System Engineering Methodology

At AFIT, recent research has focused around developing and maturing the Multiagent Systems Engineering (MaSE) methodology that is intended to cover the complete life cycle of a multiagent system [3, 5-8]. Although MaSE is still being refined, it is probably the most complete and well defined methodology that has been developed for multiagent systems. MaSE is comprised of the 7 steps shown in Figure 104. The boxes represent the different models used in the steps and the arrows indicate a flow between the models. However, MaSE is also intended to be applied iteratively. The first three steps together represent the analysis phase of the methodology, while the last three steps represent the design phase. It should also be noted that many of the models in the methodology are closely related to each other and provide a fine level of granularity in detail from the beginning of the analysis to the end of the design, sometimes blurring the lines between traditional analysis and design.

In general, the analysis phase is devoted to capturing the goals of the system and then defining roles which will accomplish those goals through a set of concurrent tasks. In the design phase, agent classes are defined to play the roles, and conversations are used to describe the detailed communication protocols that the agent classes have with each other. The designer also develops a deployment strategy through the use of a Deployment Diagram, which details on what platforms individual agent instances will reside and what communication paths exist between the different agents.

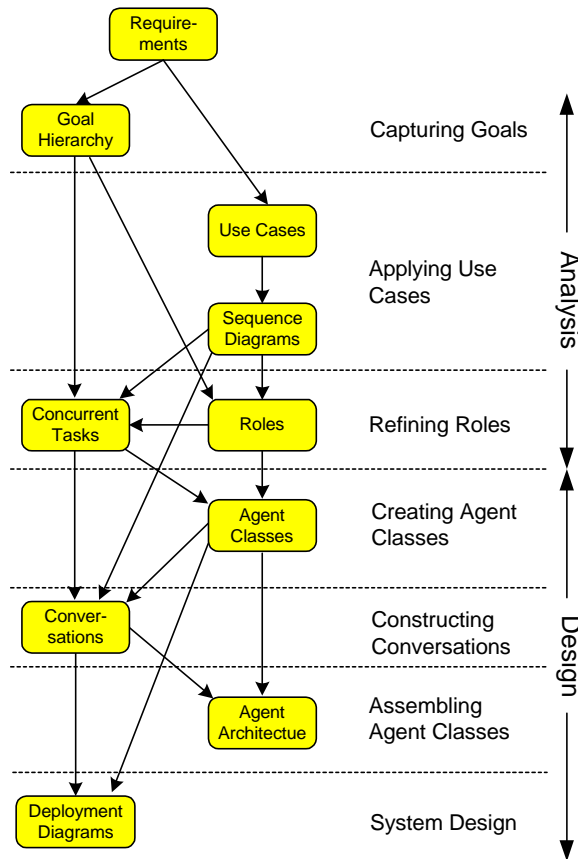


Figure 104 – Phases in the MaSE Methodology

A.1.1.1 Capturing Goals

The first step in MaSE is *Capturing Goals*, where the system analyst takes the system requirements and develops a Goal Hierarchy Diagram (shown in Figure 105), which is a structured set of system-level goals. Goals are defined as some system-level objective within the context of MaSE, and embody *what* the system is trying to achieve, and generally remain constant throughout the rest of the analysis and design process. A goal is typically a declaration of system intent, and phrased like “The system shall ...” Since MaSE uses a goal-driven approach, every action within the system must support a specific goal.

Capturing Goals is made up of two sub-steps. First, goals are identified from the initial system context, which is the collection of anything given to the analyst that is a starting point for system analysis.

Next, the goals are analyzed and structured into a Goal Hierarchy Diagram that is used later in the analysis phase. After roles have been identified, each role will be assigned some set of the goals. Intuitively, if all of the system requirements have been embodied as goals and all of the goals are being fulfilled by roles (which later become agents), then the system will meet the initial requirements.

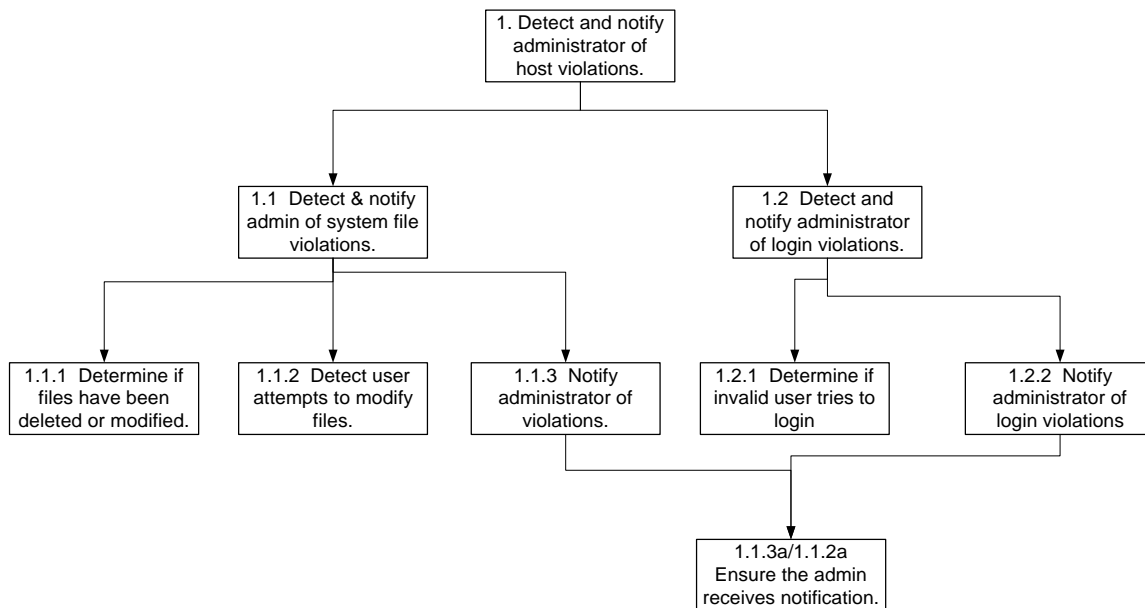


Figure 105 – Goal Hierarchy Diagram [5]

A.1.1.2 Applying Use Cases

Applying Use Cases is the next step in MaSE, where use cases are developed and then restructured as Sequence Diagrams. Uses cases are defined from the system requirements and are a narrative description of a sequence of events that capture desired system behavior. Use cases can be extracted from the requirements specification, user stories, or any other available source. Each use case should describe a particular instance of how the system will be used. It is important to capture both positive and negative use cases. *Positive use cases* describe what should happen during normal system operation, while *failure use cases* capture the desired sequence of events in the case of a breakdown or failure.

Once the system analyst has a representative set of Use Cases, those sequences of interactions are then captured in a more structured representation of a Sequence Diagram. Sequence Diagrams, as shown in

Figure 106, capture a sequence of messages between the different roles being played in the system. Sequence Diagrams provide a high-level view of how different roles interact to accomplish their goals, and are useful when constructing the tasks that each role has. The boxes at the top of the diagram represent system roles and the arrows between the lines represent events passed between roles. Time is assumed to flow from the tip of the diagram to the bottom.

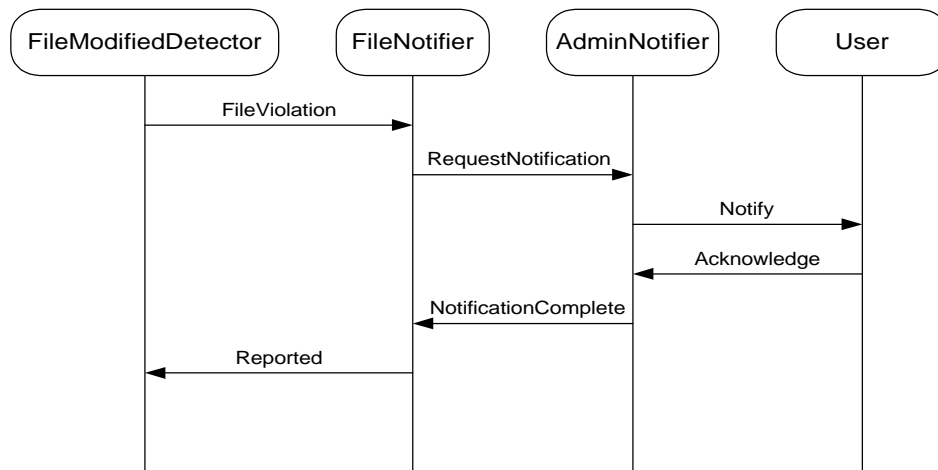


Figure 106 – Sequence Diagram [5]

A.1.1.3 Refining Roles

The next step is *Refining Roles*, where the analyst determines which roles will be played in the system and defines what tasks will be accomplished by each role. The Sequence Diagrams along with the Goal Hierarchy Diagram give the analyst insight into what roles should be played in the system. Each participant in the Sequence Diagrams is a candidate to become a role. Roles are defined much like an actor in a play, or a position in an organization (President, Vice President, Manager, etc). Each role must be responsible for accomplishing one or more goals in the Goal Hierarchy Diagram, and there must be at least one role responsible for each goal. Since roles form the foundation for creating agent classes and they represent the system goals from the analysis phase, they serve as a link between *what* the system is supposed to do (the analysis phase and goals) and *how* it accomplishes it (the design phase and agent classes).

In *Refining Roles*, a Role Model is used to graphically depict the roles in the system and the communication paths between those roles. Role Models can also enable the reuse of roles from previous systems. The basic idea is that patterns of agent roles are constructed, labeled, and archived. When a new system is developed, the patterns are recognized and a Role Model can be re-applied from an archive, resulting in a collection of agent roles that satisfy a subset of the system goals. As shown in Figure 107, the arrows on Role Models are paths of communication connecting roles, and the dots indicate multiplicity.

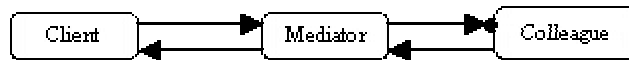


Figure 107 – A Role Model [8]

As part of defining the roles, the analyst also defines the tasks that each role has. . Tasks describe the behavior that a role must exhibit in order to accomplish its goal and are specified graphically using a finite state automaton as shown in Figure 108. A single role may have multiple concurrent tasks that define the complete behavior of the role. As a minimum, the messages in the sequence diagrams should also be messages being passed within a task. Concurrent tasks can be used to implement complex communication protocols such as Contract Net, Dutch Auction, etc. [9]. This is a very important part of the analysis as it allows the user to define how the system components will coordinate and interact with each other, which is the strength of multiagent systems. These tasks also lay the foundation for conversations between agent classes in the design phase of MaSE.

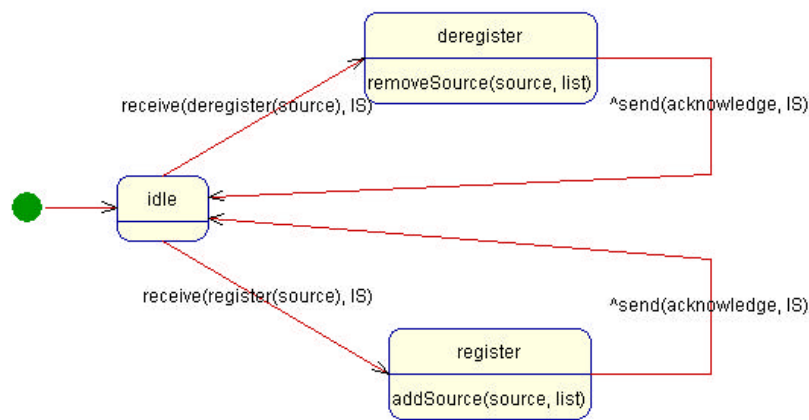


Figure 108 – Sample Task in MaSE

In order for tasks to execute concurrently, all tasks are assumed to start execution under a separate thread of control upon startup of the role and continue until the role terminates or an end state is reached. Activities take place within the states and specify functions carried out by the role. One important property of a task is that they are able to communicate with multiple tasks in order to accomplish their goals. The tasks can belong to the same role, or they may belong to a different role. Tasks that belong to the same role can coordinate with each other through internal events. In order for a task to communicate to a task of another role, events that represent external communication are specified using *send* and *receive* events. These events are defined to send and retrieve messages from an implied message-handling component of the role.

A.1.1.4 Creating Agent Classes

Creating Agent Classes is the first step in the design phase of MaSE. Agent classes are defined from roles and an Agent Class Diagram is produced, which depicts the agent classes and the conversations between them at a high level. An *agent class* is a template for an agent that will operate within the system, and is analogous to an object class in object oriented software engineering. When the system is deployed, the agents in the system will be actual instances of an agent class. Agent classes are defined by the roles they will play and the conversations they will participate in.

In order to ensure that all system goals are being met, each role must be played by at least one agent class. This will ensure that all of the goals in the analysis phase are traceable to agents in the design phase. In general, there is a one-to-one mapping from roles to agents where each role becomes an agent class. There may be some instances however where the designer decides to allow an agent class to play multiple roles, with the roles changing dynamically during execution. The designer may also allow a role to be played by more than one agent class. These design decisions are made either to share the capabilities and responsibilities of a role (allowing more than one agent class to play a role), or for performance enhancements by reducing communication overhead (combining multiple roles into an agent class).

In addition to defining the agent classes in the system, the designer must also identify the conversations those agent classes must participate in. The details of the conversations are left to the next step, Constructing Conversations, described in Section A.1.1.5. The conversations that an agent class must participate in can be derived from the external communications paths defined between the roles it plays. If roles A and B are defined by concurrent tasks that communicate with each other and agent 1 plays role A and agent 2 plays role B, then there must be a conversation between agent 1 and 2 to implement the communication described between roles A and B.

The product of this step is an Agent Class Diagram, as shown in Figure 109. Each rectangle represents an agent class and a directed line represents a conversation between the agent classes. The arrows on the lines indicate the initiator and responder in the conversation. An Agent Class Diagram is similar to object-oriented diagrams with two exceptions. First, agents are defined by the roles they play rather than by attributes and methods. Secondly, all relationships between classes are conversations that may take place between two agent classes.

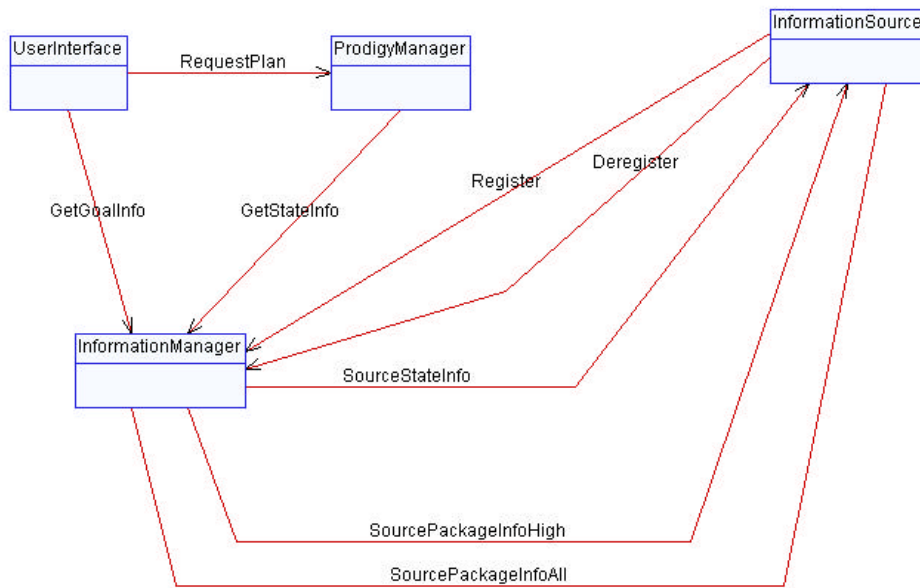


Figure 109 – Agent Class Diagram

A.1.1.5 Constructing Conversations

Constructing Conversations is the next step defined in MaSE. This step can actually happen before, after, or in parallel with the next step, Assembling Agents. The two steps are closely related, and it may be beneficial to alternate between them. In the previous step, Creating Agent Classes, the designer developed the Agent Class Diagram, which simply identified the agent classes and the conversations they have. The goal of this step is to define the details of those conversations.

Conversations are detailed coordination protocols between two agents and consist of two Communication Class Diagrams, one each for the initiator and responder. Conversations are at the heart of any multiagent system, as they detail how the different agents will communicate with each other. Like tasks, Communication Class Diagrams are finite state automaton that define the states and transitions for each half of a conversation. One example of a Communication Class Diagram is shown in Figure 110.

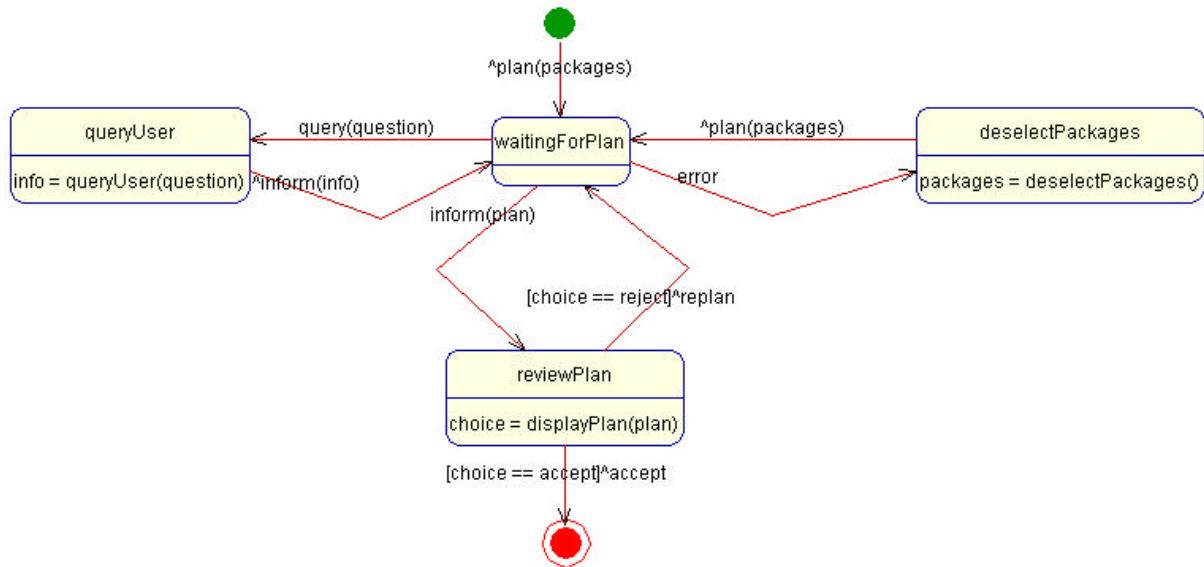


Figure 110 – Communication Class Diagram

As described in Section A.1.1.4, the roles that an agent class plays determine the set of conversations an agent class participates in. Likewise, the details of the conversations are derived from the tasks associated with those roles. Since tasks can capture communication between multiple roles as well as

communication with other tasks internal to its role, a task will likely be broken into more than one conversation. Conversations are defined to be point-to-point communication between just two agents, and every event within the Communication Class Diagram is defined to be a message to or from the other agent instance participating in the conversation. Conversations do not allow for communication with multiple agents simultaneously or internal events to be exchanged with components internal to the agent.

A.1.1.6 Assembling Agent Classes

In *Assembling Agent Classes*, the internal components of an agent are defined. This is a two-step process by first defining the agent architecture and then defining the components that make up that architecture. When constructing an agent architecture, the designer can either use a pre-existing architecture from a set of architecture style templates or design a custom architecture from scratch. Each architecture is built using components, which are also either custom-built or reused from an existing component library.

Each agent component is defined using an architectural modeling language combined with the Object Constraint Language. This allows the user to define attributes and functions that belong to the agent. Each component can also have a finite state automaton defining the dynamic characteristics of the component. The events passed within a component's dynamic model will be limited to internal events with other components that belong to that agent. There will not be any external send or receive events with other agents in the component's dynamic model. That is all accomplished through conversations.

A.1.1.7 System Deployment

The final step defined in MaSE is *System Deployment*. In this step, the designer takes the agent classes defined previously and instantiates actual agents. A Deployment Diagram is used to show all of the detailed information necessary to deploy the system, including numbers, types, and locations of agents within a system. An example of a Deployment Diagram can be found in Figure 111. The three

dimensional boxes are agents and the connecting lines between them represent conversations between those agents. A dashed-line box indicates that agents are housed on the same physical platform.

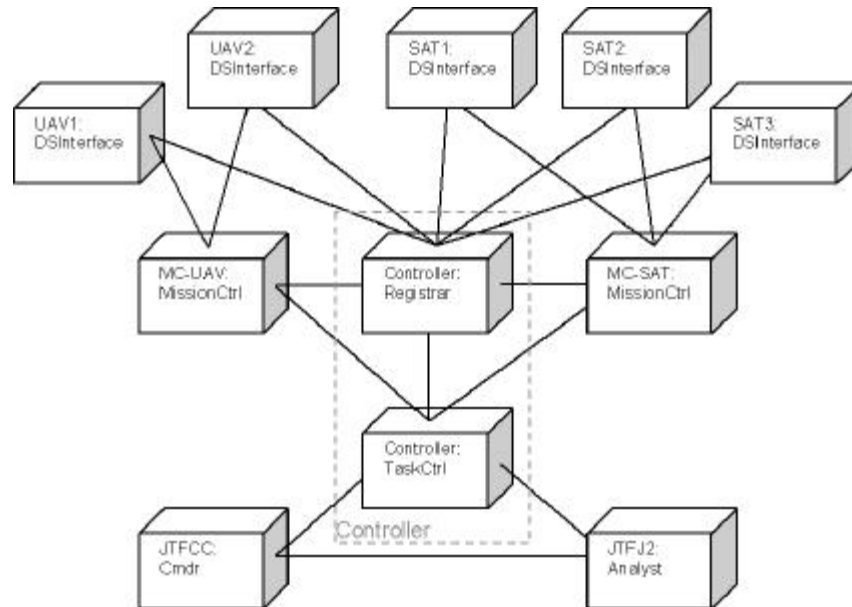


Figure 111 – Deployment Diagram [8]

Deployment Diagrams offer the designer an opportunity to tune the system by defining various configurations of agents and computers to maximize available processing power and network bandwidth. If the user has not specified the particular number of components or the specific computers on which certain agents must reside, the designer should consider the communication and processing requirements when assigning agents to computers. If two agents have a high degree of communication, then the designer may decide to deploy them on the same machine. However, overloading a machine with too many agents reduces the advantages of distribution gained using the agent paradigm. Furthermore, the designer may decide to dedicate a machine to a single agent if that agent has high processing requirements.

A.1.1.8 Transitioning from Analysis to design - MaSE

Not only does MaSE provide guidance from the analysis to design phase, but it provides guidance throughout the entire development process. The models in each step are clearly influenced by models in previous steps due to strong relationships between the information being presented in them. Specifically, it

is clear that roles are related to agent classes and the tasks that the roles perform are then both related to the conversations between those agent classes and to some aspects of the agent’s internal components. Since there are such strong relationships between the models in this methodology and there is clear guidance on making the transition from analysis to design, this methodology has the most promise for automation. While there are still many places where the developer has to make design decisions, once those decisions are made, going from one model to the next should be straightforward transformations.

A.1.2 Gaia Methodology

Another recent attempt at developing a full methodology for both analysis and design of a multiagent system is the Gaia methodology by Wooldrige, Jennings, and Kinney [15]. This methodology was developed for systems with a relatively small number (less than 100) of heterogeneous, autonomous agents attempting to maximize some global quality measure. Each agents services and the relationships they have with other agents are assumed to be static and will not change during run-time.

A.1.2.1 Analysis Phase - Gaia

The highest level of abstraction that the analysis phase attempts to capture is the *organization* of the system, which is a collection of roles that have relationships with one another and take part in systematic, institutionalized patterns of interactions with other roles (shown in Figure 112).

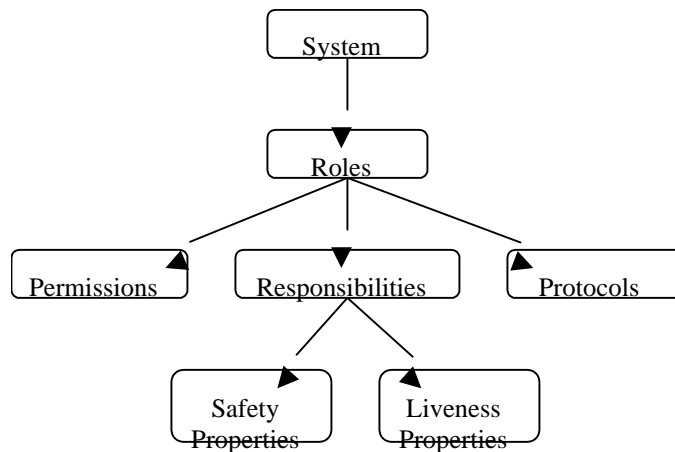


Figure 112 – Abstract Analysis Hierarchy [15]

The Gaia methodology views the system as a society or organization, and the elements of that society are defined as *roles*. Roles are a natural abstraction for a multiagent system and are analogous to a typical company structure. A company has roles such as “president”, “vice-president”, and “manager” all arranged in some hierarchical fashion. The idea of a role is not a static representation because someone acting as one role may later (or at the same time) also play the part of a different role. Roles are initially captured in a prototypical roles model, which will be incrementally expanded and fully elaborated by the end of the analysis phase.

A role is defined by four attributes: *responsibilities*, *permissions*, *activities*, and *protocols*. *Responsibilities* determine the functionality of a role and may be their key attribute. An example responsibility associated with the role of mail clerk might be to deliver and pick up mail to and from each required office. Responsibilities are divided into two types: *liveness properties* (something good that should happen) and *safety properties* (or invariants). *Permissions* are the “rights” associated with a role and identify the resources that are available to a role in order to achieve its responsibilities. In multiagent systems, these permissions tend to be information resources. *Activities* of a role are computations associated with a role that may be carried out by the agent without interacting with other agents. *Protocols* define the way that a role can interact with other roles, for example “Dutch auction”, “English auction”, or “Contract Net”. A protocol definition consists of the following attributes: *purpose*, *initiator*, *responder*, *inputs*, *outputs*, and *processing*. After protocols have been identified, an *interaction model* is produced which captures the recurring patterns of inter-role interaction.

A.1.2.2 design Phase - Gaia

In the Gaia methodology, the goal of the design phase is a little different than the traditional interpretation. The analysis model is transformed into a sufficiently low level of abstraction so that “traditional design techniques” can be applied to implement the agents. During the design phase, the designer will generate three models: the *agent model*, *services model*, and the *acquaintance model*.

The *agent model* documents the various agent types in the system. An agent type can be thought of as a set of agent roles. A designer can choose to package a number of closely related roles in the same agent type for the purpose of convenience and sometimes for better efficiency. The Gaia agent model also documents the run-time cardinalities of agent instances.

The *services model* identifies the services associated with each agent role and specifies the main properties of these services. Specifically, the inputs, outputs, pre-conditions, and post-conditions of each service are identified. Inputs and outputs are derived from the protocols model and pre- and post-conditions are derived from the safety properties of a role.

The *acquaintance model* simply defines the communication links that exist between agent types. They do *not* define what messages are sent or when messages are sent. This doesn't really seem to exploit the power inherent to multiagent systems, which is their ability to coordinate with each other through the idea of conversations or sequences of messages.

A.1.2.3 Transitioning from Analysis to design – Gaia

While the Gaia methodology gives sound guidance for developing the design models from the analysis models, the resulting design is still at a rather high level of abstraction. The methodology gives no real guidance on how to transform the design models into a sufficiently low-level of design to implement the system. The methodology needs to be expanded to either incorporate lower level design models or provide more guidance on how to refine the current models to a “traditional” system design. With such a lack of detail given, it would be very difficult to try and automate this process. To automate the generation of the design models described in this methodology would be of little use.

A.1.3 MAS-CommonKADS

Another complete multiagent system methodology that has been proposed by Iglesias, Garijo, Gonzalez, and Velasco is the MAS-CommonKADS methodology [16]. This methodology extends CommonKADS for multiagent systems by adding techniques from object oriented methodologies and

protocol engineering. The general software process combines a risk-driven approach with a component-based approach.

A.1.3.1 Analysis Phase - MAS-CommonKADS

The first phase of analysis is *Conceptualization*, where the analyst determines use cases from the initial user requirements and then formalizes them with Message Sequence Charts. The purpose of this phase is to capture roles and to develop an initial understanding of the interactions that must take place between those roles. After the *Conceptualization* phase, a requirements specification of the system will be generated through the development of six models, each consisting of constituents (the entities to be modeled) and relationships between the constituents.

The first model is the *Agent model*, which specifies the agent characteristics such as reasoning capabilities, skills (sensors / effectors), services, agent groups and hierarchies (both modeled in the organization model). The second model is the *Task model* that describes the tasks that the agents can carry out through description of goals, decompositions, ingredients and problem-solving methods. The third model, the *Expertise model*, describes the knowledge (information sources) needed by the agents to achieve their goals. The fourth model is the *Organization model* that describes the organization into which the MAS is going to be introduced and the social organization of the agent society. The *Coordination model* is the fifth model, which describes the conversations between agents: their interactions, protocols and required capabilities. The last model, the *Communication model*, details the human-software agent interactions and the human factors for developing these user interfaces.

There are no examples of the models in this methodology, but it does describe how these models are developed in a risk-driven way through the following five steps. The first step is *Agent modeling*, where you develop the initial instances of the agent model for identifying and describing the agents. The next step is *Task modeling*, where tasks are decomposed and the goals and ingredients of the tasks for each agent are determined. The third step is *Coordination modeling*, where the coordination model for describing the interactions and coordination protocols between the agents is developed. The fourth step is

Knowledge modeling, where the knowledge on the domain, the agents (knowledge needed to carry out the tasks and their proactive behavior) and the environment (beliefs and inferences of the world, including the rest of the agents) is modeled. The last step is *Organization modeling*, where the organization model is developed. Depending on the type of project, it may be necessary to model the organization of the enterprise into which the MAS is going to be introduced to study the feasibility of the proposed solution. In this case, two instances of the organization model are developed: before and after the introduction of the MAS. This model is also used to model the software agent organization.

A.1.3.2 design Phase - MAS-CommonKADS

From the initial set of models defined in the analysis phase, a *design model* is produced that is subdivided into three sub models, the *Agent Network design*, *Agent design*, and *Platform design*. The *Agent Network design* model describes the infrastructure of the MAS and consists of network, knowledge and coordination facilities. The agents that maintain this infrastructure are also defined, depending on the required facilities such as network facilities (agent name service, register and subscription service, transport / application protocols, etc.), knowledge facilities (ontology servers, knowledge representation language translators, etc.), and coordination facilities (coordination protocols, protocol servers, group management facilities, police agents, etc.).

The *Agent design* model defines the appropriate architecture for each agent, and agents can be introduced or subdivided according to pragmatic criteria. Each agent is subdivided in modules for user communication (from communication model), agent communication (from coordination model), deliberation and reaction (from expertise, agent and organization models), and external skills and services (from agent, expertise and task models).

The last model is the *Platform design* model where the decisions on software (multiagent development environment) and hardware that are needed for the system are captured.

A.1.3.3 Transitioning from Analysis to design – MAS-CommonKADS

While there are some indications of what models in the analysis phase affect the models in the design phase, exactly *how* they are related is not specified. In fact, there are no examples of some of the analysis models and no examples of the design models. Without more concrete information on the models and how they relate to each other, one can only speculate on how easy it would be to automate the process of transforming the analysis models into design models.

A.2 Formal Methods

Computer systems continue to grow in scale, functionality and complexity, increasing the likelihood of subtle errors. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by using formal methods, which are mathematically based languages, techniques and tools for specifying and verifying such systems. While formal methods do not necessarily guarantee correctness, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go unnoticed [17].

Hall [18] uses the term *formal methods* to describe the use of mathematics in software and details the main activities in using formal methods:

- writing a formal specification
- proving properties about the specification
- constructing a program by mathematically manipulating the specification
- verifying a program by mathematical argument

The first step, writing a formal specification, may be the most important part of formal development. A formal specification gives an unambiguous, precise definition of exactly what the system is intended to do, and is the foundation for all other activities relating to formal development. For many

projects, this is the only part of the development that is formal. The major benefit of using formal methods to write a system specification is that they require the analyst to more fully understand the system because errors and ambiguities become blatantly obvious.

Once a formal specification has been developed, since the specification is mathematical in nature, the developer can prove things about the specification, as well as the program. These proofs may deal with the consistency of the specification, the completeness of operation definitions, or that the specification will meet certain key requirements. For safety-critical systems, these proofs may be of great importance. In any case, errors at this stage are more costly than implementation errors, so proofs of these properties are correspondingly more important than proofs of implementation.

If a developer wants to implement a system formally, instead of writing the program and then trying to prove that it meets the specification, the program is constructed through a transformation system, described below. Since each step of the transformation system is provably correct, then the program is correct by construction and can be mathematically verified.

A.2.1 Transformational Programming

Within the recent developments in formal methods, a new paradigm for software development has emerged, transformational programming, in which software is developed, modified, and maintained at the specification level, and then automatically transformed into production-quality software [19]. The basic idea behind a transformation system is to take a formal specification for the system and apply a series of correctness-preserving transformations that translate the system specification into a system design and then into executable code. If each transformation preserves correctness, the resulting system is guaranteed to be correct, but only with respect to the specification. If the specification is not correct, neither will resulting design and code be correct.

Hartrum and Graham [11] describe a semi-automated software synthesis process using a transformation system shown in Figure 113. First, domain knowledge is stored in a formal domain model. Then a formal specification for a specific problem is generated by an application engineer from the domain

model. The developer will then apply a series of transformations, each of which are verified to preserve the correctness of the system, to produce a formal design specification. Finally, further transformations are applied to generate the executable code.

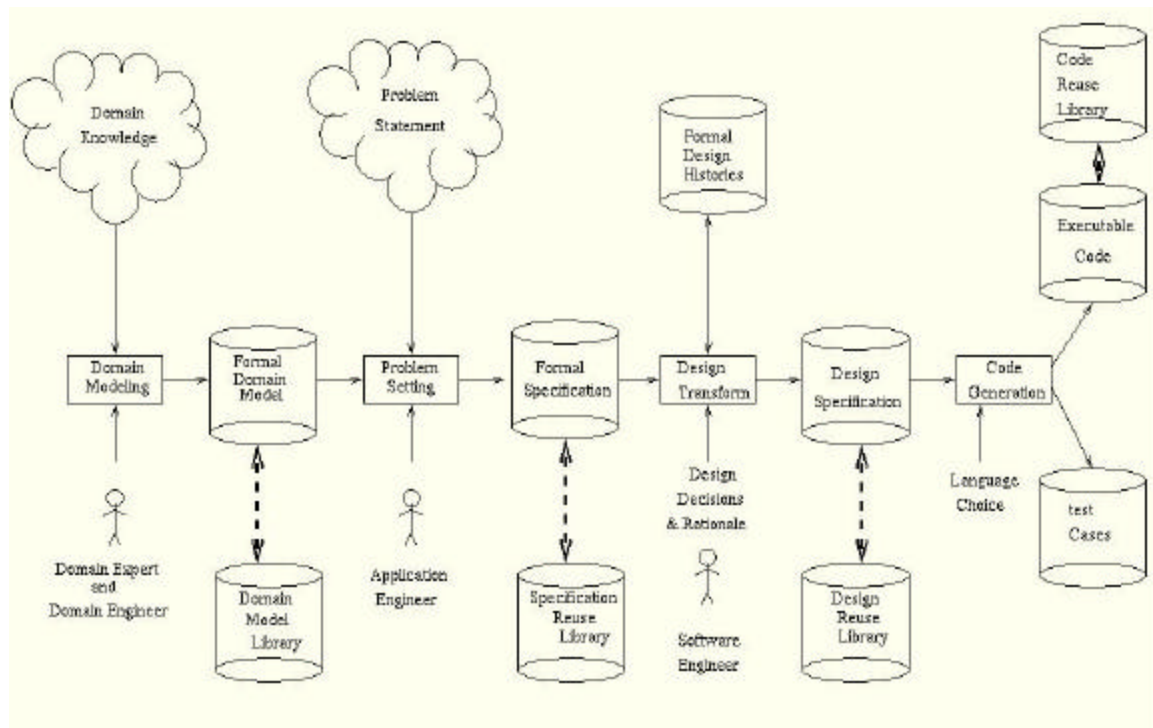


Figure 113 – Typical Transformation System [11]

While transformational programming has the obvious benefit of decreasing the chances for errors in the implementation, there are also other more subtle benefits. First, by developing the system in an automated fashion from the specification, system maintainability is greatly increased because changes to the system will also be made to the specification, not directly to the code. In the traditional software life cycle, over half of the cost is attributed to software maintenance and modifications because they are done at the code level. After a few rounds of modifications, the code has usually become unstable and is very difficult to make further changes to. The original design information is usually lost and the documentation has not been maintained, making it inadequate and outdated. The only recourse is an expensive reengineering effort that includes recovering the design of the existing system. In transformational programming, changes are made to the specification, and the code is automatically generated by re-

applying the transformations, most of which will not have changed since the last time through the transformation process.

Another benefit of transformational programming is that it makes it easy to reuse portions of previous software systems when abstract components can easily be adapted to the context of a new software system. Instead of trying to reuse portions of the code, which can be difficult to deal with even in a modularized system, a developer can simply reuse portions of the specification, which are abstract and easier to manipulate. Additionally, the specification may be contained in different analysis models, where CASE tools can make the reuse of these models almost trivial.

A.2.2 Formalisms for Multiagent Systems

Agents are a natural next step in software engineering, representing fundamentally new ways of viewing complex distributed systems in the context of societies of cooperating autonomous components. Since agents have unique properties, new formal representations must be developed in order to take advantage of formal methods in the development of agent-oriented and multiagent systems. d'Iverno, et al. [20] list the necessary attributes of formalisms for agents:

- provide a precise and unambiguous language for specifying systems' components and behavior
- address the needs of practical applications of agents, by being capable of expressing some or all of various aspects of agency including, but not limited to, perception, action, belief, knowledge, goals, motivation, intention, desire, motivation
- help identify properties of agent systems against which implementations can be measured and assessed
- measure, evaluate, classify, and study implementations

They also further detail attributes of a formalism for multiagent systems, as they add another dimension to agent-oriented systems. They state that formalisms for multiagent systems should also deal with the multiplicity of agents, group properties of agent systems, such as common knowledge and group intentions, and interaction among agents, such as communication and cooperation. In a later paper,

d'Iverno and Luck [21] extend the framework to include inter-agent relationships, and give an approach using Z.

A.3 Summary

This appendix provided background information on previous research that supports this thesis. The first three sections presented three multiagent engineering methodologies; Multiagent Systems Engineering (MaSE), the Gaia Methodology, and MAS-CommonKADS. The analysis and design phases were described for each methodology, as well as any guidance for transitioning from the analysis models to the design models. The last section presented some background information on formal methods and transformation systems.

Appendix B. Functions Used in the Transformations

This appendix provides formal definitions for some of the functions used in the transformations in Chapter III. Each function is defined by pre- and post-conditions, and returns a Boolean value based on the evaluation of the post-condition expression.

B.1 The *isAssigned* Function

The *isAssigned* function is a recursive function that takes a *SendEvent* and a transition and looks backward in the state table to see if an action was used to set the recipient of the send event.

function *isAssigned*(*se* : **SendEvent**, *t* : **Transition**, *st* : **StateTable**) returns **Boolean**

Precondition : true

Postcondition :

$$\begin{aligned} & (\exists a : \mathbf{Action}, s : \mathbf{State} \bullet s \in \text{st.states} \wedge s = \text{t.from} \wedge a \in \text{s.actions} \wedge \text{se.recipient} \in \text{a.lhs}) \\ & \vee (\exists t2 : \mathbf{Transition}, \text{re} : \mathbf{ReceiveEvent} \bullet t2 \in \text{st.transitions} \wedge t2.\text{to} = \text{t.from} \\ & \quad \wedge \text{re} = \text{t2.receiveEvent} \wedge (\text{re.sender} \neq \text{se.recipient} \vee \text{re} = \text{null}) \\ & \quad \wedge \neg (\exists \text{se2} : \mathbf{SendEvent} \bullet \text{se2} \in \text{t2.sendEvents} \wedge \text{se2.recipient} = \text{se.recipient}) \\ & \quad \wedge ((\exists a : \mathbf{Action} \bullet a \in \text{t2.actions} \wedge \text{se.recipient} \in \text{a.lhs}) \wedge \text{isAssigned}(\text{se}, \text{t2}, \text{st}))) \end{aligned}$$

B.2 The *usedInAction* Function

The *usedInAction* function returns true under three conditions: 1) the parameter's name is used in the action's lhs tuple 2) the parameter's name is used in a tuple in the action's rhs and 3) the parameter is used in a *FunctionCall* in the action's rhs.

function *usedInAction*(*p* : **Parameter**, *a* : **Action**) returns **Boolean**

Precondition : true

Postcondition :

$$\begin{aligned} & \exists \text{param} : \mathbf{String}, f : \mathbf{FunctionCall} \bullet \\ & \text{param} = \text{p.name} \wedge (\text{param} \in \text{a.lhs} \vee \text{param} \in \text{a.rhs} \vee (f = \text{a.rhs} \wedge \text{p} \in \text{f.parameters})) \end{aligned}$$

B.3 The *usedInTransition* Function

The *usedInTransition* function returns true if the parameter given as input is also a parameter of the transition's receive Event, the Event of the receiveEvent, one of the send Events, the Event of one of the sendEvents, or it is used in one of the transition's actions. There was no formal definition given for the Boolean expressions used in a transition's guard condition. Boolean expressions can be represented in whatever formal language is chosen. Therefore, the *usedInGuard* function is defined to return true if the parameter is used somewhere in the guard condition.

function *usedInTransition*(p : **Parameter**, t : **Transition**) returns **Boolean**

Precondition : true

Postcondition :

$$\begin{aligned} & \exists e : \text{Event}, se : \text{SendEvent}, re : \text{ReceiveEvent}, a : \text{Action} \bullet \\ & (e = t.\text{receive} \wedge p \in e.\text{parameters}) \vee (e \in t.\text{sends} \wedge p \in e.\text{parameters}) \\ & \vee (re = t.\text{receiveEvent} \wedge e = re.\text{event} \wedge p \in e.\text{parameters}) \\ & \vee (se \in t.\text{sendEvents} \wedge e = se.\text{event} \wedge p \in e.\text{parameters}) \\ & \vee (a \in t.\text{actions} \wedge \text{usedInAction}(p, a)) \\ & \vee \text{usedInGuard}(p, t.\text{guard}) \end{aligned}$$

B.4 The *isNeeded* Function

The *isNeeded* function is used to determine if a parameter needs to be supplied to an action that starts a conversation. The function returns true if there is a transition belonging to the conversation that uses a parameter and that parameter is not assigned within the conversation prior to being used. The function also returns true if the parameter is used in an action in a state that belongs to the conversation, and that parameter is not assigned prior to being used. The *usedInAction()* and *usedInTransition()* functions are used as defined earlier, and the *isAssigned()* function returns true if the parameter has been set in an action either in a state or on a transition before the parameter is used. This means that parameters that are used in a conversation before explicitly being set must be supplied as a parameter when the conversation is started.

function `isNeeded(p : Parameter, convs : { Conversation }, st : StateTable)` returns **Boolean**

Precondition: true

Postcondition:

$(\exists t : \mathbf{Transition} \bullet$

$t \in \text{st.transitions} \wedge (\text{convs} \cap t.\text{conversations} \neq \{\}) \wedge \text{usedInTransition}(p, t)$

$\wedge \neg \text{isAssigned}(p, \text{convs}, \text{st})$)

$\vee (\exists s : \mathbf{State}, a : \mathbf{Action} \bullet$

$s \in \text{st.states} \wedge (\text{convs} \cap s.\text{conversations} \neq \{\}) \wedge a \in s.\text{actions} \wedge \text{usedInAction}(p, a)$

$\wedge \neg \text{isAssigned}(p, \text{convs}, \text{st})$)

VITA

Lt Clint Houston Sparkman was born in September 1974 in Nacogdoches, Texas. He graduated from The Colony High School in The Colony, Texas in June 1993, and he married Casey J. Hall on June 22, 1996. In December 1997, he received a Bachelor of Science degree in Computer Science from Southern Methodist University, and was commissioned as a distinguished graduate through Detachment 835 AFROTC at the University of North Texas. His first assignment was to the Air Force Research Lab at Kirtland AFB, New Mexico. In August 1999 he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, Lt Sparkman will be assigned to the 690th Computer Systems Squadron, Air Intelligence Agency at Kelly AFB, Texas.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 20-03-2001	2. REPORT TYPE Master's Thesis	3. DATES COVERED (From - To) APR 2000 - Mar 2001
---	-----------------------------------	---

4. TITLE AND SUBTITLE TRANSFORMING ANALYSIS MODELS INTO DESIGN MODELS FOR THE MULTIAGENT ENGINEERING SYSTEMS (MASE) METHODOLOGY	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S) Clint Houston Sparkman	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Bldg 640 Wright-Patterson AFB OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/01M-12
---	---

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NM Attn: Dr. Robert L. Herklotz 801 N. Randolph St., Room 732 Arlington, VA 22203-1977 Phone: 703-696-8421/6565 Fax: 703-696-8450	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
Maj Scott A. DeLoach, Assistant Professor of Computer Science and Engineering, 785-3636x4622, scott.deloach@afit.edu

14. ABSTRACT
Agent technology has received much attention in the last few years because of the advantages that multiagent systems have in complex, distributed environments. For multiagent systems are to be effective, they must be reliable, robust, and secure. AFIT's Agent Research Group has developed a complete-lifecycle methodology, called Multiagent Systems Engineering (MaSE), for analyzing, designing, and developing heterogeneous multiagent systems. However, developing multiagent systems is a complicated process, and there is no guarantee that the resulting system meets the initial requirements and will operate reliably with the desired behavior.
The purpose of this research was to develop a semi-automated formal transformation system for the MaSE methodology, as one part of formal agent synthesis, that derives the system design based on the analysis. Since each transform in the transformation system preserves correctness, the designer can be sure that the resulting system design is correct with respect to the system specification. A secondary goal of this research was to develop a proof-of-concept module for agentTool that implements the transforms.

15. SUBJECT TERMS
Software Engineering, Agent, Transformations, Formal, Methodology, Components, Conversations, agentTool, State Table

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT UNCLASS	b. ABSTRACT UNCLASS	c. THIS PAGE UNCLASS	UU	161	Maj Scott A. DeLoach 19b. TELEPHONE NUMBER (include area code) 785-3636 x4622