DESIGN & SPECIFICATION OF DYNAMIC, MOBILE, AND RECONFIGURABLE MULTIAGENT SYSTEMS

THESIS

Athie L. Self, Captain, USAF

AFIT/GCS/ENG/01M-11

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

DESIGN & SPECIFICATION OF DYNAMIC, MOBILE, AND RECONFIGURABLE MULTIAGENT SYSTEMS

THESIS

Presented to the faculty of the Graduate School of Engineering and Management

of the Air Force Institute of Technology

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Athie Lee Self, B. S.

Captain, USAF

AFIT/GCS/ENG/01M-11

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 the United States Government.

AFIT/GSC/ENG/01M-11

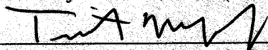
DESIGN & SPECIFICATION OF DYNAMIC, MOBILE, AND RECONFIGURABLE MULTIAGENT SYSTEMS

THESIS

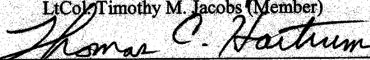
Athie Lee Self, B. S.
Captain, USAF

Approved:

Maj. Scott A. DeLoach (Chairman) — 2 Mar 01 (date)



LtCol Timothy M. Jacobs (Member) — 2 Mar 01 (date)



Dr. Thomas C. Hartrum (Member) — 2 Mar 01 (date)

iv

ACKNOWLEDGMENTS

First on the list to give thanks to, as always, is my Lord and Savior Jesus Christ for giving me the strength to finish this program and thesis and for loving me despite all my faults and failures. I want to thank my wife, Janet, for all her support during this process. I love her with all my heart and even more so after her sacrifice in helping me finishing this work. Next, I would like to thank my advisor Major DeLoach for his guidance and patience with me during this research. Finally, I thank my fellow students for their support and help during the entire time here at AFIT.

Athie Lee Self

TABLE OF CONTENTS

ACKNOWLEDGMENTS .... **V**

TABLE OF CONTENTS .... **VI**

TABLE OF FIGURES .... **XI**

TABLE OF TABLES .... **XVI**

I. INTRODUCTION 1

1.1 Problem Background 1

1.2 Problem Statement 6

1.2.1 Scope 7

1.2.2 Assumptions 7

1.3 Approach 7

1.4 Related Research 8

1.5 Thesis Overview 11

II. PROBLEM APPROACH 12

2.1 Analysis Phase 12

2.1.1 Options 14

2.1.1.1 Move Activity Contained Exclusively Within Concurrent Tasks 14

2.1.1.2 Special Move Role and Task 14

2.1.2 Mobile Agent Search System (MASS) 15

2.1.2.1 Analysis Option 1 15

2.1.2.2 Analysis Option 2 19

2.1.3 Summary 23

2.2 Design Phase 23

2.2.1 Creating Agent Classes 24

2.2.1.1 Agent Component 24
2.2.1.1.1 Step 1: Basic Agent Component..26
2.2.1.1.2 Step 2: Transient Agent Component...27
2.2.1.1.3 Step 3: Persistent Agent Component 28
2.2.1.1.4 Step 4: Transient/Persistent Agent Component 29
2.2.2 Constructing Conversations...30
2.2.3 Assembling Agent Classes 30
2.2.4 Options 30
2.2.4.1 Individual Components Handle Move Functionality 32
2.2.4.2 Mobility Component Handles Move Functionality..33
2.2.4.3 Agent Component Handles Move Functionality..34
2.2.5 MASS System 35
2.2.5.1 Design Option 1 37
2.2.5.2 Design Option 2 40
2.2.5.3 Design Option 3 42
2.3 Selection of Analysis and Design Options for Mobility.. 44
2.4 Summary 44
III. DESIGN 46
3.1 Transformations 46
3.1.1 Transformation Functions..47
3.1.2 Analysis to Design..49
3.1.2.1 Ensuring Mobility Remains in Component..49
3.1.2.2 Agent Component 50
3.1.2.2.1 Agent Component Transform 1 51
3.1.2.2.2 Agent Component Transform 2 52
3.1.2.2.3 Agent Component Transform 3 53
3.1.2.2.4 Agent Component Transform 4 54
3.1.2.2.5 Agent Component Transform 5 55
3.1.2.2.6 Agent Component Transform 6 56

3.1.2.2.7 Agent Component Transform 7 57
3.1.2.2.8 Agent Component Transform 8 58
3.1.2.2.9 Agent Component Transform 9 59
3.1.2.3 Summary 60
3.1.3 Design to Design 60
3.1.3.1 Agent Component Mobility 60
3.1.3.1.1 Agent Component Mobility Transform 1 61
3.1.3.1.2 Agent Component Mobility Transform 2 62
3.1.3.1.3 Agent Component Mobility Transform 3 63
3.1.3.1.4 Agent Component Mobility Transform 4 64
3.1.3.1.5 Agent Component Mobility Transform 5 65
3.1.3.1.6 Agent Component Mobility Transform 6 66
3.1.3.2 Component Mobility 67
3.1.3.2.1 Component Mobility Transform 1 68
3.1.3.2.2 Component Mobility Transform 2 68
3.1.3.2.3 Component Mobility Transform 3 69
3.1.3.2.4 Component Mobility Transform 4 69
3.1.3.2.5 Component Mobility Transform 5 70
3.1.3.2.6 Component Mobility Transform 6 71
3.1.3.2.7 Component Mobility Transform 7 72
3.1.3.2.8 Component Mobility Transform 8 73
3.1.3.2.9 Component Mobility Transform 9 74
3.1.3.2.10 Component Mobility Transform 10 74
3.1.3.2.11 Component Mobility Transform 11 75
3.2 Summary 75
IV. DEMONSTRATION 77
4.1 Travel Planning System (TPS) 77
4.2 MaSE/AgentTool Analysis 78
4.3 MaSE/AgentTool Mobile Design 81

4.3.1 Creating Agent Classes ... 82

4.3.2 Constructing Conversations/Assembling Agent Classes ... 82

4.3.2.1 Create Agent Component ... 83

4.3.2.2 Agent Component Mobility ... 85

4.3.2.3 Component Mobility ... 86

4.4 Carolina Implementation ... 90

4.5 Summary ... 95

V. CONCLUSIONS AND FUTURE WORK ... 96

5.1 Conclusions ... 96

5.2 Future Research ... 97

5.2.1 Specifying Cloning and Instantiation ... 97

5.2.2 Adding Security to Design Model ... 98

5.2.3 Formal Proof of Transformations ... 98

5.2.4 Automatic Code Generation ... 98

5.3 Summary ... 99

A. BACKGROUND ... 100

A.1 Dynamic Agents ... 100

A.2 Dynamic Agent Advantages ... 103

A.2.1 Summary ... 105

A.3 Agent-Oriented Software Engineering (AOSE) Methodologies ... 105

A.3.1 Agent-Oriented Analysis and Design (GAIA) ... 106

A.3.2 Multi Agent Scenario-Based (MASB) ... 107

A.3.3 Multiagent Systems Engineering (MaSE) ... 109

A.3.4 Summary ... 116

A.4 Mobile Agent Platforms ... 117

A.4.1 Concordia ... 118

A.4.2 Telescript/Odyssey ... 119

A.4.3 Aglets Workbench ... 120

A.4.4 Carolina 121

A.4.5 Summary 124

A.5 Summary 125

B. TRANSFORMATION MODELS 126

B.1 Design Model Definitions 126

B.1.1 Agents 127

B.1.2 Components 127

B.1.3 State Tables 128

B.1.3.1 States 129

B.1.3.2 Transitions 130

B.1.3.3 Actions 131

B.1.3.4 Events 132

B.2 Summary 133

BIBLIOGRAPHY 134

VITA 136

TABLE OF FIGURES

FIGURE 1. THE MASE METHODOLOGY [23] 2

FIGURE 2. RELATIONSHIP BETWEEN ANALYSIS AND DESIGN PHASE MODELS IN MASE [18] 4

FIGURE 3. CAROLINA ARCHITECTURE [16] 6

FIGURE 4. ANALYSIS OPTION 1 GOAL HIERARCHY DIAGRAM FOR MASS SYSTEM 16

FIGURE 5. ANALYSIS OPTION 1 USE CASE FOR MASS SYSTEM 16

FIGURE 6. ANALYSIS OPTION 1 SEQUENCE DIAGRAM FOR MASS SYSTEM 17

FIGURE 7. ANALYSIS OPTION 1 ROLE DIAGRAM FOR MASS 17

FIGURE 8. BID TASK FOR MASS SYSTEM 18

FIGURE 9. ANALYSIS OPTION 1 SEARCH TASK FOR MASS SYSTEM 19

FIGURE 10. ANALYSIS OPTION 2 GOAL HIERARCHY DIAGRAM FOR MASS SYSTEM 20

FIGURE 11. ANALYSIS OPTION 2 USE CASE FOR MASS SYSTEM 20

FIGURE 12. ANALYSIS OPTION 2 SEQUENCE DIAGRAM FOR MASS SYSTEM 21

FIGURE 13. ANALYSIS OPTION 2 ROLE DIAGRAM FOR MASS SYSTEM 22

FIGURE 14. ANALYSIS OPTION 2 SEARCH TASK IN MASS SYSTEM 22

FIGURE 15. ANALYSIS OPTION 2 MOVE TASK IN MASS SYSTEM 23

FIGURE 16. MESSAGE PASSING ARCHITECTURE FOR STARTING CONVERSATIONS 26

FIGURE 17. BASIC AGENT COMPONENT DIAGRAM 27

FIGURE 18. AGENT COMPONENT FOR AGENT WITH ONLY TRANSIENT COMPONENTS 27

FIGURE 19. AGENT COMPONENT FOR AGENT WITH ONLY PERSISTENT COMPONENTS 28

FIGURE 20. AGENT COMPONENT FOR AGENT WITH BOTH TRANSIENT AND PERSISTENT COMPONENTS 29

FIGURE 21. SEQUENCE DIAGRAM FOR MOBILE COMPONENT HANDLING MOVE 33

FIGURE 22. SEQUENCE DIAGRAM FOR MOVE COMPONENT HANDLING MOVE 34

FIGURE 23. SEQUENCE DIAGRAM FOR AGENT COMPONENT HANDLING MOVE 35

Figure 24. Analysis Option 1 Agent Class Diagram for MASS System 35

Figure 25. Analysis Option 2 Agent Class Diagram for MASS System 36

Figure 26. Bidder Component From MASS System 37

Figure 27. Design Option 1 Search Component for MASS System 38

Figure 28. Design Option 1 Agent Component for MASS System 39

Figure 29. Design Option 2 Mobility Component for MASS System 41

Figure 30. Design Option 2 Search Component for MASS System 42

Figure 31. Design Option 3 Agent Component for MASS System 43

Figure 32. MaSE Transformation Architecture 46

Figure 33. Example Component After Transformation Defined by [18] 50

Figure 34. Example Component Diagram From Figure 32 After Transformations 28a 50

Figure 35. Agent Component Diagram After ACT1 Transformation 52

Figure 36. Agent Component Diagram After ACT2 Transformation 53

Figure 37. Agent Component Diagram After ACT3 Transformation 54

Figure 38. Agent Component Diagram After ACT4 Transformation 55

Figure 39. Agent Component Diagram After ACT5 Transformation 56

Figure 40. Agent Component Diagram After ACT6 Transformation 57

Figure 41. Agent Component Diagram After ACT7 Transformation 58

Figure 42. Agent Component Diagram After ACT8 Transformation 59

Figure 43. Agent Component Diagram After ACT9 Transformation 60

Figure 44. Agent Component Diagram After ACMT1 Transformation 62

Figure 45. Agent Component Diagram After ACMT2 Transformation 63

Figure 46. Agent Component Diagram After ACMT3 Transformation 64

Figure 47. Agent Component Diagram After ACMT4 Transformation 65

Figure 48. Agent Component Diagram After ACMT5 Transformation 66

Figure 49. Agent Component Diagram After ACMT6 Transformation 67

Figure 50. State and Transition Added to Non-Mobile Components by CMT2 69

Figure 51. State and Transition Added to Mobile Components by CMT3 69

Figure 52. States Added by CMT4 for Mobile Components 70

Figure 53. Transitions Added to Mobile Components by CMT5 71

Figure 54. Transitions Altered in Mobile Components by CMT6 72

Figure 55. State and Transition Added to a Component by CMT7 73

Figure 56. Transition Added to Mobile Components by CMT8 73

Figure 57. Transition Altered in Components by CMT9 74

Figure 58. Transition Added to a Component by CMT10 75

Figure 59. MaSE Role Model for TPS System 78

Figure 60. Gather Travel Information Task for TPS System 79

Figure 61. GetFlightReservation Task for TPS System 80

Figure 62. AcceptFlightReservations Task for TPS System 81

Figure 63. Agent Classes for TPS System 82

Figure 64. Transformation Menu in agentTool 83

Figure 65. Components of InterfaceUser Agent Class for TPS System 84

Figure 66. Agent Component for InterfaceUser Agent Class 84

Figure 67. Agent Component for Plan Travel Agent Class 85

Figure 68. Agent Component for PlanTravel Agent Class in TPS System After Agent Component Mobility Transformations 86

Figure 69. GetFlightReservation Component after Transformations Defined by [18] 87

Figure 70. Informational Dialog before Selecting States to Receive Move Required Messages 88

Figure 71. State Selection Window 89

Figure 72. GetFlightReservation Component After Component Mobility Transformations 89

Figure 73. Travel Planner Graphical User Interface for TPS System 91

Figure 74. TPS System Output Showing Agents Starting the Travel Planning Process 92

Figure 75. PlanTravel Agent at First Address 93

FIGURE 76. PLANTRAVEL AGENT AT SECOND AND FINAL ADDRESS 94
FIGURE 77. FINAL RESULTS FROM TPS SYSTEM 95
FIGURE 78. FIPA SIMPLE MOBILITY PROTOCOL [6] 103
FIGURE 79. THE GAIA METHODOLOGY [25] 106
FIGURE 80. EXAMPLE ROLES MODEL SCHEMA [25] 107
FIGURE 81. FILL PROTOCOL DEFINITION [25] 107
FIGURE 82. THE MASE METHODOLOGY [23] 109
FIGURE 83. GOAL HIERARCHY DIAGRAM [5] 110
FIGURE 84. SEQUENCE DIAGRAM [5] 111
FIGURE 85. GOALS MAPPED TO ROLES [5] 111
FIGURE 86. REGISTRATION TASK [4] 112
FIGURE 87. MASE ROLE MODEL [5] 113
FIGURE 88. AGENT CLASS DIAGRAM [23] 114
FIGURE 89. MASE COMPONENT DIAGRAM [15] 115
FIGURE 90. DEPLOYMENT DIAGRAM [23] 116
FIGURE 91. CONCORDIA ARCHITECTURE [22] 118
FIGURE 92. CAROLINA ARCHITECTURE [16] 121
FIGURE 93. AGENT STATES IN CAROLINA [16] 124
FIGURE 94. EXAMPLE GRAPHICAL REPRESENTATION OF A TYPE [18] 126
FIGURE 95. CLASS DIAGRAM FOR THE TYPES USED IN THE DESIGN PHASE OF MASE [18] 127
FIGURE 96. AGENT TYPE [18] 127
FIGURE 97. COMPONENT TYPE [18] 128
FIGURE 98. STATETABLE TYPE [18] 128
FIGURE 99. STATETABLE CLASS DIAGRAM [18] 129
FIGURE 100. STATE TYPE [18] 129
FIGURE 101. TRANSITION TYPE [18] 131
FIGURE 102. ACTION TYPE [18] 132

FIGURE 103. FUNCTIONCALL TYPE [18] 132

FIGURE 104. PARAMETER TYPE [18] 132

FIGURE 105. EVENT TYPE [18] 132

FIGURE 106. RECEIVEEVENT TYPE [18] 133

TABLE OF TABLES

TABLE 1. AGENT COMPONENT TRANSFORMATIONS 51

TABLE 2. AGENT COMPONENT MOBILITY TRANSFORMATIONS 61

TABLE 3. COMPONENT MOBILITY TRANSFORMATIONS 67

AFIT/GCS/ENG/01M-11

ABSTRACT

Multiagent Systems use the power of collaborative software agents to solve complex distributed problems. There are many Agent-Oriented Software Engineering (AOSE) methodologies available to assist system designers to create multiagent systems. However, none of these methodologies can specify agents with dynamic properties such as cloning, mobility or agent instantiation.

This thesis starts the process to bridge the gap between AOSE methodologies and dynamic agent platforms by incorporating mobility into the current Multiagent Systems Engineering (MaSE) methodology. Mobility was specified within all components composing a mobile agent class. An agent component was also created that integrated the behavior of the components within an agent class and was transformed to handle most of the move responsibilities for a mobile agent. Those agent component and component mobility transformations were integrated into agentTool as a proof-of-concept and a demonstration system built on the mobility specifications was implemented for execution on the Carolina mobile agent platform.

DESIGN & SPECIFICATION OF DYNAMIC, MOBILE, AND RECONFIGURABLE MULTIAGENT SYSTEMS

I. Introduction

1.1 Problem Background

Imagine a platoon leader on a battlefield that needs to have detailed battlespace information at a critical point in the mission. Instead of having to stop the mission to search the global information grid and analyze the information needed, the leader could send out dynamic software agents. These agents would travel to the servers holding the data needed, analyze the data on those servers to compile the requested information and then travel back to the leader to report their findings. All the while the leader is freed to direct the soldiers in their current environment while waiting on the required information in order to move forward to complete the mission.

Joint Vision 2020 [9] and Air Force Vision 2020 [20] stress that decision superiority is the next major battleground and that the armed forces need to control that battleground. Ongoing multiagent system research in the field of artificial intelligence may provide the solution to gaining and maintaining decision superiority to support the warfighters in this century.

True multiagent systems bring the power of multiple collaborative software agents to solve complex distributed problems. However, the current Agent-Oriented Software Engineering (AOSE) methodologies used to design these multiagent systems have mainly focused on agents without the properties of mobility, cloning or agent instantiation that may be critical to solving problems faster and more efficiently. Many researchers have also developed numerous dynamic agent platforms to handle agents with the properties mentioned above but with no multiagent design methodology to go along with the platform.

AOSE differs from the object-oriented methodology in terms of the agent-oriented abstractions used to model a system. These abstractions include agent-oriented decomposition, characterizing complex systems by subsystems, interactions between agents, and mechanisms for agents to flexibly form, maintain, and disband various organizational structures. Some current AOSE methodologies include Agent-Oriented Analysis and Design (GAIA) [25], Multi Agent Scenario-Based (MASB) [13] and Multiagent Systems Engineering (MaSE) [23].

The Air Force Institute of Technology (AFIT) Agent Research Group has been focused on developing and maturing an AOSE methodology. The MaSE methodology [23], as shown graphically in Figure 1, has been designed to cover the entire life cycle of developing and implementing a multiagent system. MaSE has an analysis phase that consists of the following three steps: capturing goals, applying use cases and refining roles. The design phase of MaSE consists of the following four steps: creating agent classes, constructing conversations, assembling agent classes and system design. Graphical models are used in each of the seven MaSE steps to assist the designer in the process.

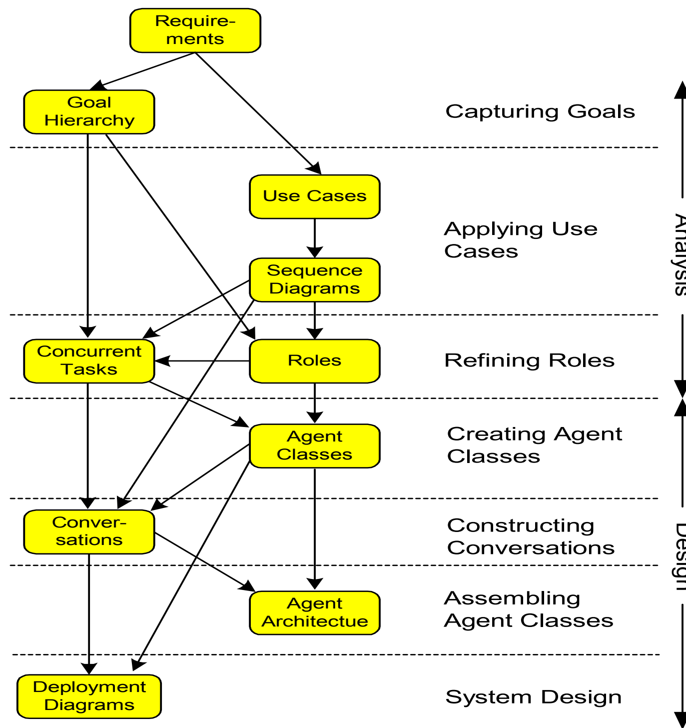


Figure 1. The MaSE Methodology [23]

Defining a methodology specifically for formal agent system synthesis has been the main goal of the AFIT Agent Research Group. In order to accomplish this synthesis, the analysis models in MaSE must be transformed into the design models. These design models, in turn, must then be transformed into executable software code. In his thesis, Sparkman [18] developed the necessary transformations to convert MaSE analysis models into design models. In order to build the transformations, the semantics of the analysis and design models were defined and a mapping between the models was created [18].

The MaSE analysis phase models consist of roles that agents will play and the concurrent tasks that define the behavior of the roles and the coordination between those roles. These analysis phase models map to the following design phase models [18], as shown in Figure 2. Components existed in the former MaSE design architecture [15] but there was no guidance on how to use them to build agents or coordinate conversations between those agents.

A component within an agent class, in the new design architecture, is created in an agent's internal architecture from each task that belongs to a role that the agent is playing. Since components correspond to the concurrent tasks, and those tasks were assumed to execute under their own thread of control, the components must also execute under their own thread of control. The coordination for passing internal events between components of the same agent class and the coordination of the conversations between different agent classes that was contained in the state tables of the concurrent tasks remains intact in the component state tables.

States and transitions that compose the conversations are extracted from the state diagram of a component and replaced by an action on a transition that represents the execution of the conversation [18]. This is not the only way to model the organizational structure of agents, components, and conversations in the design phase but it does captures all of the information that is present in the analysis models while maintaining multiagent framework independence by retaining the basic idea of a conversation.

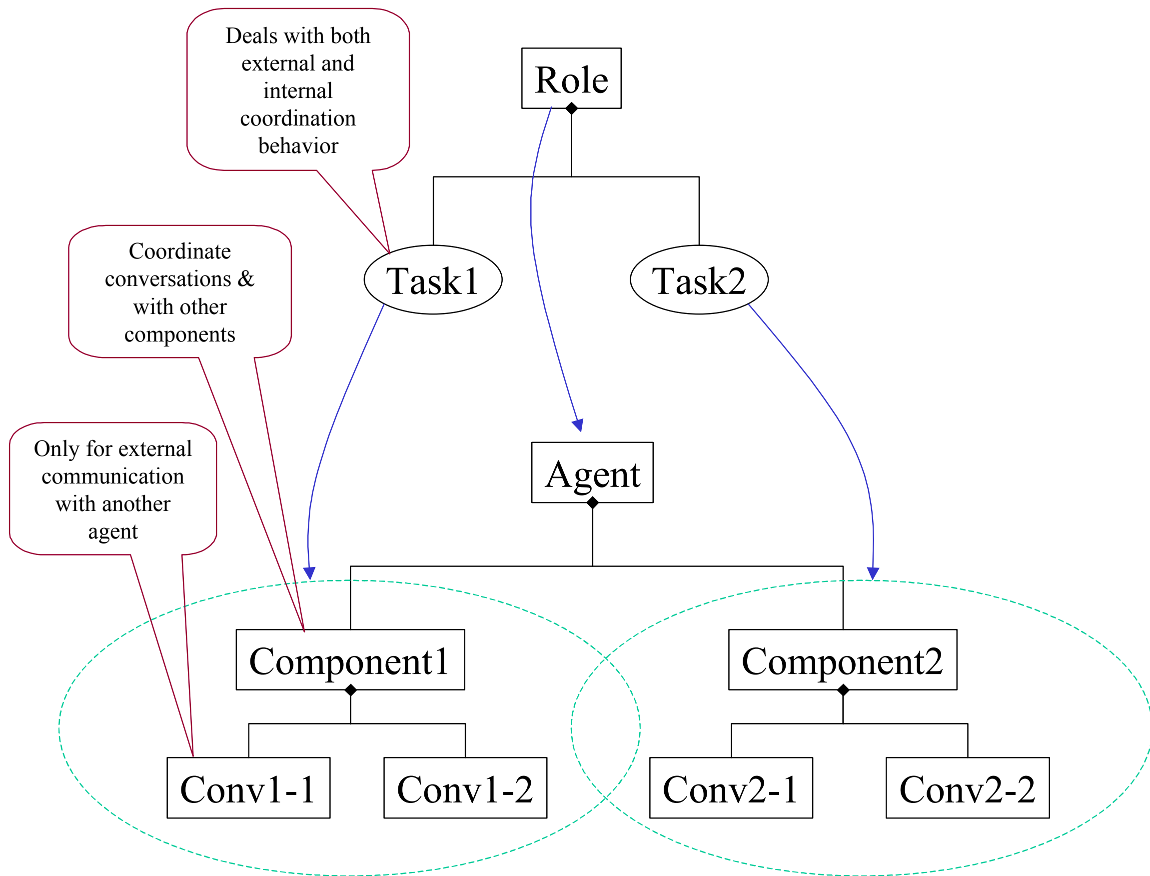


Figure 2. Relationship Between Analysis and Design Phase Models in MaSE [18]

Sparkman's transformation system consists of three stages. The inputs to the system are the completed Role Model, Concurrent Task Diagrams and assignment of roles to agent classes. In the first stage, the components for each agent class are created from the tasks, the protocols for external events are determined, protocols between components are replicated in the design and converting external events to internal events if appropriate. Stage two activities include labeling the start and end of conversations in the component state diagrams and matching up conversation start events between the components. Finally, in stage three, the conversations are harvested from the component state tables. States and transitions belonging to each half of the conversations are extracted from the component state diagrams and are replaced with a single transition and action that represents the execution of the conversation.

Currently, however, agents designed using MaSE cannot be dynamic because they cannot clone themselves, are not mobile and do not instantiate other types of agents. Dynamic agent systems have shown promise in being able to solve certain network related problems such as finding services and information on the World Wide Web (WWW). These systems have shown an advantage in robustness of functionality over other client-server interaction solutions, such as Remote Procedure Calls (RPC), messaging and sockets. *Dynamic* agents are agents that possess the following properties:

- Cloning – the ability of an agent to create another instance of itself
- Instantiation or spawning – the ability of an agent to create instances of another type or class of agent other than itself
- Mobility – the ability of an agent to move from machine to machine in a network

These three agent properties, or traits, have been the focus of new research in the distributed artificial intelligence arena with the property of mobility receiving the most attention.

Once a multiagent system has been analyzed and designed utilizing dynamic agents, an agent platform is required to handle the execution of those agents. An *agent platform*, by general definition, is a software system that has the ability to create, name, dispatch and terminate agents [1]. The Foundation for Intelligent Physical Agents (FIPA) [6] has defined an agent management reference model, which outlines many logical components that can be included in any physical implementation of an agent platform. A few examples of mobile agent platforms are: Concordia [22], Telescript/Odyssey [21], Aglets Workbench [10] and Carolina [16].

The Carolina platform (Figure 3) is currently under development at the University of Connecticut as part of the Multi-Agent Distributed Goals Satisfaction project. AFIT, the University of Connecticut, and Wright State University are conducting this research jointly. Three of the logical components listed by FIPA are provided by Carolina: the Agent Execution Environment, the Agent Management System and an Agent Communication Channel. Carolina's Execution Container is the agent execution environment and provides the place where the agent code executes. An AgentManager and AgentDirectory handle all agent management within Carolina. Agent management deals with handling the agent moves between platforms

and maintaining relevant information about the agent. An agent communication channel allows agents to exchange information between one another concerning services and communication messages. The MessageManager provides that channel within Carolina. As mentioned above Carolina provides a framework for handling dynamic agents. However, there is no multiagent design methodology that one can use to develop agents that will execute in the Carolina environment.

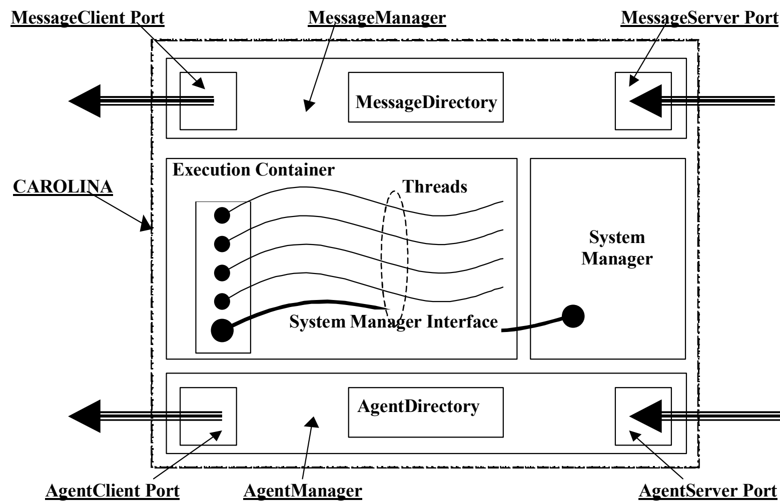


Figure 3. Carolina Architecture [16]

1.2 Problem Statement

As was described above, multiagent systems bring the power of multiple collaborative software agents to solve complex distributed problems. However, the current Agent-Oriented Software Engineering (AOSE) methodologies used to design these multiagent systems have mainly focused on agents without the properties of mobility, cloning or agent instantiation that may be critical to solving problems faster and more efficiently. Adding these properties to AOSE methodologies gives those methodologies even more power to solve complex problems. Therefore, the problem addressed in this thesis is as follows:

> Modify the appropriate phases, steps and models of the current Multiagent Systems Engineering methodology to allow for the design and specification of multiagent systems using dynamic agents.

1.2.1 Scope

Even though many AOSE methodologies exist, only the MaSE methodology was modified to incorporate the specification and design of dynamic agents. The MaSE methodology was used because it has been developed at AFIT and is a complete multiagent system design methodology as opposed to other AOSE methodologies like GAIA [25].

The mobility aspect of dynamic agents was the only one of the three dynamic properties incorporated into MaSE because of the research time limitation. A simple demonstration system was designed and implemented using the Carolina Server software provided by the University of Connecticut as the Agent Platform (AP) for the dynamic agents [16]. This server software was used for this research because it was readily available and it supported the use of mobile agents. The Carolina Server software was not modified in any way to support this thesis.

1.2.2 Assumptions

The MaSE methodology and or the Carolina server changed during the course of this research due to ongoing improvements and research by other master students either at AFIT or the University of Connecticut. These changes were accounted for before deciding on any modifications to MaSE to support this research effort.

1.3 Approach

The following steps were taken to solve the problem of incorporating the concepts of dynamic agents into the MaSE methodology:

1. Defined the properties of dynamic agents for the purposes of this thesis.

2. Examined different options for integrating the mobility property into the analysis phase of MaSE. The advantages and disadvantages were explored for each option and the best option overall was selected.

3. Examined different options for integrating the mobility property into the design phase of MaSE. Again, the advantages and disadvantages were outlined and the best option overall was selected.

4. Developed transformations from the analysis phase models that incorporated mobility into design phase models in MaSE according to the methods chosen above.

5. Demonstrated the feasibility of the problem solution in MaSE by analyzing, designing and implementing a multiagent system using mobile agents in Carolina.

1.4 Related Research

Building high quality software for real world applications is one of the most difficult construction tasks facing humans today [1]. The number and complexity of real world components and the relationships between those components makes modeling the real world difficult. Real world problems are also distributed, dynamic and heterogeneous. Many different approaches have been tried in the discipline of software engineering to tackle the task of modeling complex, distributed systems including the object-oriented and more recently, agent-oriented software engineering methodologies.

Agent-oriented methodologies, as the name implies, are methodologies for building software systems that use agents. An *agent* is a software system that is [24]:

- Autonomous – not controlled directly by humans or other agents
- Cooperative – agents communicate amongst themselves to help achieve goals
- Perceptive – agents perceive, react and can make changes to their environment.
- Pro-active – agents are not passive entities like objects. They exhibit goal-directed behavior.

Agents are a natural extension of objects from object-oriented methodologies. Agents differ from objects mainly in respect to exhibiting flexible autonomous behavior and operating in their own thread of control.

Most current intelligent systems consist of a single agent [19]. However, a single agents' capacity to solve problems is limited by its knowledge, perspective and computing resources. Real world problems are generally too large, complex and unpredictable to be handled by a single agent. Some artificial

intelligence researchers have realized the limitations of a single agent and have focused work on multiagent systems. These systems can be executed on one machine or can be distributed across multiple networked machines. Distributed multiagent systems bring the power of modularity in the form of flexible agent-based organizations to handle the complexity and size of real world problems. Multiagent systems can also provide efficient solutions to real world problems where resources or expertise is distributed.

The GAIA, MASB and MaSE methodologies introduced in Section 1.1 all cover the entire lifecycle of multiagent systems design. All three methodologies are similar because they are based on the view of a multiagent system being a computational organization consisting of various interacting roles. In each methodology an agent can play one or many roles at any given time. Agents within these methodologies interact through conversations with each other.

These methodologies differ mainly in terms of the amount of detail they provide to build multiagent systems. MaSE and MASB provide much more detail for defining conversations than GAIA. MASB is the only methodology that emphasizes human/agent interactions. Only MASB and MaSE provide models for building the internal knowledge structures for agents. MaSE is the only methodology that specifies the locations of the agents in the final system. One major shortfall that exists in all three of the methodologies described above is the ability to analyze and design a multiagent system where the agents can be dynamic. Appendix A contains a more detailed analysis of these methodologies.

The properties of dynamic agents, namely mobility, cloning and instantiation, can be thought of as extensions of the general definition of an agent. Dynamic agents posses tremendous network functionality and could replace current network protocols such as Remote Procedure Calling (RPC) and messaging. In fact, if all the functionality provided by a dynamic agent system is taken together, there is no single alternative that can provide that same level of functionality [2]. Advantages of using dynamic agents include [2][7][11]:

1) They can reduce communication and bandwidth costs
2) They can be hardware and operating system independent depending on the mobile agent platform they execute on

3) They can be fault tolerant

4) They can maintain an optimal configuration

However, there are three main reasons why dynamic agent technology has not replaced Remote Procedure Calls and messaging. The first deals with the lack of a specific network function that can be accomplished only by the use of dynamic agents. Everything that a dynamic agent system can do can be done using other network protocols [2]. The second reason is that RPC and messaging have been used successfully for many years. Finally, dynamic agent technology is still in its infancy. Many issues still need to be worked out including security and standards/interoperability. There are many dynamic agent systems (mostly focusing on the mobility aspect) already in industry but there is very little interoperability to date. FIPA and the Object Modeling Group have begun to develop standards, but they have not been widely accepted.

Dynamic agents, as opposed to static agents, need certain capabilities provided by their execution environment. FIPA has defined an agent management reference model that defines many of these capabilities or logical components. Dynamic agent systems, such as Concordia, Telescript/Odyssey, Aglets Workbench and Carolina, have at least the following three components: Agent Platform, Agent Management System and Agent Communication Channel. All four of these platforms were written in Java to provide hardware and operating system independence.

Most of the differences between the platforms are either in functionality provided, the way in which agents communicate or how agent mobility is handled. Concordia provides the most functionality in terms of the FIPA logical components but takes away some of the autonomous ability of the agents. Concordia requires each agent to have a fixed itinerary and uses the Agent Manager to deal with resending agents to an unresponsive platform. Carolina forces all agent-to-agent communication through the server. Telescript/Odyssey is the only platform where the agent moves by calling a command within the agent system itself. The other platforms have a hierarchy of methods that form an agent life cycle with a "job" method that once ended specifies a location for agent travel. Appendix A contains a more detailed analysis of these platforms.

1.5 Thesis Overview

This document is organized as follows. Chapter II discusses the different options for integrating mobility into the analysis and design phases of MaSE, where advantages and disadvantages of each option are discussed. Chapter III begins with the selection of the best options from the choices presented in Chapter II. Then, transformations, based upon those selections, are developed to transform the mobile analysis models into mobile design models. Chapter IV demonstrates the use of a multiagent system, analyzed and designed using MaSE, using the solution described in Chapters II and III. Chapter V completes this thesis by summarizing the contributions of this thesis effort and describes future research that could expand upon and contribute to the work started in this thesis.

II. Problem Approach

Incorporating dynamic agents into any multiagent design methodology entails studying each phase of the methodology to determine how the concepts should be added. This chapter describes the approach taken to add mobility concepts to the analysis and design phases of MaSE.

For illustration and understanding, an example system will be analyzed and designed using the different options presented in the following sections. Section 2.1.2 introduces the sample problem that will be used to illustrate the differences in the options presented. This sample problem will also be used in Chapter III.

2.1 Analysis Phase

All three of the MaSE analysis phase steps, Capturing Goals, Applying Use Cases and Refining Roles, were examined for possible inclusion of mobility. In the Capturing Goals step the overall goals for the system are extracted from the initial system context. These goals should not contain detailed information. An example of a goal might be, "Find Requested Information". The goal is to find the information that is requested and does not include the detailed definition of how to find that information, which may change with time.

Mobility concepts fall more into the detailed process of how an agent behaves in order to fulfill its goal. An agent might not have to move in order to fulfill its goals or find requested information as in the example above. However, a sub-goal of the previous goal could be "Move to Information Source". The functionality of moving is not described in this sub-goal just the requirement to move. Thus, mobility could, but does not have to be, included in the overall system goals.

Mobility would pertain to the Applying Use Cases step only if mobility was a system goal. Since roles are responsible for system goals and the goals can include mobility, then there could be a mobility role in the system. On the other hand, if the goals do not include mobility then there should not be a

mobility role. These two possibilities form the basis of the analysis options presented in subsequent sections.

Even if mobility was not included in the two previous steps, mobility was already an option in the Refining Roles step. As described in Appendix A, the MaSE methodology uses *concurrent tasks* to define internal agent behaviors and the conversations between agent classes [134]. Concurrent tasks operate within their own thread of control and define the decision process for actions taken by the roles. Activities within those tasks represent the actual functions that a role is to perform. There are currently a number of predefined activities available to include in a task. One such activity is the *move* activity that moves an agent to a new address. A single Boolean variable is the result of the move activity. The Boolean result represents whether the move actually occurred. The syntax for the move activity is shown below.

```
Boolean = move(location)
```

However, having only a simple Boolean result from the move action is not adequate to allow agents to properly reason about mobility. An explanation, or reason why the move failed, is required to provide the agent with enough flexibility to recover from moving failures. Different reasons for failure to move include: the mobile agent platform on the destination address is not operational, the machine that the agent is moving from is isolated from the rest of the network, the mobile agent platform denies the move because of security or other reasons, etc. Therefore, a new move activity was defined that returns two values: a Boolean value and a Reason value. The Boolean value again represents whether the move actually occurred while the Reason variable denotes the reason for failure, if applicable. The syntax for the new move activity is shown below.

```
<Boolean,Reason> = move(location)
```

If, however, a designer does not wish to know or use the reason for a moving failure then the original move activity can still be used.

2.1.1 Options

There are two possible options for specifying mobility in the MaSE analysis phase. The first option is to specify mobility only within concurrent tasks in the Refining Roles step. Whereas the second option is to allow mobility to be specified within all steps of the analysis phase but with the requirement to use a special move task and role in the Refining Roles step. Both options are discussed in detail and then illustrated by the MASS system in the following sections.

2.1.1.1 Move Activity Contained Exclusively Within Concurrent Tasks

The first possible option to specifying mobility in the MaSE analysis phase is to ignore mobility until the Refining Roles step. The concept of mobility would be confined to using the special move activities discussed above in the concurrent diagrams defining tasks. A *mobile task*, then, is a task that contains a move activity. In contrast, a *non-mobile* task is a task that does not contain a move activity.

The first, and primary, advantage for using this approach is that it only requires the addition of a more robust move activity to the existing concurrent task model. A second advantage for using this approach is that it allows more flexibility in the design of the system. This will be more evident when the design options are discussed in following sections. Finally, by keeping the concept of mobility self-contained within individual tasks, any tasks attached to a role with a mobile task do not have to be altered to deal with mobility, however, those tasks will have to be modified in the design phase.

2.1.1.2 Special Move Role and Task

Placing either form of the move activity into a special move task under a separate move role is the second possible option for incorporating mobility into the analysis phase. Currently, the MaSE role model architecture does not allow for same task being placed under multiple roles [18]. Therefore, the special move task would be placed under a separate move role and combined with other roles in the design phase. Protocols would then be established from any other task in the system (requiring mobility) to that move task.

Reusability and modularity are the main advantages for using this approach. Whenever any role in a multiagent system needs mobility the predefined mobility task and role can be inserted with appropriate protocols defined from that role to the Move task under the Move role.

There is one main disadvantage to this option. Protocols defined between each task requiring mobility and the special move task would need to be added to the MaSE Role Diagram. Thus making the Role Diagram more complicated.

2.1.2 Mobile Agent Search System (MASS)

Now, an example multiagent system is introduced that will illustrate the two options discussed above. This system will be analyzed by using the current MaSE methodology with the addition of the move activity discussed in Section 2.1.

Users have vast amounts of information at their disposal distributed across their organization's data storage infrastructure. As these technology users perform their jobs, they are required to research certain topics as they generate documents. Having to search each pertinent document or database to find required information would be a daunting and very time consuming task. To solve this problem a multiagent system could be developed to assist these users in searching. The goal of these agents is to search through the organization's data stores, compile the results and report that information to the user. To illustrate each analysis option, this problem will now be taken through each step in the MaSE analysis phase.

2.1.2.1 Analysis Option 1

Figure 4 shows the MaSE goal hierarchy diagram for the Mobile Agent Search System (MASS) for analysis option one. Notice that mobility is not listed within any of the system goals. Next, the roles for the MASS system are initially defined by the Applying Use Cases step. A use case that defines the overall operation of the MASS system is shown in Figure 5. Figure 6 shows the corresponding sequence diagram for that use case.

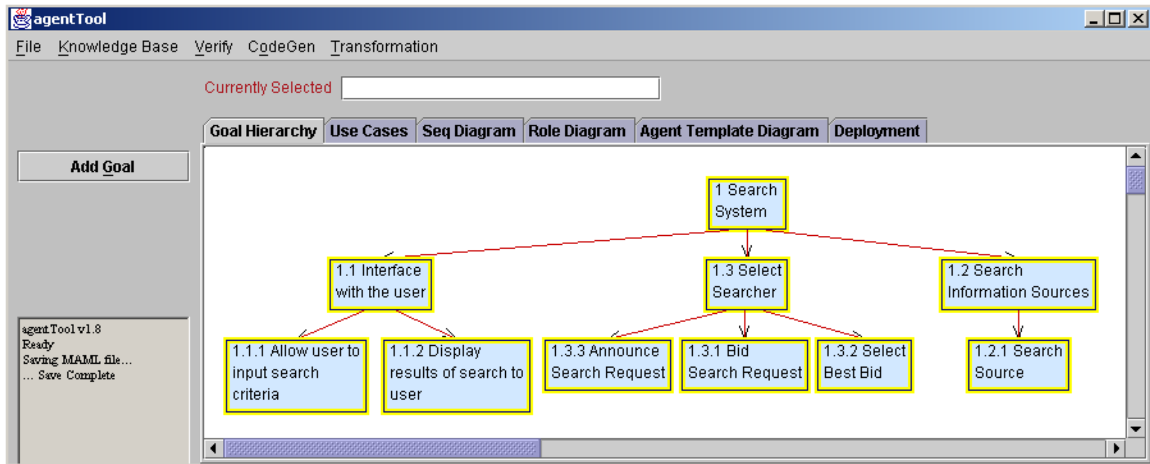


Figure 4. Analysis Option 1 Goal Hierarchy Diagram for MASS System

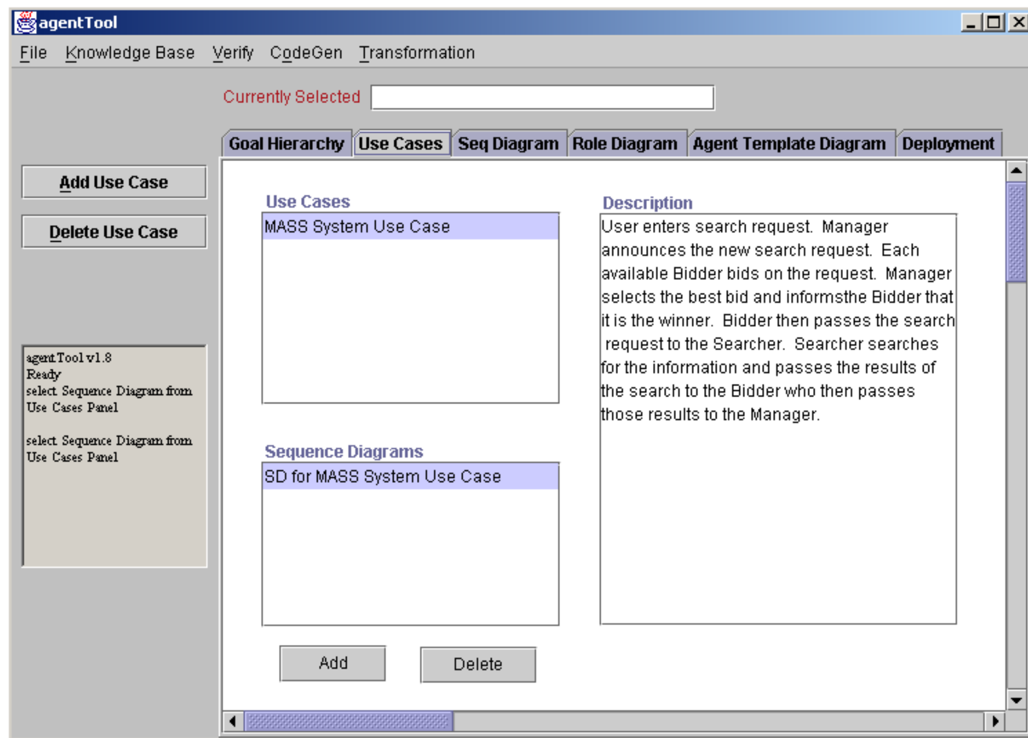


Figure 5. Analysis Option 1 Use Case for MASS System

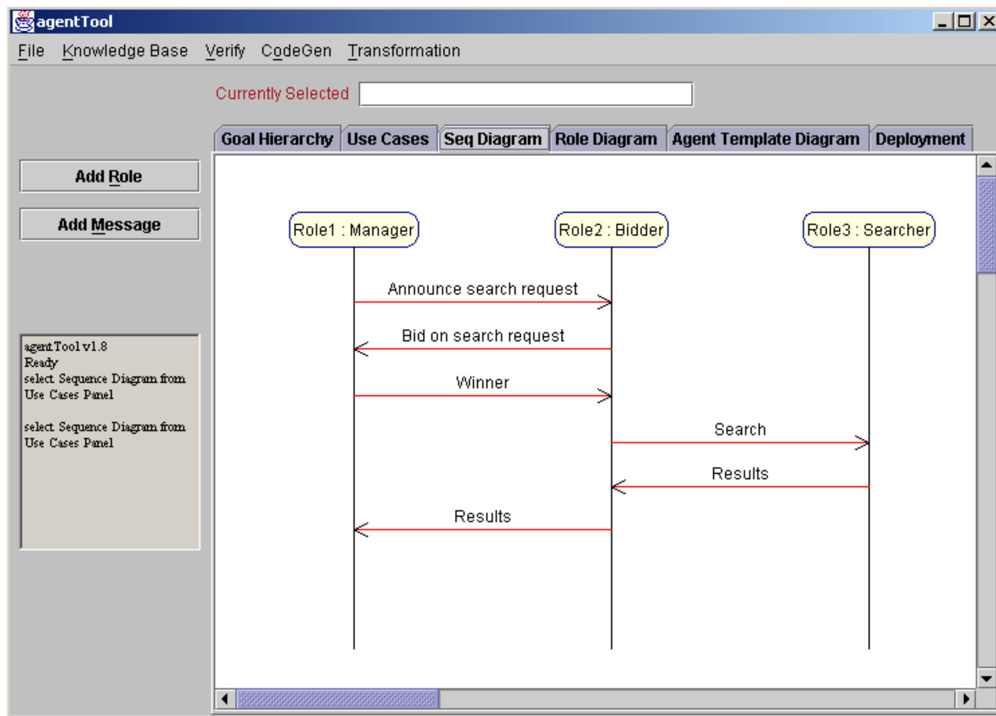


Figure 6. Analysis Option 1 Sequence Diagram for MASS System

Figure 7 shows the analysis option one MaSE role diagram for the MASS. The Manager role announces tasks to the Bidder role using the Contract Net protocol. Once the Bidder role has won the bidding process for a search request, it sends the request to the Search role using the Search Request protocol. The Search task under the Search role searches for the requested information and reports the results back to the Bidder role, which then forwards it back to the Manager role.

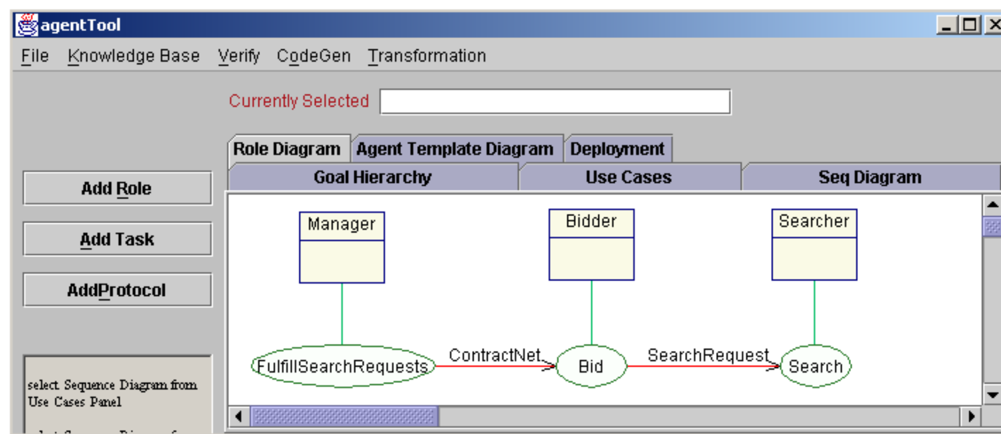


Figure 7. Analysis Option 1 Role Diagram for MASS

Finally, defining the concurrent tasks is the last part of the Refining roles step. Figure 8 shows the Bid task for the MASS system. This task starts in an idle state and begins processing only after receiving an announcement of a task from the Manager. Thus, this task is a persistent reactive task. Once receiving the announcement, the Bidder role decides whether to bid on the task in the prepareBid state. The Bid task transitions back to the idle state to wait for more announcements if either the task is not bid on or the task was bid but the bid was rejected. If the Manager accepts the bid, the Bidder role sends the search task to the Search role. The Search role replies with the results of the search whether successful or not.

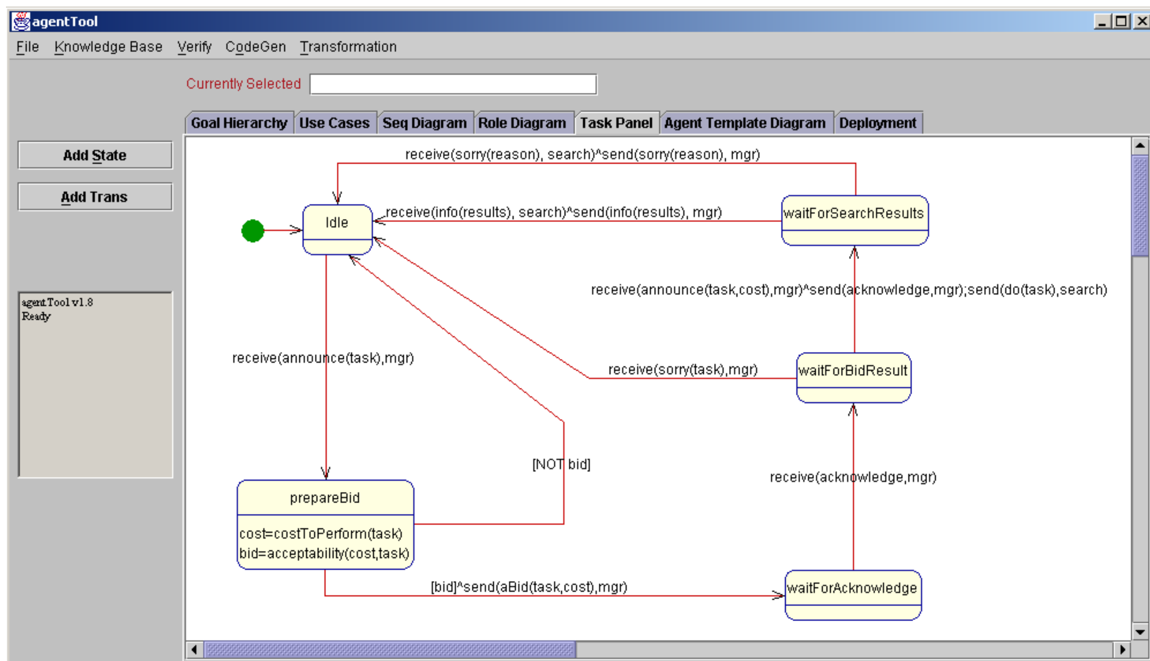


Figure 8. Bid Task for MASS System

The designer determines that the Search role needs the ability to move in order to find the requested data. Figure 9 shows the complete Search task that includes a move activity. The task is started by receiving a "do(task)" message from the Bidder role. In the moveNeeded state, the searchDestination activity takes the task as input and returns the location where the data source is located, while the checkLocation activity returns where the current address of the agent. The compare activity takes the destination and current addresses and determines whether the agent needs to move.

If the agent does not need to move, the find activity in the search state executes and returns the results and a reason for failure, if applicable. If the agent does need to move, the move activity in the tryMove state is executed. If the move is successful then the find activity is executed in the search state as described above. If the move is unsuccessful, a message is sent back to the Bid task with the reason for the failure. This completes the analysis of the MASS using analysis option one.

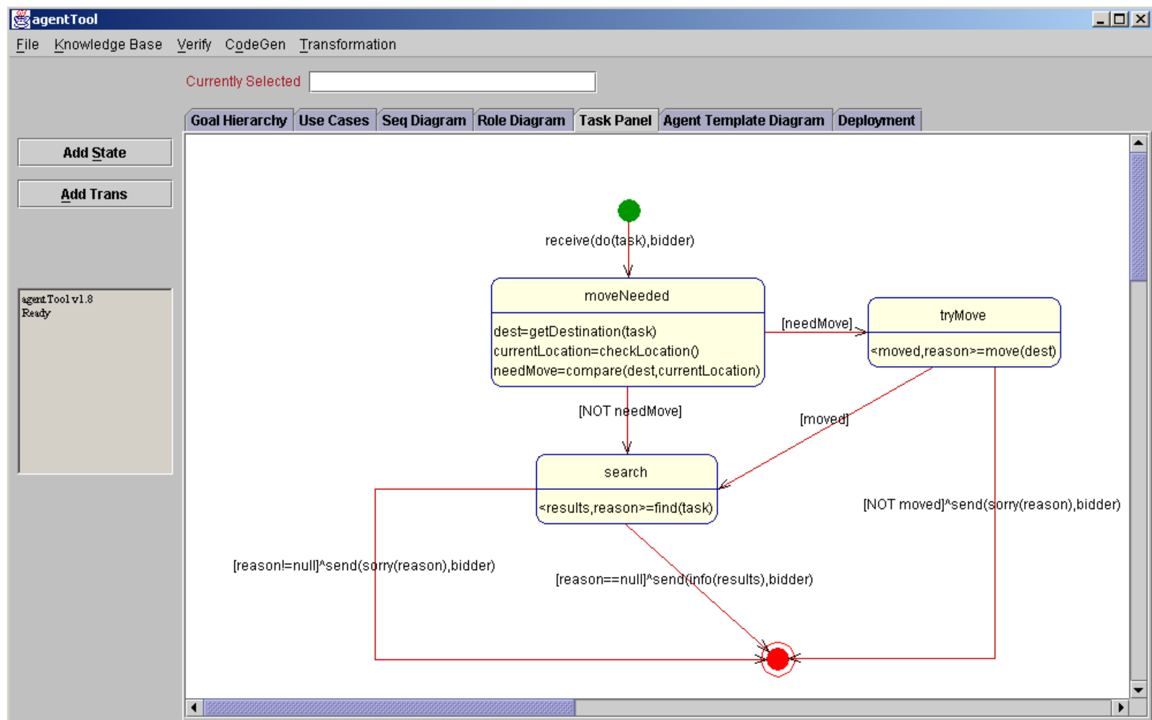


Figure 9. Analysis Option 1 Search Task for MASS System

2.1.2.2 Analysis Option 2

Analysis option two incorporates mobility starting in the Capturing goals step. The only difference between the Goal Hierarchy Diagram in Figure 4 and the one shown in Figure 10 is the inclusion of the "Move to Source" goal.

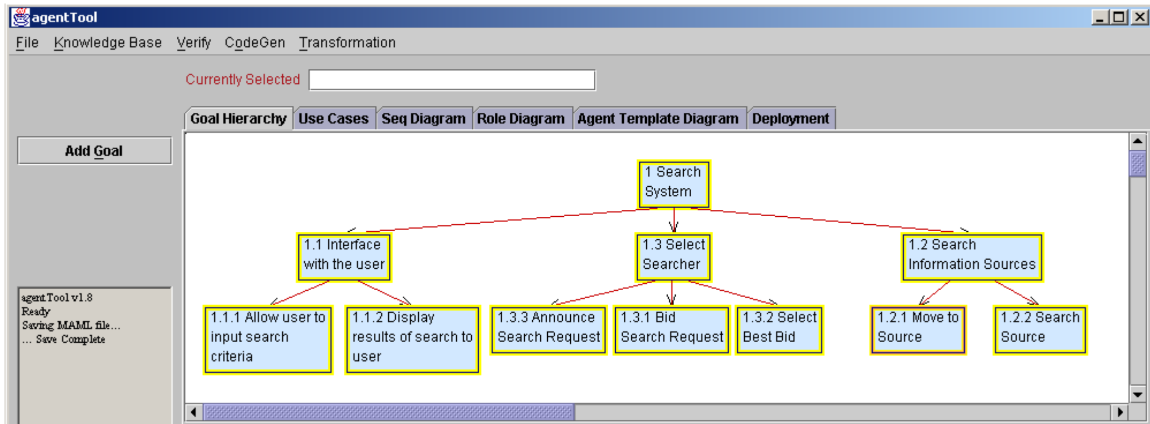


Figure 10. Analysis Option 2 Goal Hierarchy Diagram for MASS System

Next, the use case and sequence diagram for analysis option two also include mobility. The Searcher role sends a destination to a Mobile Agent role, which sends back confirmation of the move. Figure 11 shows the use case while Figure 12 shows the corresponding sequence diagram.

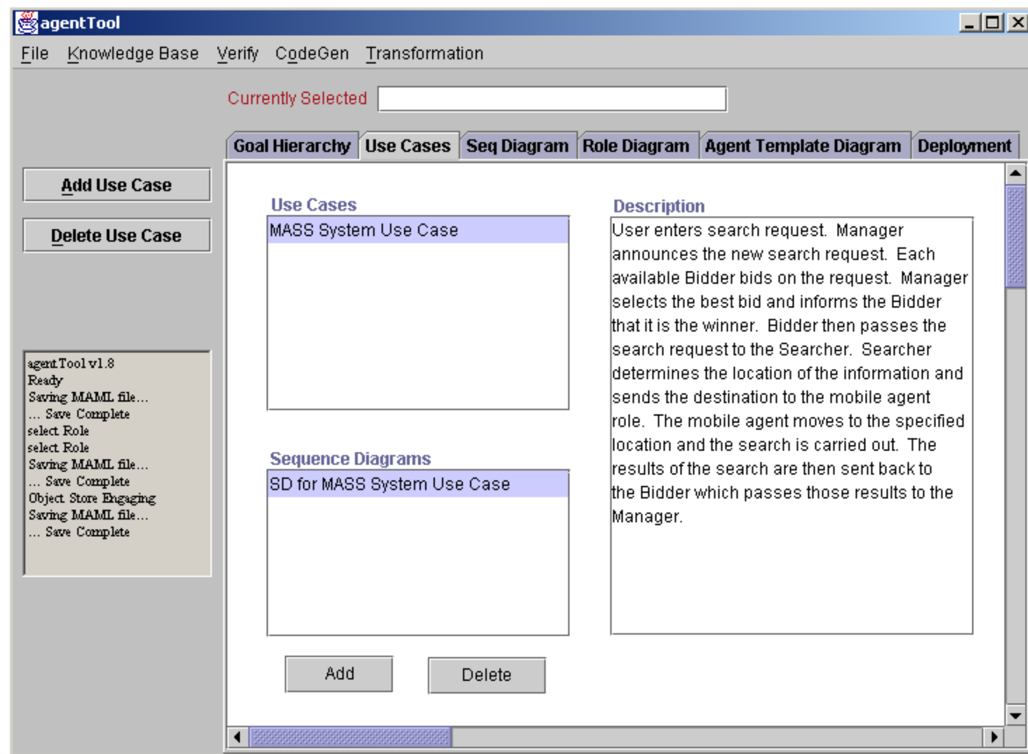


Figure 11. Analysis Option 2 Use Case for MASS System

The expanded MaSE role diagram for the MASS system for this option is shown in Figure 13. The difference from analysis option one is the addition of the mobile agent role and associated move task. In this case, after the Search task under the Search role determines that a move is needed a message containing the destination is sent to the move task. The move task uses the either of the previously defined move activities to perform the move.

The Bid task is the same as in the previous section. Changes to the Search task to call the move task instead of using a move activity are shown in Figure 14. In the Search task, when the determination to move is made, in the moveNeeded state, a "go" message containing the destination is sent to the Move task. The Search task will receive back either a success message or a sorry message containing the reason for the failure from the Move task.

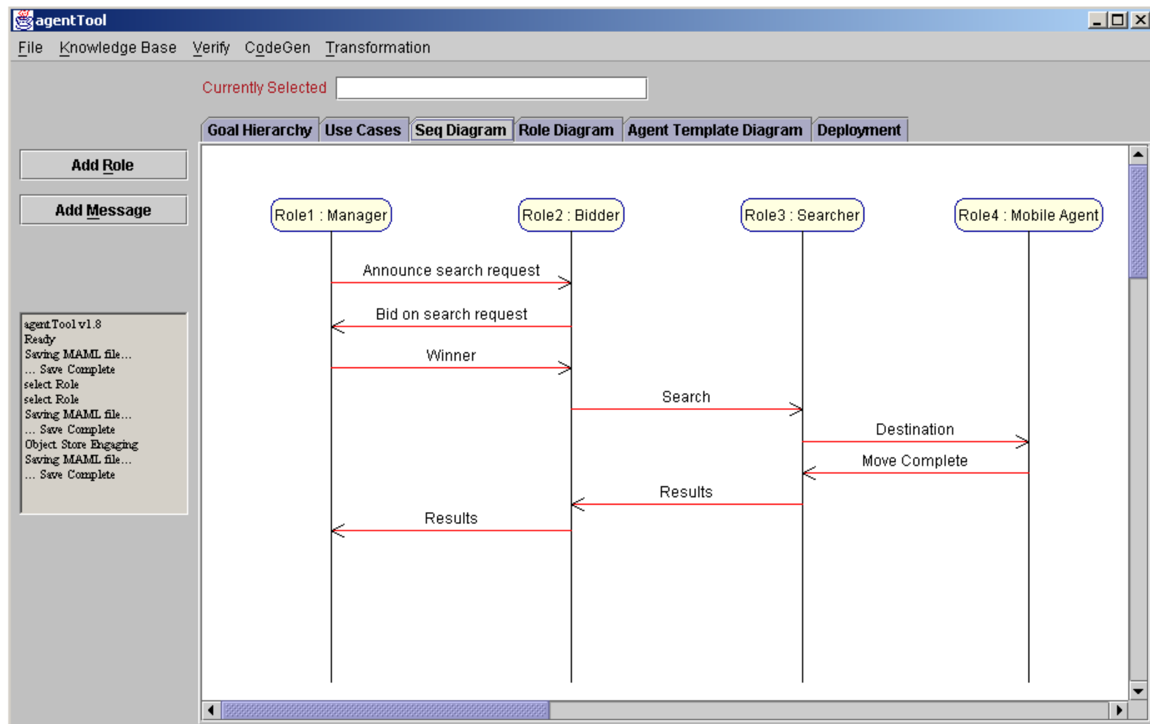


Figure 12. Analysis Option 2 Sequence Diagram for MASS System

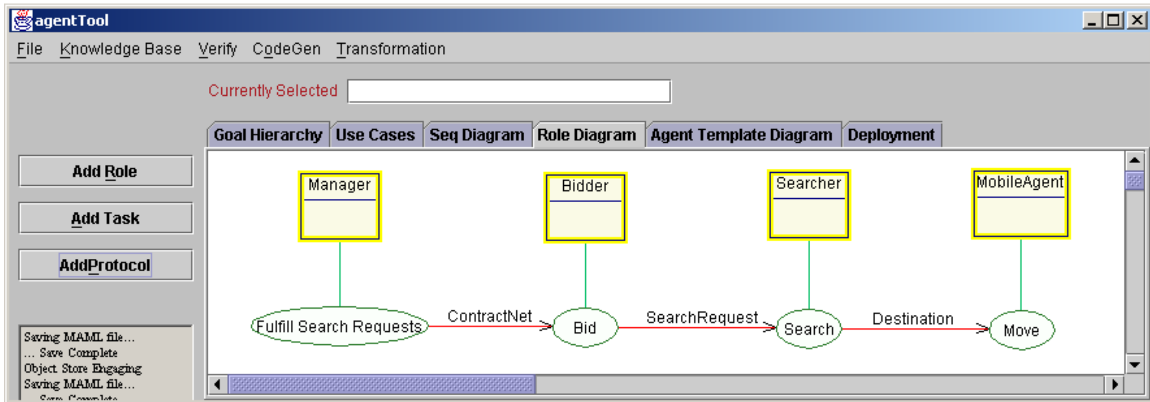


Figure 13. Analysis Option 2 Role Diagram for MASS System

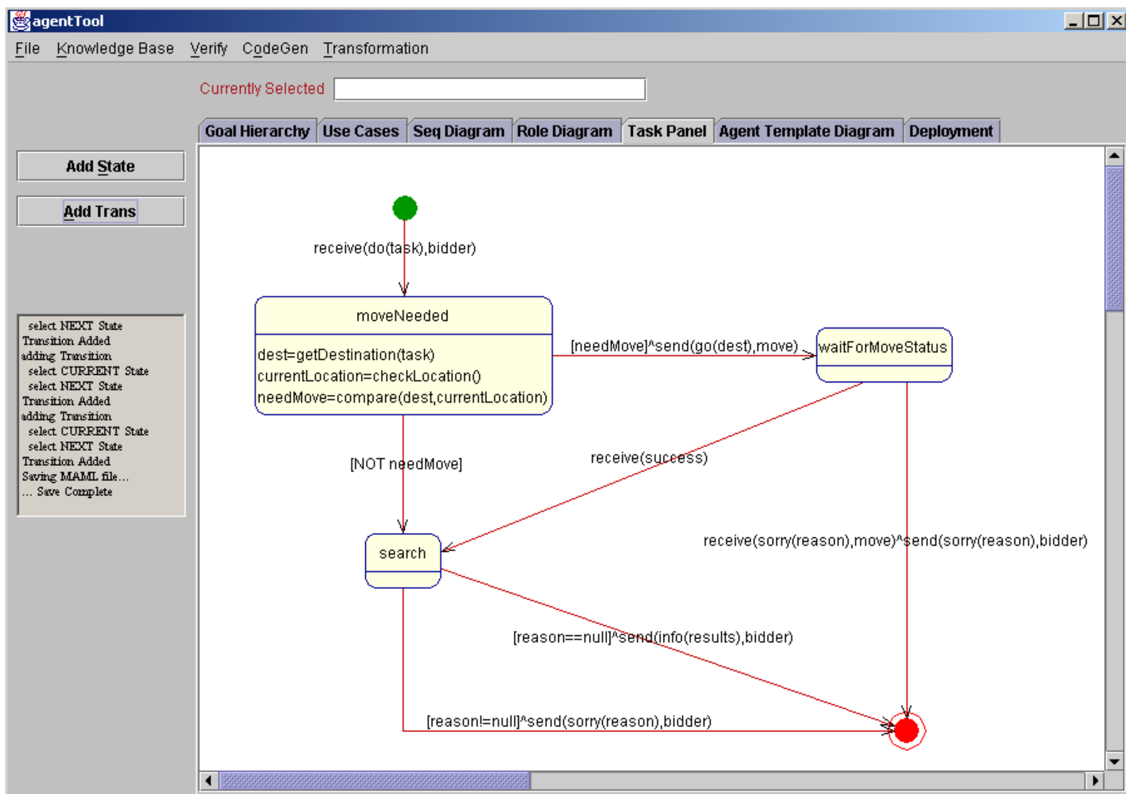


Figure 14. Analysis Option 2 Search Task in MASS System

The functionality for the Move task under the Mobile Agent role is basically taken out of the Search task in the first analysis option and is shown in Figure 15. The receive message contains the parameter role which is generic so any other task that needs mobility just needs a protocol defined between

that task and the move task. This completes the MaSE analysis for the MASS system using analysis option two.

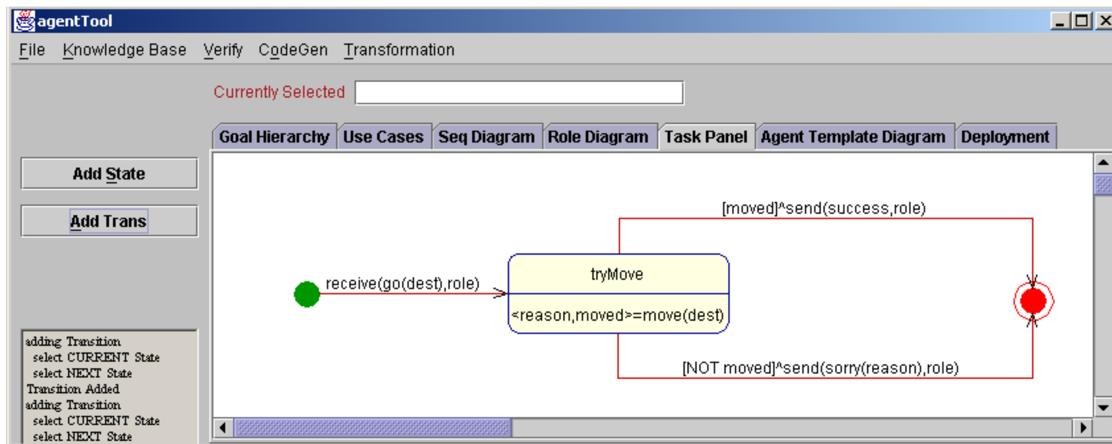


Figure 15. Analysis Option 2 Move Task in MASS System

2.1.3 Summary

Adding mobility to the analysis phase of MaSE can possibly stretch across all three steps or be confined to just one. If mobility goals are included in the Capturing Goals step then it will also be included in the Applying Use Cases and Refining Roles steps. However, mobility concepts can also be confined only to the Refining Roles step with no loss of system definition. The best option will be chosen along with the best option for incorporating mobility into the design phase at the end of this chapter.

2.2 Design Phase

As was the case with incorporating mobility into the analysis phase, each of the four design steps was reviewed for possible impacts due to mobility. There were no changes to the Creating Agent Classes, Constructing Conversations or System Deployment steps with respect to mobility. Changes were made, however, in the Assembling Agent Classes step. After the changes were made, three possible options for incorporating mobility in the MaSE design phase were discussed.

2.2.1 Creating Agent Classes

In the Creating Agent Classes design step, the roles in the system are placed into agent classes. Each agent class can consist of one or more roles. If analysis option 1 were chosen then there would be no changes to this step. However, if analysis option 2 were chosen, then the special move role would have to be incorporated into every agent class that contained a role with a task that had a protocol defined between it and the move task in the analysis phase.

Before analyzing the Constructing Conversations and Assembling Agent Classes steps, changes that were made to the MaSE agent architecture and conversations need to be addressed [18]. These changes necessitated the creation of an agent component, which is discussed in the following section.

2.2.1.1 Agent Component

In the former MaSE agent architecture, each agent consisted of a group of components with connectors between the components and from those components to the environment. But there was no defined mapping between the tasks and protocols defined in the analysis phase and the components that represented the actual functionality of the agent class in the design phase.

Tasks, within MaSE, are categorized by their *life span* and *responsiveness*. Task life spans are either persistent or transient. *Persistent tasks* always have a *null* transition from the start state to the first state. These tasks are started when the agent is created and run until either the task or agent terminates. *Transient tasks*, however, always have a trigger event on the transition from the start state. These tasks are not started upon agent creation but only when the agent receives its trigger event. Transient tasks make it possible to have multiple concurrently executing tasks of the same type.

There are three types of task responsiveness: reactive, proactive or heterogeneous. *Reactive tasks* can be either persistent or transient. A *persistent reactive task* has a null transition from the start state to an idle state, where it remains until it receives a triggering event from the agent. On the other hand, a *transient reactive task* must receive a triggering event from the agent before it can begin processing. A

proactive task continuously generates requests for other agents or tasks, is always persistent and does not contain any idle states. And finally, a *heterogeneous task* is a combination of a proactive and reactive task. It is persistent, but does not start in an idle state and must generate at least one request for another task or agent before entering an idle state.

In the new MaSE agent architecture defined by Sparkman [18], a component is created for each task that is part of a role in an agent class. Each component is classified as reactive, proactive or heterogeneous depending on the type of the task from which it was created [3]. Since concurrent tasks were assumed to operate under their own thread of control, now each component is also assumed to operate in the same way. Thus, a *reactive agent* is an agent with only reactive tasks while a *proactive agent* is an agent with at least one proactive or heterogeneous task.

Because of the fact that an agent might only have transient components, the agent itself has to exist in order to receive an external request by another agent that starts a transient component. This problem necessitated creating an *agent component*, which is started upon agent creation, controls the initiation of the other components, and handles initial conversation messages received from other agents in a system. Figure 16 shows the new message passing architecture using agent components to start new conversations.

Creating the agent component will be accomplished in four separate steps. In step 1, the basic agent component, with states and transitions that are common to all agents, is created. In step 2, the agent component is completed for agents with only transient components. In step 3, the agent component is completed for agents with only persistent components. And finally, in step 4, the agent component is completed for agents with both transient and persistent components.

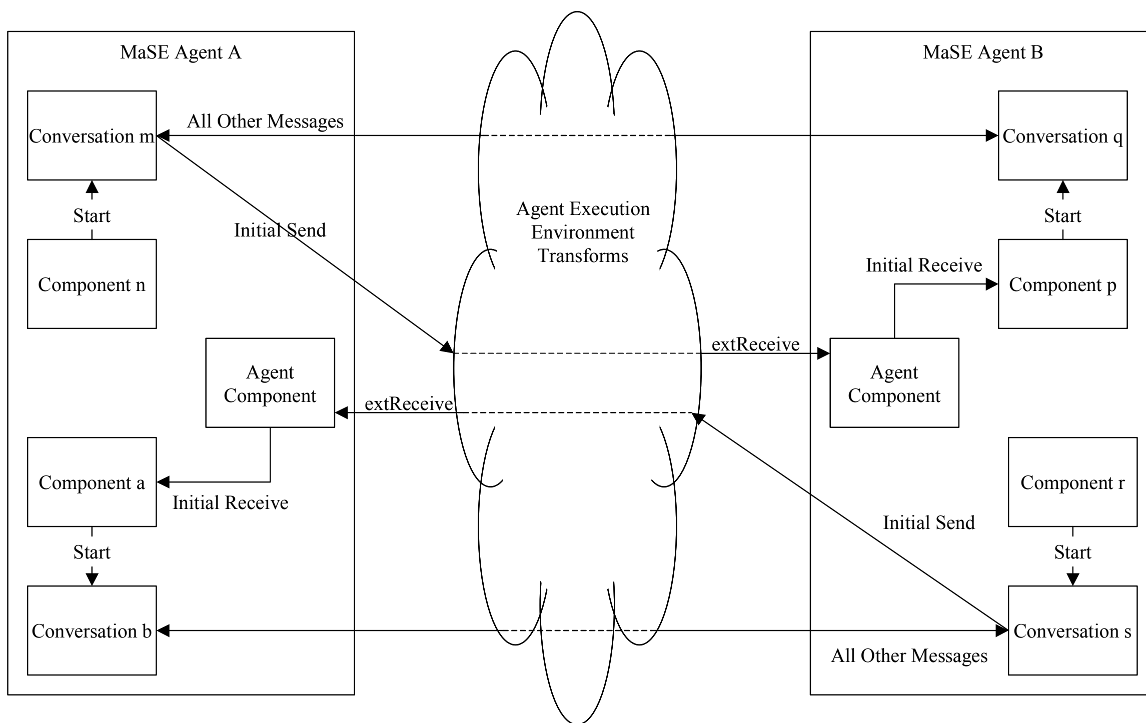


Figure 16. Message Passing Architecture for Starting Conversations

2.2.1.1.1 Step 1: Basic Agent Component

The basic MaSE agent component is shown in Figure 17. The parts of this component are the same regardless of the combination of transient and persistent components that an agent class possesses. All messages from other agents that initiate conversations are handled in the agent component as shown by the transition from the idle state to the determineRecipient state. The getComponent activity takes the received message and determines which component, either currently executing or needing to be started, is the recipient.

Determining which component should receive or which component needs to be started with a message could be ambiguous. The solution to this problem is left to future research. If the component is already running then the relay action forwards the message to the component. The agent can also receive a terminate message from its owner in which case the agent will terminate along with any components that are still operating.

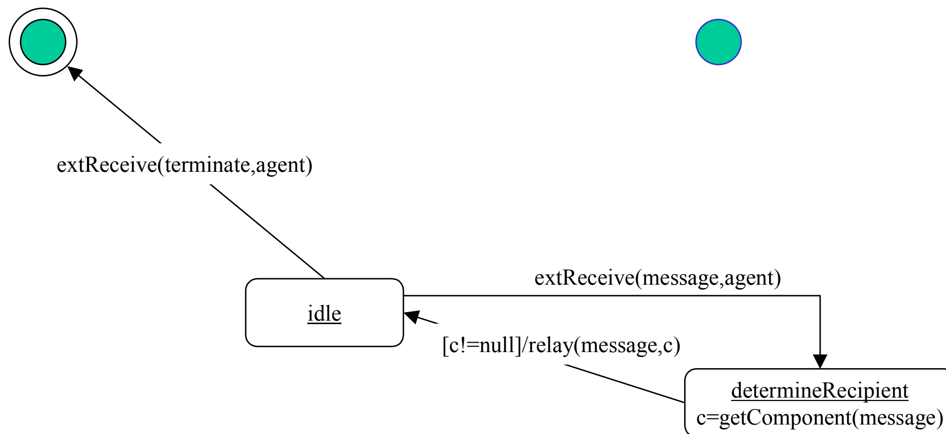


Figure 17. Basic Agent Component Diagram

2.2.1.1.2 Step 2: Transient Agent Component

A transient agent component is created if an agent class only consists of transient components and there are no move activities specified in any of those components. The additions to the basic agent component to support this case are shown in bold in Figure 18.

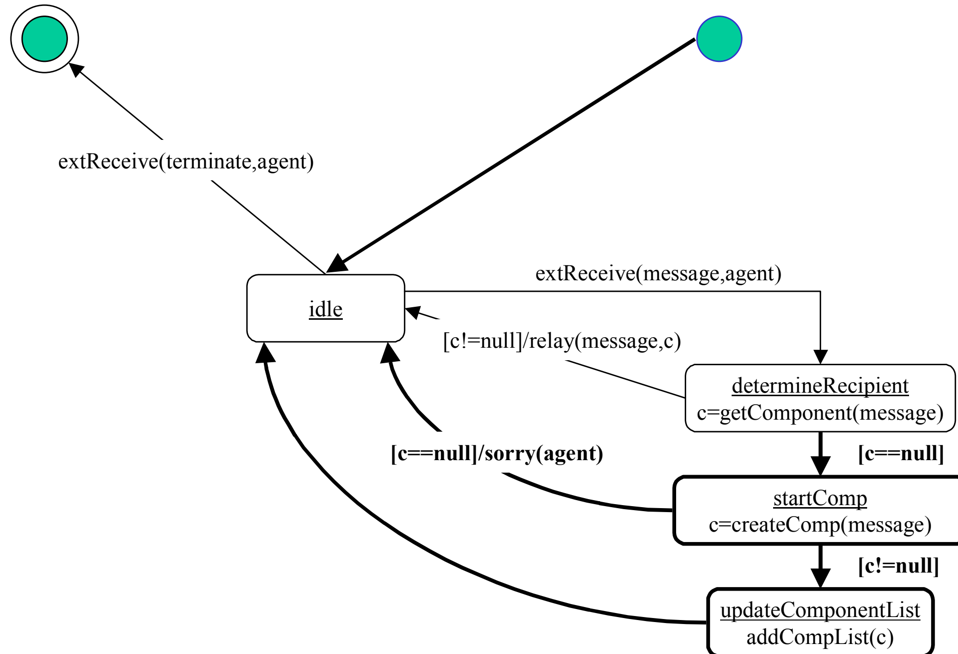


Figure 18. Agent Component for Agent with only Transient Components

A null transition is added from the start state to the idle state because transient components are only started by receipt of an external or internal message. The startComp state was added with the createComp activity to handle the case when a message does not belong to an existing component. Every component upon startup is added to the list of active components by the addCompList activity in the updateComponentList state. If a received message is not the event that starts a transient component, then a "sorry" conversation is started with the agent that sent the message.

2.2.1.1.3 Step 3: Persistent Agent Component

A persistent agent component is created when an agent class contains only persistent components with no move activities specified. Additions to the basic agent component to support this case are shown in bold in Figure 19. The startPersistentComps state containing the startComps activity was added along with a null transition from the start state to the startPersistentComps state because all persistent components are started when an agent is created. In case there is an error starting the components, the transition from the startPersistentComps state to the end state is triggered. If the components are started without error, then the transition from the startPersistentComps state to the idle state is triggered. This transition contains an action with the function setTimer that takes as input from the designer a period of time and sets a timer.

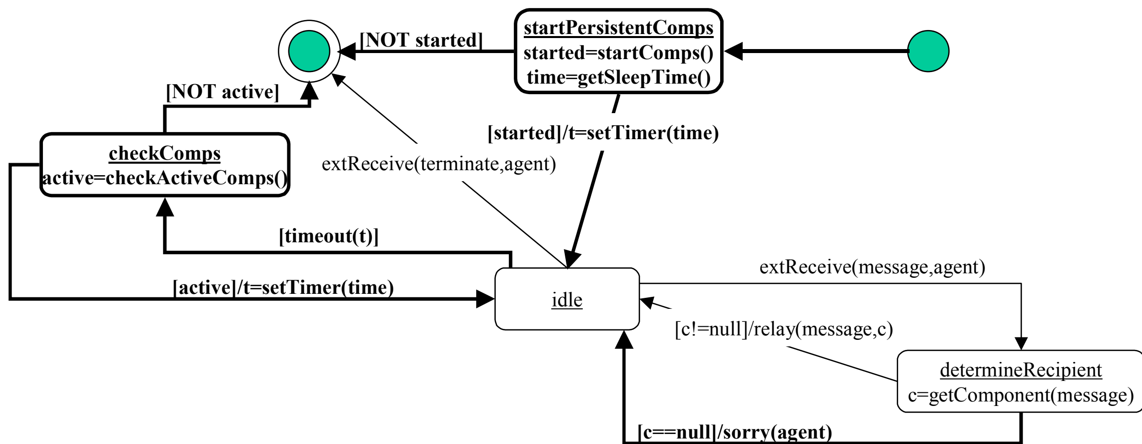


Figure 19. Agent Component for Agent with only Persistent Components

Once the time interval specified has passed, the transition from the idle state to the checkComps state is triggered. The checkActiveComps activity within the checkComps state determines if at least one

component is still processing. If one or more components are still processing, the transition from the checkComps to the idle state is triggered and the timer is reset. If all components have ceased processing either normally, or with an error, the agent's work is complete and the transition from the checkComps to the EndState is triggered.

Checking the status of the persistent components allows the agent to terminate normally, even in the case of component failure. Finally, the transition from the determineRecipient state to the idle state was required to handle the case of an external message that does not belong to an existing component.

2.2.1.1.4 Step 4: Transient/Persistent Agent Component

A transient/persistent agent component is created when an agent class consists of both transient and persistent components with no move activities specified. The state transition diagram for this option is basically the same as an agent with only transient components with the addition of the startPersistentComps state. The agent component for this combination is shown in Figure 20.

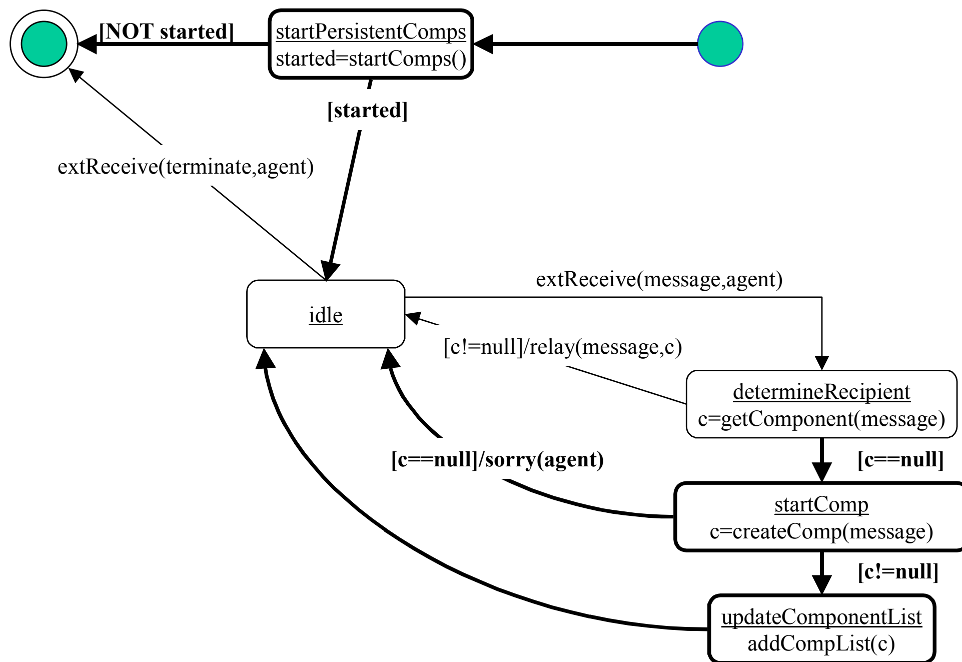


Figure 20. Agent Component for Agent with both Transient and Persistent Components

The timer logic was not added because an agent containing even one transient component is considered to process forever or until a terminate message is received from its owner. An agent containing this component can be labeled either reactive or proactive.

2.2.2 Constructing Conversations

Once the agent component has been constructed, the analysis of the Constructing Conversations and Assembling Agent Classes steps for mobility impacts can be accomplished. A conversation is taken from the state diagram of a component and replaced in that diagram with an action on a transition that represents the entire execution of the conversation [18]. Since conversations are only supposed to model agent-to-agent interaction, allowing a conversation to call for a move was not permitted. Restricting move activities to components also eliminated the complexity of interrupting conversations by calling for a move and then having to reconstruct a conversation into the proper state after a move. Thus, ensuring that move activities defined within components are not placed within a conversation was the only addition needed in the Constructing Conversations design step.

2.2.3 Assembling Agent Classes

Significant changes to the components created in the Assembling Agent Classes step were also needed. For either analysis option 1 or 2, the move activities specified in each are transformed directly into the corresponding state in the diagram for the component [18]. A *non-mobile component* is a component of a mobile agent that does not include a move activity in any of its states. Thus a *mobile component* is a component of a mobile agent that includes a move activity in at least one of its states. In the case of analysis option 2, only the move component created from the special move task would be a mobile component.

2.2.4 Options

The three options discussed below for incorporating mobility into the MaSE design phase, are as follows:

1. Mobile components handle the move responsibilities.

2. A separate mobility component handles the move responsibilities.

3. The agent component created in Section 2.2.1 is modified to handle the move responsibilities.

Each of these options is discussed in detail in the following sections. Before discussing the options, however, a list of moving requirements that each option has to address needs to be presented. The following activities at a minimum need to be performed in order for a move to be successfully completed:

1. Each active component needs to be informed that a move has been requested by a mobile component. If an agent contains only one persistent mobile component then that component does not need to receive its own move request message. However, if an agent contains a single transient mobile component then that component does need to receive its own move request message since there can be many instances of that transient mobile component executing at the same time [3]. Finally, if an agent contains two mobile components of any type, all components need to be able to respond to a move request message.

2. Once informed of a move request, each component needs to save the work it was performing and terminate without error. This implies that each component must be able to save its current state and then terminate. Furthermore, if a non-mobile task did not have an end state then an end state will have to be added to the component that is created from that task.

3. After all components have terminated the agent can then move to another address.

4. Once the agent has moved, all components that were active need to be restarted. This requirement implies that functionality needs to be added to restart a component into the correct state at the new address.

Additional activities add robustness and fall under the autonomous nature of agents:

5. Each Component decides whether it can stop current processing and move. There are two possible responses to a move call:

    a. A component can decide that a move would not allow it to complete its goals. In this case the component would deny a move request.

    b. A component can decide that a move would not hinder it from completing its goals. In this case the component would accept a move request.

6. Every component that calls for a move must be able to handle both of the responses to a move call listed above.

Each design option fulfills the requirements listed above but varies with respect to which component handles those requirements.

2.2.4.1 Individual Components Handle Move Functionality

In this option, each mobile component is transformed to handle the moving responsibilities for it and for the entire agent. These mobile components need to satisfy the requirements discussed above as well as handle the interface between the agent and the mobile agent system in which the agent is executing.

An advantage to this approach is that only minor changes are required for the agent component and other non-mobile components. Most all of the moving logic is contained within the move capable components. A disadvantage is that each component with move capability has to be aware of all active components so it can notify those components that a move is occurring. This communication overhead is shown in Figure 21. Another disadvantage would be the duplication of move functionality in the components that require mobility. Each move capable component would have the exact same code for conducting a move. A final disadvantage is that in this approach a move capable component appears to have control over the other components. Having control over the components should be kept within the notion of an agent rather than in one of its components.

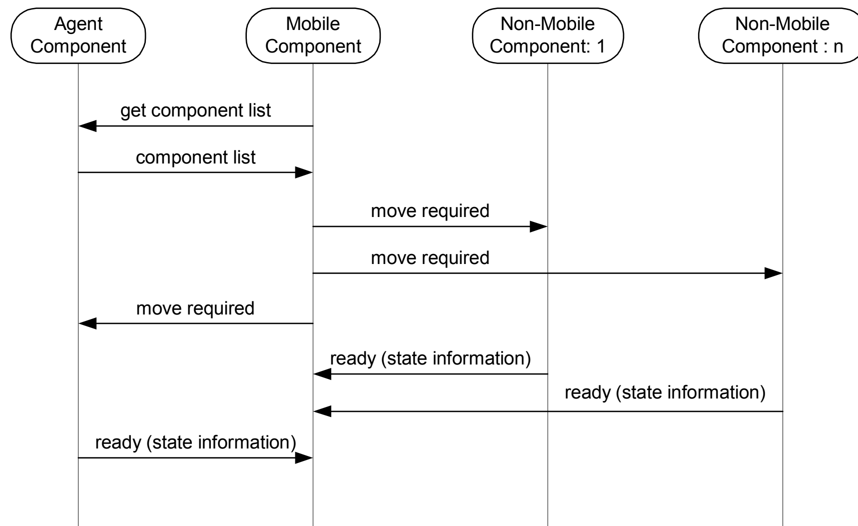


Figure 21. Sequence Diagram for Mobile Component Handling Move

2.2.4.2 Mobility Component Handles Move Functionality

In this option, a special move component is created to handle the moving responsibilities for an entire agent. As was the case with the first design option, this move component needs to satisfy the requirements discussed above as well as handle the interface between the agent and the mobile agent system in which the agent is executing.

This option is an obvious result of following analysis option 2. The separate move role and task would be combined into an agent class with roles with move requirements. This special move task would then be transformed into a special move component in the design phase. This move component would handle the moving responsibilities for the agent much like a mobile component in the previous design option.

An advantage to this approach is that there is no duplication of move functionality in the components. Most of the move functionality is contained in the move component. One disadvantage to this approach is increased communication. The move component, like a mobile component in design option 1, has to get information about all other components from the agent component after an initial move request is received from another component. This results in adding one additional message to the overall

move process as presented in design option 1. The sequence diagram showing the communication overhead is shown in Figure 22.

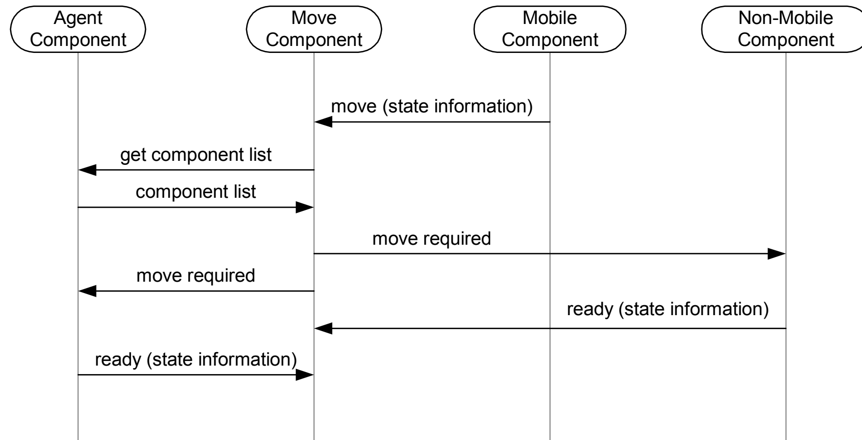


Figure 22. Sequence Diagram for Move Component Handling Move

2.2.4.3 Agent Component Handles Move Functionality

In this option, the agent component is transformed to handle the move responsibilities for the entire agent. As was the case with the first two design options, the agent component needs to satisfy the requirements discussed above as well as handle the interface between the agent and the mobile agent system in which the agent is executing.

One advantage to this option is reduced communication when activating the move process compared to design options 1 and 2 as shown in Figure 23. The reduced communication comes from the fact that the two messages required for a component to receive information about other components are eliminated. The agent component already has access to all other component information and can easily send out “move required” messages to those components. Another advantage is that there is no duplication of move functionality in each mobile component as in design option 1. All of the moving functionality is handled by the agent component. This option is also more centralized than either design option 1 or 2. The one disadvantage to this option is that it is less modular than option 2.

This concludes the discussion of the design options. Each of these options will be illustrated by using the MASS system in the following sections.

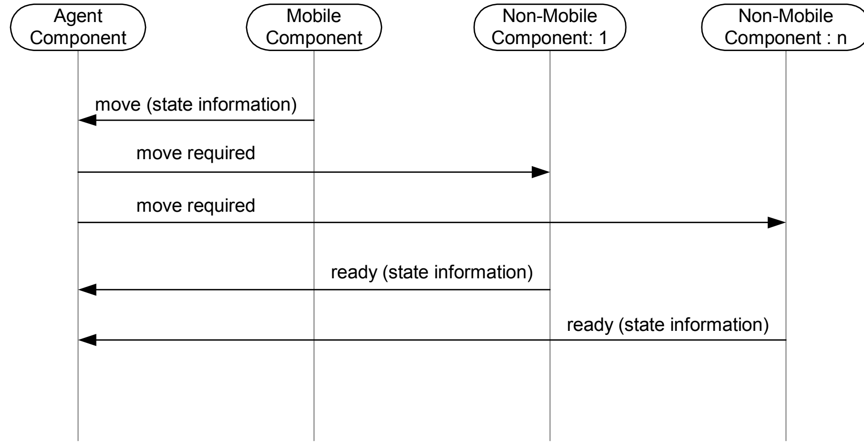


Figure 23. Sequence Diagram for Agent Component Handling Move

2.2.5 MASS System

In the Creating Agent Classes step, the system roles are placed into agent classes. As shown in Figure 24 for analysis option 1, the Bidder role and the Searcher role were combined into the MobileSearcher agent class and the Manager role became a separate agent class called SearchManager. This agent class diagram is the same for the examples used to illustrate design options 1 and 3.

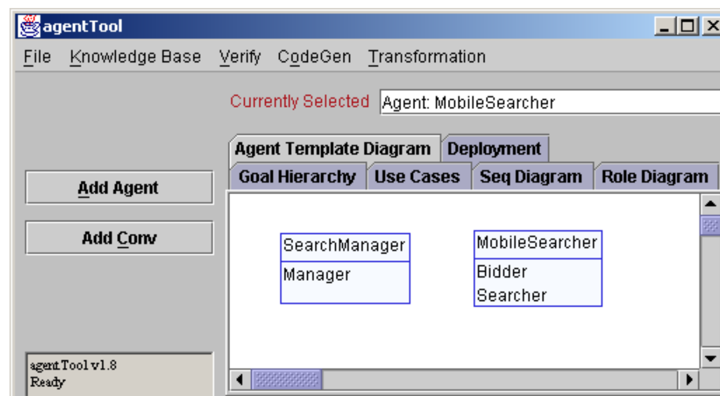


Figure 24. Analysis Option 1 Agent Class Diagram for MASS System

As shown in Figure 25 for analysis option 2, the Bidder, Searcher and MobileAgent roles were combined into the MobileSearcher agent class and the Manager role became a separate agent class called SearchManager. This agent class diagram is used to illustrate design option 2.

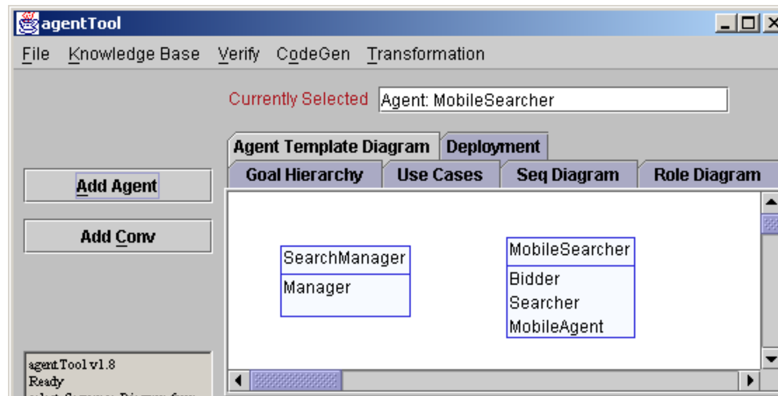


Figure 25. Analysis Option 2 Agent Class Diagram for MASS System

Next, the components belonging to the different agent classes are transformed from the tasks using the transformation process defined by Sparkman [18]. Then, the appropriate mobility requirements described in Section 2.2.4 are added to those components.

The Bidder component is the same for all of the design options presented in the following sections. Since this component resides in an agent class that contains a mobile component, it must have the functionality to fulfill the first four requirements listed in Section 2.2.4. The Bidder component (after transformations [18]) with required mobility additions is shown in Figure 26.

The moveReq transition from the waitForBidResult state to the moveReceived state fulfills the first requirement. In this example, the waitForBidResult state was the only state selected by the designer for receipt of a “moveReq” message. However, all states except for the start state and end state are eligible to receive a move request message. The second and third requirements are satisfied by the function saveCompState in the moveReceived state and the ready(stateInfo,comp) transition from that moveReceived state to the EndState. The fourth requirement is fulfilled by the start(stateInfo) transition from the StartState to the restore state and the transitions from the restore state to other states in the

diagram. Adding the functionality to fulfill the fifth requirement listed in Section 2.2.4, that is, to deny a move request message, is left up to the designer and was not included in the Bidder component. This implies automatic acceptance once a move required message is received.

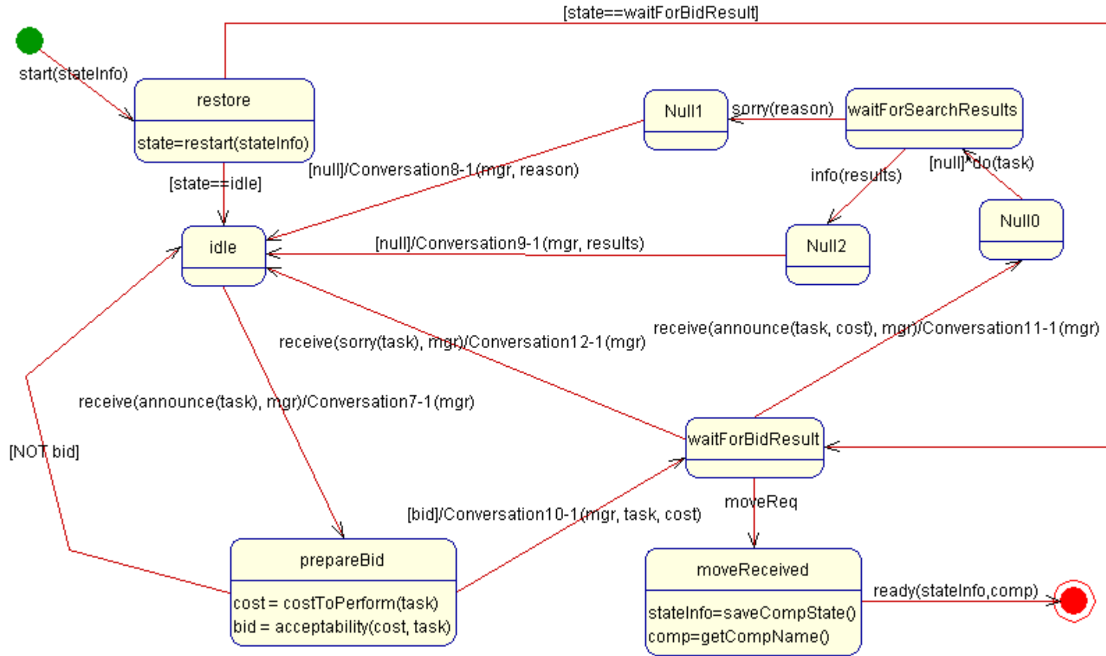


Figure 26. Bidder Component From MASS System

The Agent component will be different for design option 3 and the Search component will be different for design option 1. Because of this reason the diagrams for these components will be presented as appropriate in each design option section.

2.2.5.1 Design Option 1

Figure 27 shows the Search component for the MobileSearcher agent class from Section 2.1.2.1 transformed to handle mobility responsibilities. The Search component is started with an internal start message with the parameter stateInfo. This state information contains the processing state of the Search component as well as for the other components before the Search component called for a move. If the stateInfo is null and the bid component sent the Search component a search task then the Search component

transitions to the moveNeeded state. If the stateInfo is not null then the agent has just moved and the Search component must restart any components that were shutdown at the previous address.

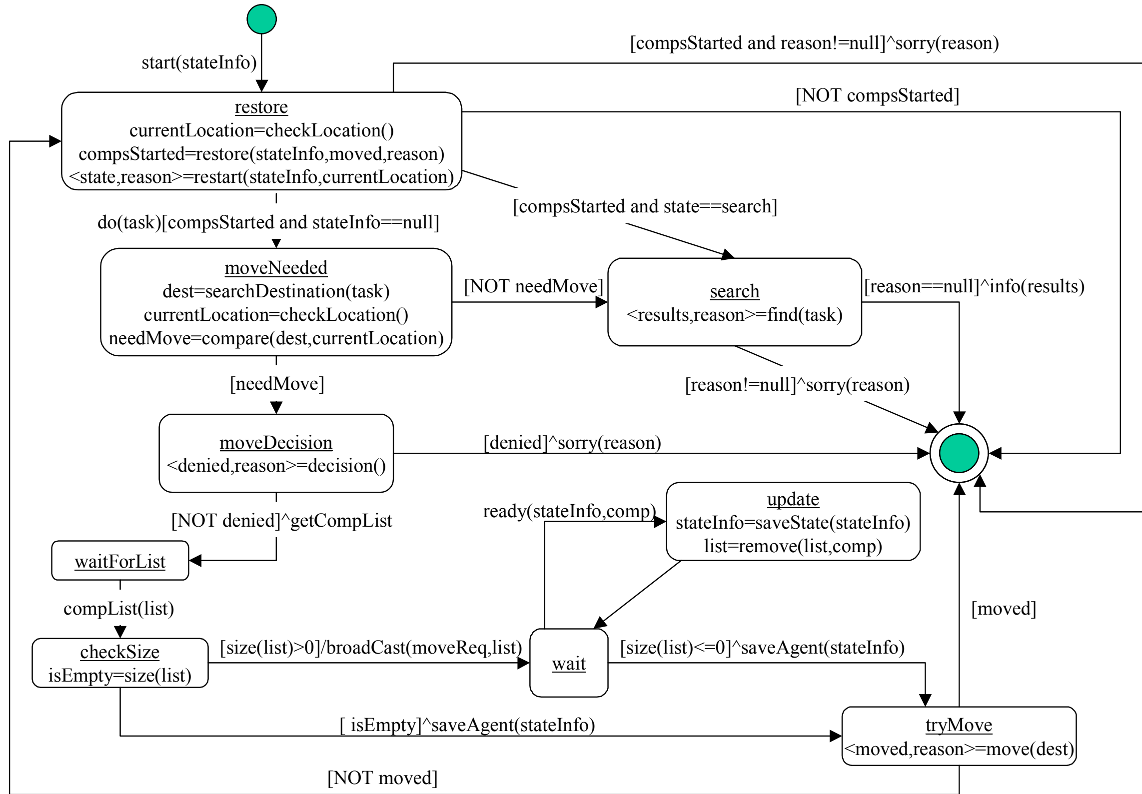


Figure 27. Design Option 1 Search Component for MASS System

The restore function in the restore state handles restarting the other components. If the components do not start for whatever reason then the agent will not be able to perform its goals. An internal terminate message is sent to the agent component. The checkLocation function in the restore state is used to determine the agent's current address. The result of this function is used in conjunction with the stateInfo to determine what state the component should start in. The Boolean variable state is used to transition from the restore state to the proper state for continuing execution. The Search component transitions to the search state if the agent moved.

Once a decision to move has been made in the moveNeeded state, the decision function in the moveDecision state is where the designer can specify how an agent decides to move. If a move is not

allowed, then the Search component has to get the list of currently active components from the agent component. If the list is not empty then a broadCast function is used to send move required (moveReq) messages to all active components including the agent component. Once all the other components have saved state and terminated, the Search component then signals the mobile agent system that the agent needs to move. If the move is denied then the other components have to be restarted. If the move is successful then the agent component is started on the new address and starts the Search component in the reestablish function in the reestablish state.

Figure 28 shows the corresponding agent component for design option 1. This agent component is the combination of transient and persistent discussed in Section 2.2.1.4, because the Bidder component is a persistent component and the Search component is a transient component.

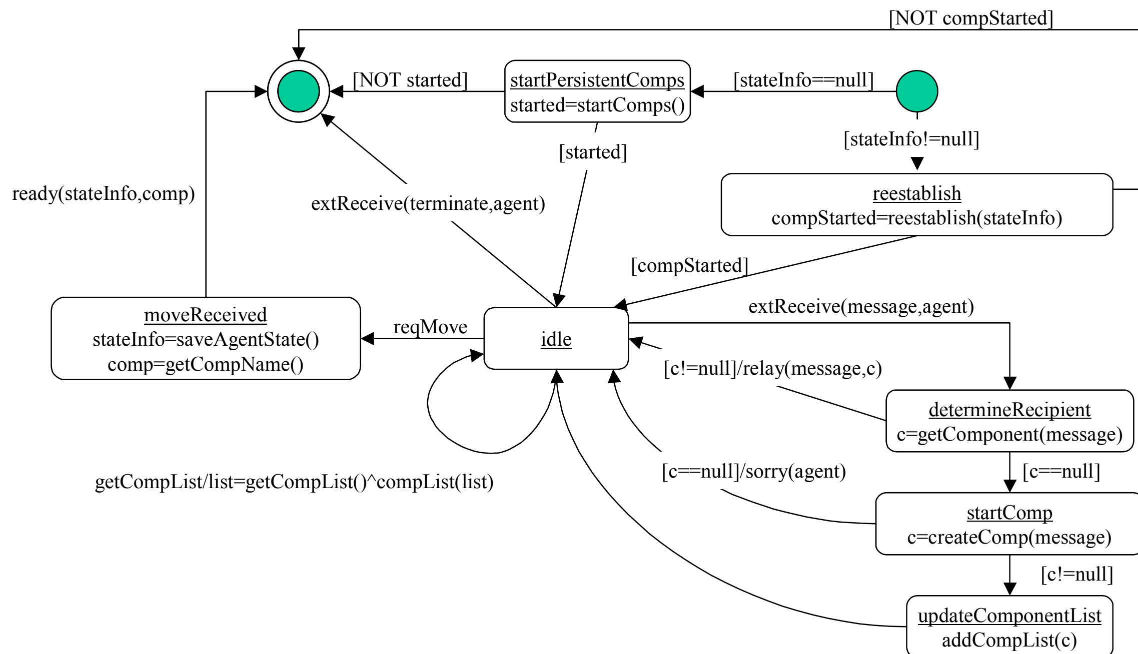


Figure 28. Design Option 1 Agent Component for MASS System

However, there are a few additions due to mobility. If the parameter stateInfo is null then the agent starts by transitioning to the startPersistentComps state. If stateInfo is not null, then the agent starts with a transition to the reestablish state. In this state all of the active components at the time of the move are restarted into the proper state. If there is a problem starting the components then the agent terminates.

If not, the agent transitions to the idle state. If a getCompList internal message is received from a mobile component the agent component retrieves the list and sends it to the requesting component. Then, when a reqMove message is received from a mobile component the agent component transitions to the moveReceived state. The agent component state is saved, sent to the mobile component and the agent component terminates.

2.2.5.2 Design Option 2

In this option a Move component is created from the Move task that is part of the MobileSearcher agent class. Figure 29 shows the Move component for the MASS system. The move component is a transient component and is only started after the agent has moved or a mobile component has requested a move. In the MASS system, once the Search component has requested a move the process to move is exactly the same as it was for the Search component in design option one.

Figure 30 shows the corresponding Search component for this option. This component was unchanged by the transformations defined in [18] but there are numerous mobility additions. The component is started by an internal message from the agent component that contains the stateInfo for the component. If the stateInfo is null then that implies that the component has just been created for the first time with a do(task) message from the Bidder component. If the stateInfo is not null, then the agent has moved and the component is being restarted at the new location.

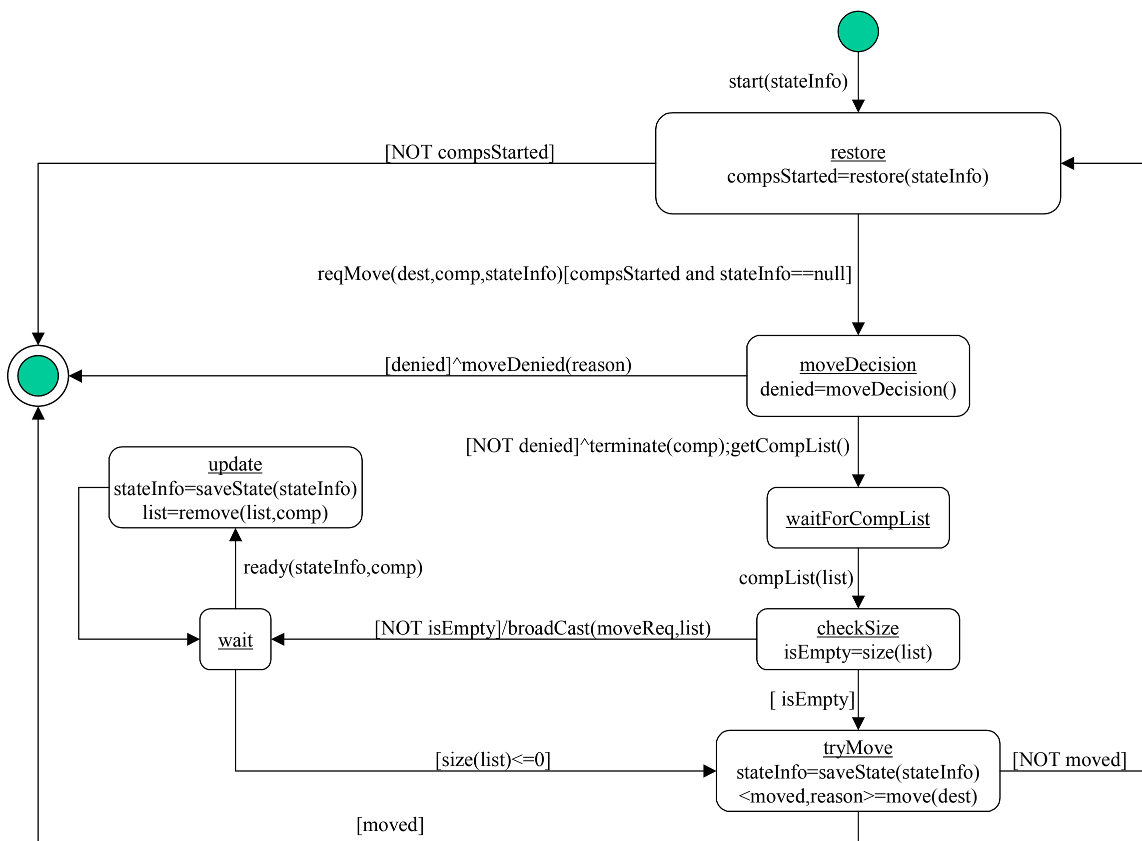


Figure 29. Design Option 2 Mobility Component for MASS System

The component transitions into the search state if the move was successful. But if the move was unsuccessful, a sorry message is sent to the Bidder component and the Search component terminates. However, if the Search component has just been created with a new search task, the compare function in the moveNeeded state determines whether a move is needed in order to fulfill the search request. If a move is needed the component transitions into the moveCalled state where the state of the component is saved and an internal reqMove message is sent to the Move component. If the move is approved, a terminate message is received from the Move component and the component terminates. If the move is denied, the component sends a sorry message to the Bidder component and terminates.

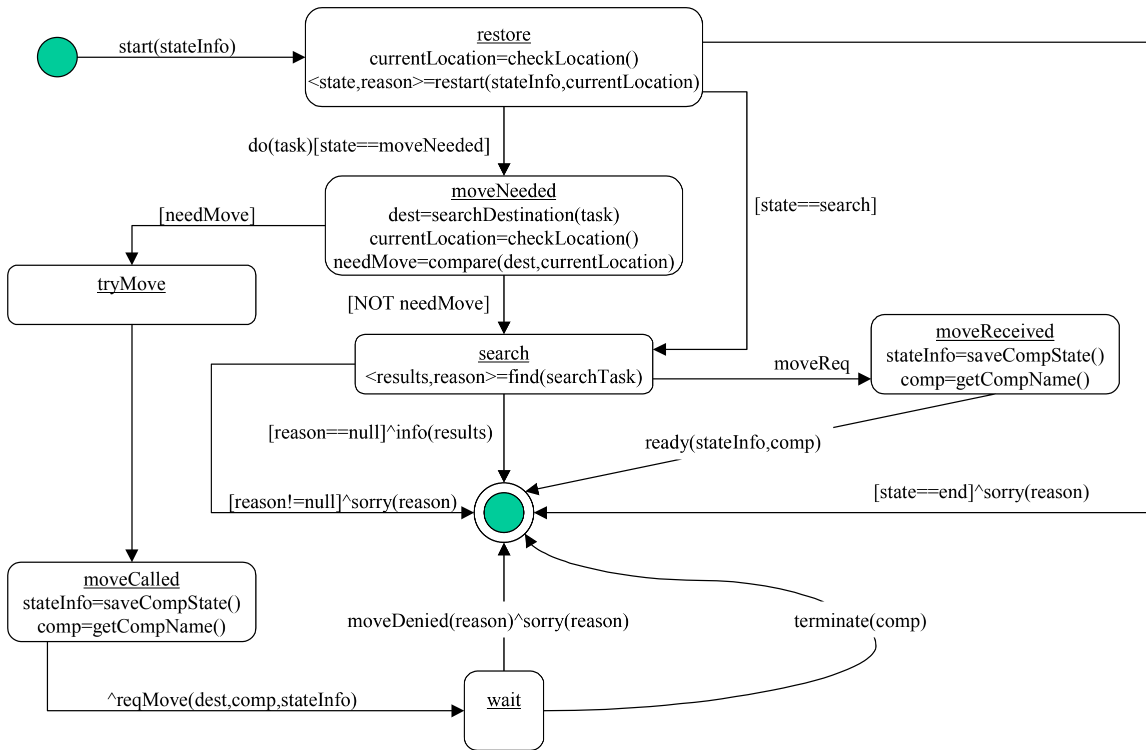


Figure 30. Design Option 2 Search Component for MASS System

2.2.5.3 Design Option 3

In this option the Agent component handles the move for the agent. Figure 31 shows the Agent component for this option. The Search component is exactly the same as for design option 2. There are two possible transitions that can be triggered on startup of the agent component. If stateInfo is null, then the agent has just been instantiated and the agent component transitions to the startPersistentComps state. If stateInfo is not null, then the agent has moved and the agent component transitions to the reestablish state. All of the components that were active when a move was called on the previous address are restarted and the agent component transitions to the idle state.

When the agent component receives an internal request move message (reqMove) the agent component transitions to the moveDecision state. As was the case with the two previous design options the designer can include agent-moving logic in the decision function within the moveDecision state. If the

move is denied then the agent component sends an internal move denied message (moveDenied) to the component that called for the move. If the move is not denied then the agent component transitions to the buildComponentList state.

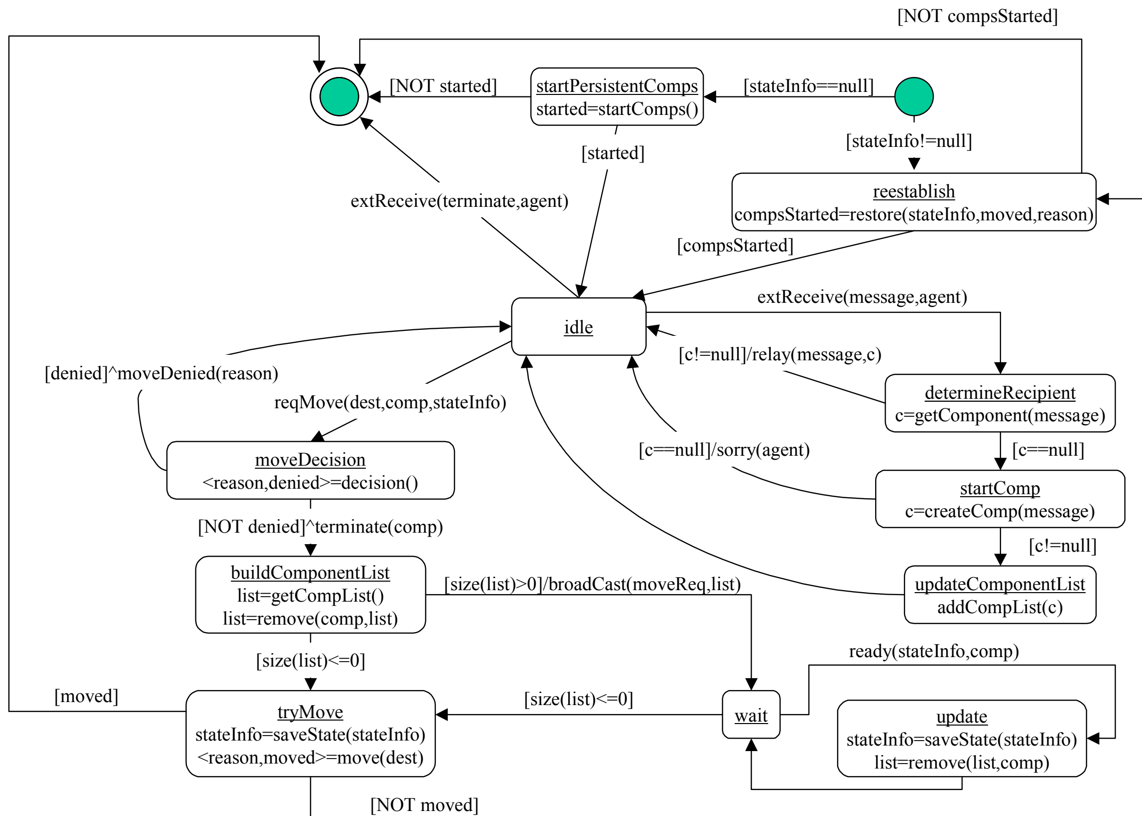


Figure 31. Design Option 3 Agent Component for MASS System

The getCompList function in the buildComponentList state returns a list of all of the active components. The agent component has access to all of the agents' methods so it can call the getCompList function directly whereas the search and move components in the previous design options had to send messages to the agent component requesting that information.

The component that called for the move is removed from the list because the stateInfo for that component has already been received. If the list of active components is not empty then each active component is sent an internal moveReq message. Once a reply is received from a component that component is removed from the list. When the list is empty the agent component saves state for itself and

calls the move method that interfaces with the mobile agent system to perform the move. If the move fails the agent component transitions to the reestablish state and the components are restarted in the proper state. If the move is successful the mobile agent system will shutdown the agent component on the pre-move address and restart the agent component on the new address.

2.3 Selection of Analysis and Design Options for Mobility

Selecting the best option for either phase was determined by examining the advantages and disadvantages of each option as well as the relationships between the options. Analysis option 1 was selected over analysis option 2 because it did not restrict the design choice, as did analysis option 2. All of the design options were feasible if option 1 was selected whereas only design option 2 would have been feasible for the design phase if option 2 were selected. Analysis option 1 was also contained in only one step of the analysis phase while option 2 could affect every analysis phase step. Therefore, option 1 left the majority of the moving details to be handled in the design phase, which reduced the complexity of the analysis. Option 1 also allowed for the additional move specification described in Section 2.1.

Since analysis option 1 was selected, as was mentioned above, any design choice was possible. However, design option 3 was selected for the following reasons. It does not duplicate moving functionality similar to option 1 or require an additional component to be added to the system as did option 2. Option 3 kept the overall decision process for moving an agent in the agent component, not in the other components like options 1 and 2. Options 1 and 2 gave mobile components a certain amount of power over the other components whereas option 3 kept the power in the agent component. Option 3 also had the lowest communication overhead as opposed to options 1 and 2 during the moving process.

2.4 Summary

Each phase and step of the MaSE methodology was examined to determine where mobility should be incorporated. A way of specifying mobility was already present in the MaSE analysis phase but needed

to be expanded. After adding the new functionality, two analysis phase mobility options were explored in detail with advantages and disadvantages given for each option.

In contrast, there was no existing way of specifying mobility in the design phase even though mobility could be present in the analysis phase. But, with the creation of mapping between the concurrent tasks and existing agent component architecture, mobility was then carried over from the analysis phase to the design phase. However, even with these changes the new models in the design phase needed additional functionality in order to properly handle mobility. Three different options for adding this functionality were presented and analyzed according to their advantages and disadvantages.

Finally, the best options for both the analysis and design phases were selected. In Chapter III, the transformations needed to add mobility to the analysis and design models of MaSE are defined.

III. Design

This chapter presents the transformations that incrementally add mobility functionality to the agent architecture. Section 3.1 presents the transformations developed to change the generic agent architecture design model into a generic mobility design model. Section 3.1.2 presents the analysis phase transformations while Section 3.1.3 presents the design phase transformations.

3.1 Transformations

Now that the best options for adding mobility functionality to the MaSE methodology have been selected, formal transformations can now be defined to first complete the conversion of the analysis models to design models. The transformation process is shown graphically in Figure 32.

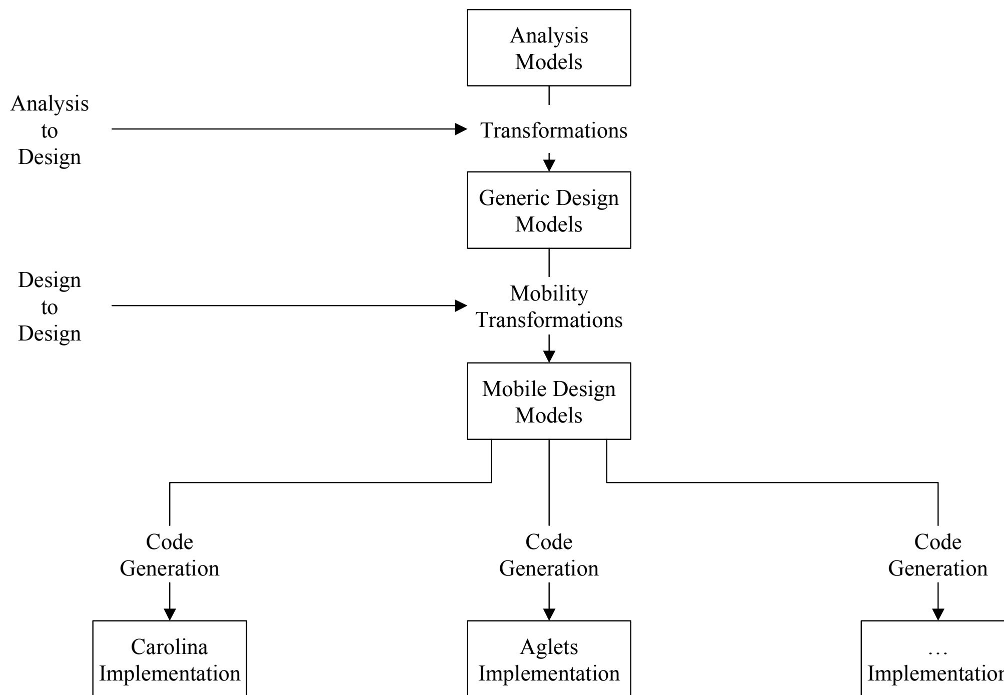


Figure 32. MaSE Transformation Architecture

The analysis-to-design transformations are composed of the transformation process defined by Sparkman [18], including a transformation to ensure move activities remain in component state tables, and

the transformations that create the agent component for each agent class as defined in the following sections of this chapter. The design-to-design transformations can consist, for example, of mobility transformations (as defined in subsequent sections of this chapter), security transformations, or communication protocol transformations. Only mobility transformations are discussed in this thesis. Software code generation (a future research area) for mobile agent systems can be accomplished once the mobile design models are complete.

In order to define formal transformations, the models used in those transformations must be formally defined as well. These models are required to have exact semantics to guarantee predictable behavior of the transformations. The models used in this thesis are a subset of the models defined by Sparkman [18] and are discussed in detail in Appendix B.

3.1.1 Transformation Functions

Before defining the transformations, eight functions that determine what type of components exist had to be defined and are listed below:

1. Function isConversation_Before returns true if there is a start but no end of a conversation before a state containing a move activity.

2. Function hasTransientComponent returns true if there is at least one transient component in an agent class.

3. Function hasPersistentComponent returns true if there is at least one persistent component in an agent class.

4. Function isMobility_Specified returns true if mobility has been specified in at least one component within an agent class.

5. Function isMobility_Specified_Component returns true if mobility has been specified in a given component.

6. Function isMobility_Specified_State returns true if mobility has been specified in a given state.

7. Function mobileComponentCount counts how many components have a move activity specified within a mobile agent class.

8. Function isTransientComponent returns true if a given component is transient.

They are defined as follows.

function isConversation_Before (t : **Transition**) returns boolean
Precondition :
Postcondition :
(t.start = true $\wedge$ t.end = false) $\vee$ ($\exists$ t2 : **Transition** • t2.to = t.from $\wedge$ isMove_Conversation(t2))

function hasTransientComponent (ag : **Agent**) returns Boolean
Precondition : true
Postcondition :
$\exists$ c : **Component**, st : **StateTable**, s : **State**, t : **Transition** •
(c $\in$ ag.components $\wedge$ st = c.stateTable $\wedge$ s $\in$ st.states $\wedge$ s.name = start $\wedge$ t $\in$ st.transitions $\wedge$
t.from = s $\wedge$ (t.receive != null $\vee$ t.receiveEvent != null))

function hasPersistentComponent (ag : **Agent**) returns boolean
Precondition : true
Postcondition :
$\exists$ c : **Component**, st : **StateTable**, s : **State**, t : **Transition** •
(c $\in$ ag.components $\wedge$ st = c.stateTable $\wedge$ s $\in$ st.states $\wedge$ s.name = “start” $\wedge$ t $\in$ st.transitions $\wedge$
t.from = s $\wedge$ t.receive == null $\wedge$ t.receiveEvent == null)

function isMobility_Specified (ag : **Agent**) returns boolean
Precondition : true
Postcondition :
$\exists$ c : **Component**, st : **StateTable**, s : **State**, a : **Action**, f : **FunctionCall** •
(c $\in$ ag.components $\wedge$ c.name = “AgentComponent” $\wedge$ st = c.stateTable $\wedge$ s $\in$ st.states $\wedge$
a $\in$ s.actions $\wedge$ a.rhs = f $\wedge$ f.name = “move”)

function isMobility_Specified_Component (c : **Component**) returns boolean
Precondition : true
Postcondition :
$\exists$ st : **StateTable**, s : **State**, a : **Action**, f : **FunctionCall** •
(st = c.stateTable $\wedge$ s $\in$ st.states $\wedge$ a $\in$ s.actions $\wedge$ a.rhs = f $\wedge$ f.name = “move”)

function isMobility_Specified_State (s : **State**) returns boolean
Precondition : true
Postcondition :
$\exists$ a : **Action**, f : **FunctionCall** •
(a $\in$ s.actions $\wedge$ a.rhs = f $\wedge$ f.name = "move")

function mobileComponentCount (ag : **Agent**) returns integer
Precondition : integer intMobileComponentCount = 0
Postcondition :
$\forall$ c : **Component**, st : **StateTable**, s : **State**, a : **Action**, f : **FunctionCall** •
(c $\in$ ag.components $\wedge$ c.name $\neq$ "AgentComponent" $\wedge$ st = c.stateTable $\wedge$
($\exists$ s : **State**, a : **Action** f : **FunctionCall** • s $\in$ st.states $\wedge$ a $\in$ s.actions $\wedge$ a.rhs = f $\wedge$
f.name = "move"))
$\Rightarrow$
(intMobileComponentCount = intMobileComponentCount + 1)

function isTransientComponent (c : **Component**) returns boolean
Precondition : true
Postcondition :
$\exists$ st : **StateTable**, t : **Transition** •
(st = c.stateTable $\wedge$ t $\in$ st.transitions $\wedge$ t.from = "StartState" $\wedge$ (t.receive $\neq$ null $\vee$ t.ReceiveEvent $\neq$
null))

3.1.2 Analysis to Design

To complete the analysis-to-design transformation process, move activities need to be carried over from the analysis phase to the design phase and the agent component needs to be created for each agent class. A transformation was created to ensure that move activities remain in component state tables and are not placed into conversations. This transformation is inserted into Stage 2 of the transformation process defined by Sparkman [18]. After that special case has been handled, transformations that build the agent component are defined.

3.1.2.1 Ensuring Mobility Remains in Component

In a component state table, a check is needed to ensure that all states that contain a call to the move activity remain in the component state table and do not get placed into a conversation. This could happen if an external SendEvent or ReceiveEvent for a conversation occurs in a component before a state with a move activity and there is an external ReceiveEvent or SendEvent pertaining to that same conversation that occurs after the move state. The component state diagram shown in is the result of

performing transformation 25, 26 27 or 28 [18]. If left like this, “state n” would end up in a conversation, which, as discussed in Section 2.2.2, is not allowed in the new MaSE agent architecture.

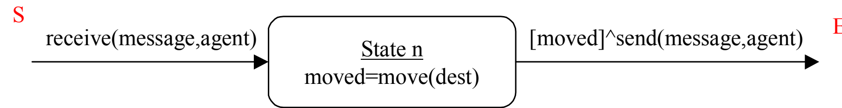


Figure 33. Example Component After Transformation Defined by [18]

Therefore, if there is a start but no end of a conversation before a state containing a move activity then the transition to the move state is labeled as the end of the conversation and the corresponding transition after the move state is labeled as the start of new conversation. Transformation 28a adds the end conversation markers to the transitions before the move state and the end conversation markers to the transitions following the move state. Figure 34 shows the component from Figure 33 after transformation 28a is complete.

Transformation 28a

∀ s : **State** •
(isMobility_Specified_State(s) ∧ (∃ t : Transition • t.to = s ∧ isConversation_Before(t))
⇒
(∀ (t2 : **Transition** • t.to = s ⇒ t.end = true)

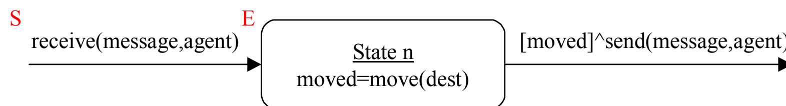


Figure 34. Example Component Diagram From Figure 32 After Transformations 28a

3.1.2.2 Agent Component

Next, the creation of the agent components for all the agent classes in the system is required even if mobility has not been specified in any components. The following table determines which Agent Component Transform (ACT) is executed given the agent type, component types, and whether or not mobility is specified. The transformations build the agent component in accordance with the functionality defined in Chapter II Sections 2.2.1.1, 2.2.1.2, 2.2.1.3 and 2.2.1.4.

Table 1. Agent Component Transformations

Mobile	Type of Components	ACT Transformations Needed
Yes	Transient	1, 2
No		1, 2, 3
Yes	Persistent	1, 4, 6, 7
No		1, 4, 6, 7, 9
Yes	Transient and Persistent	1, 2, 5, 8
No		1, 2, 5, 8, 9

3.1.2.2.1 Agent Component Transform 1

Agent Component Transform 1 (ACT1) creates the basic agent component, as presented in Section 2.2.1.1, regardless of the types of components possessed by an agent class. This transform adds the start, end, idle and determineRecipient states as well as the transitions for handling agent termination and external messages to start conversations or components. Figure 35 shows the agent component after the ACT1 transform is complete.

ACT 1

∀ ag : **Agent •**
⇒
(∃ ac : **Component**, st : **StateTable**, t1, t2, t3, t4 : **Transition**, s1, s2, s3, s4 : **State**, se : **SendEvent**, re1, re2 : **ReceiveEvent**, a1, a2 : **Action**, e1, e2, e3 : **Event**, f1, f2 : **FunctionCall**, p1, p2, p3 : **Parameter •**
p1.name = “agent” ∧ p2.name = “message” ∧ p3.name = “c” ∧ f1.name = “getComponent” ∧
f1.parameters = [p2] ∧ f2.name = “relay” ∧ f2.parameters = [p2,p3] ∧ e1.name = “message” ∧
e1.parameters = [] ∧ e2.name = “send” ∧ e2.parameters = [p2] ∧ e3.name = “terminate” ∧ e3.parameters = [] ∧
a1.lhs = “c” ∧ a1.rhs = f1 ∧ a2.lhs = null ∧ a2.rhs = f2 ∧ re1.event = e1 ∧ re1.sender = p1 ∧ re1.protocol = {} ∧
re2.event = e3 ∧ re2.sender = p1 ∧ re2.protocol = {} ∧ se.event = e1 ∧ se.recipient = p1 ∧ se.protocol = {} ∧
se.convID = null ∧ s1 ∈ st.states ∧ s1.name = “start” ∧ s1.actions = [] ∧ s1.convIDs = {} ∧ s2 ∈ st.states ∧
s2.name = “idle” ∧ s2.actions = [] ∧ s2.convIDs = {} ∧ s3 ∈ st.states ∧ s3.name = “determineRecipient” ∧
s3.actions = [a1] ∧ s3.convIDs = {} ∧ s4 ∈ st.states ∧ s4.name = “end” ∧ s4.actions = [] ∧ s4.convIDs = {} ∧
t1 ∈ st.transitions ∧ t1.from = s2 ∧ t1.receive = null ∧ t1.receiveEvent = re1 ∧ t1.guard = null ∧ t1.to = s3 ∧
t1.actions = [] ∧ t1.sends = {} ∧ t1.sendEvents = {} ∧ t1.start = false ∧ t1.end = false ∧ t1.convIDs = {} ∧
t1.AgentID = null ∧ t2 ∈ st.transitions ∧ t2.from = s3 ∧ t2.receive = null ∧ t2.receiveEvent = null ∧
t2.guard = “c!=null” ∧ t2.to = s2 ∧ t2.actions = [a2] ∧ t2.sends = {} ∧ t2.sendEvents = {} ∧ t2.start = false ∧
t2.end = false ∧ t2.convIDs = {} ∧ t2.AgentID = null ∧t3 ∈ st.transitions ∧ t3.from = s2 ∧ t3.receive = e2 ∧
t3.receiveEvent = null ∧ t3.guard = null ∧ t3.to = s2 ∧ t3.actions = [] ∧ t3.sends = {} ∧ t3.sendEvents = {se} ∧
t3.start = false ∧ t3.end = false ∧ t3.convIDs = {} ∧ t3.AgentID = null ∧ t4 ∈ st.transitions ∧ t4.from = s2 ∧
t4.receive = null ∧ t4.receiveEvent = re2 ∧ t4.guard = null ∧ t4.to = s4 ∧ t4.actions = [] ∧ t4.sends = {} ∧
t4.sendEvents = {} ∧ t4.start = false ∧ t4.end = false ∧ t4.convIDs = {} ∧ t4.AgentID = null ∧
st = ac.stateTable ∧ ac ∈ ag’.components ∧ ac ∉ ag.components ∧ ac.name = “AgentComponent”)

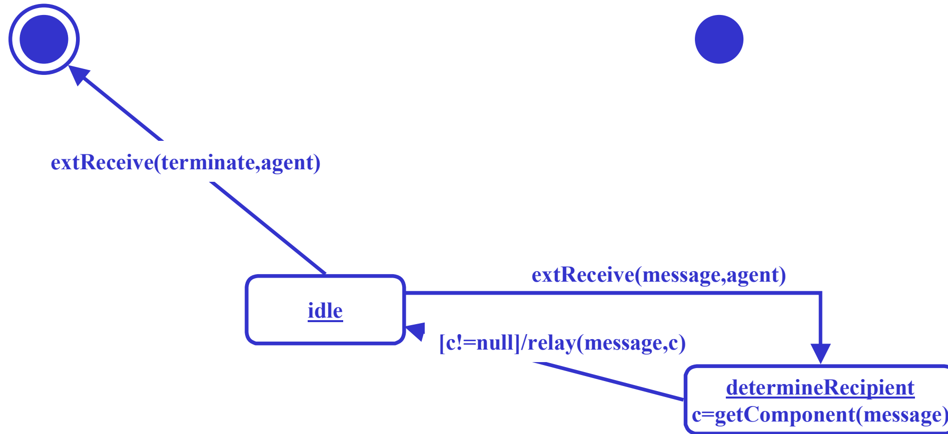


Figure 35. Agent Component Diagram After ACT1 Transformation

3.1.2.2.2 Agent Component Transform 2

Agent Component Transform 2 (ACT2) adds the logic, as presented in Section 2.2.1.2, that handles starting transient components and handling the receipt of invalid messages from other agents to the agent component. Figure 36 shows the agent component diagram after the ACT2 transform is complete.

<u>**ACT 2**</u>

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, drs : **State** •
(c = ag.component ∧ st = c.stateTable ∧ is ∈ st.states ∧ is.name = "idle" ∧ drs ∈ st.states ∧
drs.name = "determineRecipient" ∧ hasTransientComponent(ag))
⇒
(∃ s1, s2 : **State**, t1, t2, t3, t4 : **Transition**, a1, a2 : **Action**, f1, f2, f3 : **FunctionCall**, p1, p2, p3 : **Parameter** •
p1.name = "message" ∧ p2.name = "c" ∧ p3.name = "agent" ∧ f1.name = "createComp" ∧ f1.parameters = [p1] ∧
f2.name = "addCompList" ∧ f2.parameters = [p2] ∧ f3.name = "sorry" ∧ f3.parameters = [p3] ∧ a1.lhs = "c" ∧
a1.rhs = f1 ∧ a2.lhs = null ∧ a2.rhs = f2 ∧ a3.lhs = null ∧ a3.rhs = f3 ∧ s1 ∈ st'.states ∧ s1.name = "startComp" ∧
s1.actions = [a1] ∧ s1.convIDs = {} ∧ s2 ∈ st'.states ∧ s2.name = "updateComponentList" ∧ s2.actions = [a2] ∧
s2.convIDs = {} ∧ t1 ∈ st'.transitions ∧ t1.from = drs ∧ t1.receive = null ∧ t1.receiveEvent = null ∧
t1.guard = "c==null" ∧ t1.to = s1 ∧ t1.actions = [] ∧ t1.sends = {} ∧ t1.sendEvents = {} ∧ t1.start = false ∧ t1.end = false ∧ t1.convIDs = {} ∧ t1.AgentID = null ∧ t2 ∈ st'.transitions ∧ t2.from = s1 ∧ t2.receive = null ∧
t2.receiveEvent = null ∧ t2.guard = "c==null" ∧ t2.to = s2 ∧ t2.actions = [a3] ∧ t2.sends = {} ∧
t2.sendEvents = {} ∧ t2.start = false ∧ t2.end = false ∧ t2.convIDs = {} ∧ t2.AgentID = null ∧
t3 ∈ st'.transitions ∧ t3.from = s1 ∧ t3.receive = null ∧ t3.receiveEvent = re2 ∧ t3.guard = "c!=null" ∧
t3.to = s2 ∧ t3.actions = [] ∧ t3.sends = {} ∧ t3.sendEvents = {} ∧ t3.start = false ∧ t3.end = false ∧
t3.convIDs = {} ∧ t3.AgentID = null ∧ t4 ∈ st'.transitions ∧ t4.from = s2 ∧ t4.receive = null ∧
t4.receiveEvent = re2 ∧ t4.guard = null ∧ t4.to = is ∧ t4.actions = [] ∧ t4.sends = {} ∧ t4.sendEvents = {} ∧
t4.start = false ∧ t4.end = false ∧ t4.convIDs = {} ∧ t4.AgentID = null)

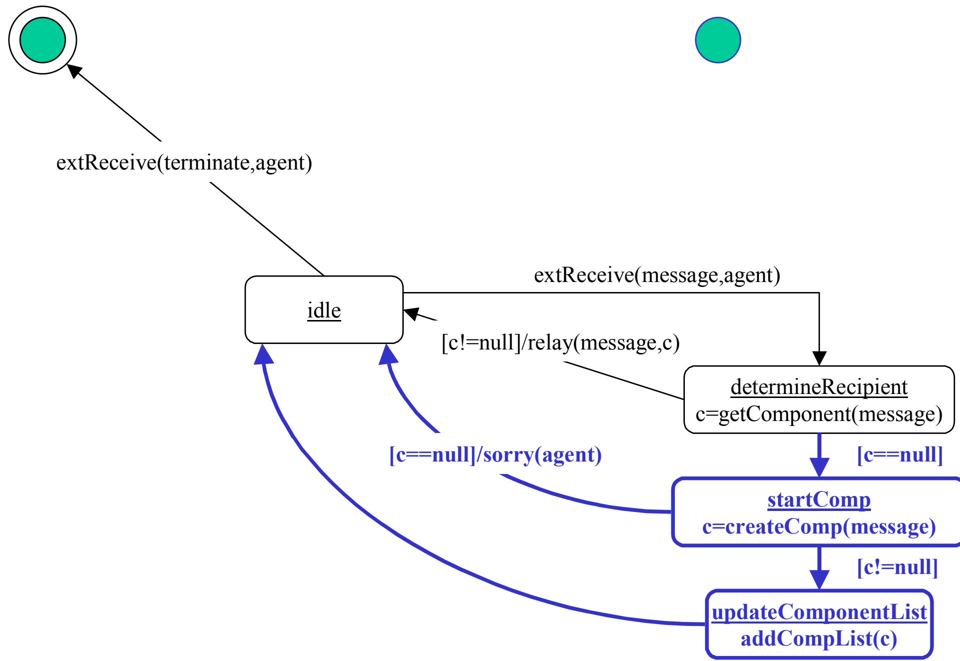


Figure 36. Agent Component Diagram After ACT2 Transformation

3.1.2.2.3 Agent Component Transform 3

Agent Component Transform 3 (ACT3) adds, as presented in Section 2.2.1.4, the transition from the start state to the idle state, if the components of an agent class are transient only and do not contain move activities, to the agent component. Figure 37 shows the agent component diagram after the ACT3 transform is complete.

ACT 3

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, ss : **State** •
(c = ag.component ∧ st = c.stateTable ∧ is ∈ st.states ∧ is.name = "idle" ∧ ss ∈ st.states ∧ ss.name = "start" ∧ ¬hasPersistentComponent(ag) ∧ ¬isMobility_Specified(ag))
⇒
(∃ t : **Transition** •
t ∈ st'.transitions ∧ t.from = ss ∧ t.receive = null ∧ t.receiveEvent = null ∧ t.guard = null ∧ t.to = s1 ∧ t.actions = [] ∧ t.sends = {} ∧ t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧ t.AgentID = null)

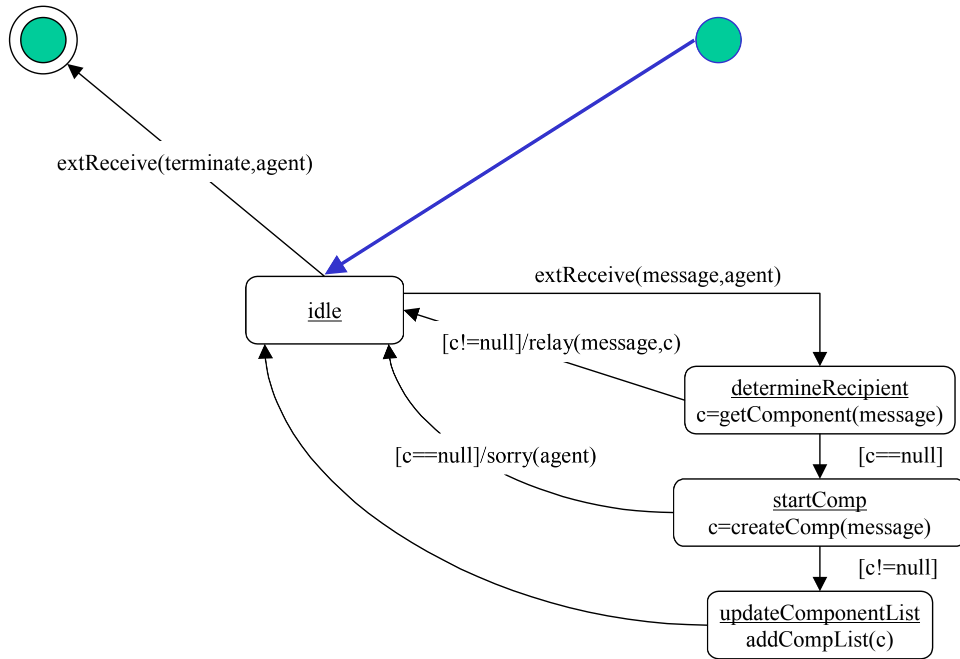


Figure 37. Agent Component Diagram After ACT3 Transformation

3.1.2.2.4 Agent Component Transform 4

Agent Component Transform 4 (ACT4) adds the logic, as presented in Section 2.2.1.3, that handles received messages not belonging to the agent to the agent component. Figure 38 shows the proactive agent component diagram after the ACT4 transform is complete.

ACT 4

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, drs : **State** •
(c = ag.component ∧ st = c.stateTable ∧ is ∈ st.states ∧ is.name = "idle" ∧ drs ∈ st.states ∧
drs.name = "determineRecipient" ∧ ¬hasTransientComponent(ag))
⇒
(∃ t : **Transition**, a : **Action**, f : **FunctionCall**, p : **Parameter** •
p.name = "agent" ∧ f.name = "sorry" ∧ f.parameters = [p] ∧ a.lhs = null ∧ a.rhs = f ∧ t ∈ st'.transitions ∧
t.from = drs ∧ t.receive = null ∧ t.receiveEvent = null ∧ t.guard = "c==null" ∧ t.to = is ∧ t.actions = [a] ∧
t.sends = {} ∧ t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧ t.AgentID = null)

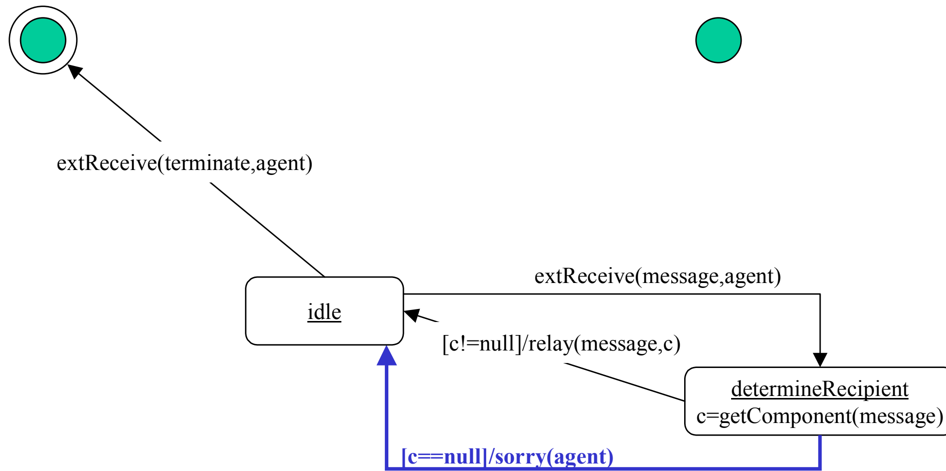


Figure 38. Agent Component Diagram After ACT4 Transformation

3.1.2.2.5 Agent Component Transform 5

Agent Component Transform 5 (ACT5) adds the logic, as presented in Section 2.2.1.3, to start all persistent components on agent creation without setting a time interval to the agent component. This transform is executed on agents with both transient and persistent components. Figure 39 shows the agent component diagram after the ACT5 transform is complete.

ACT 5

∀ ag : **Agent**, c : **Component**, st : **StateTable**, es : **State** •
(c = ag.component ∧ st = c.stateTable ∧ es ∈ st.states ∧ es.name = “end” ∧ ¬hasTransientComponent(ag))
⇒
(∃ s : **State**, t : **Transition**, a : **Action**, f : **FunctionCall** •
f.name = “startComps” ∧ f.parameters = [] ∧ a.lhs = “started” ∧ a.rhs = f ∧ s ∈ st’.states ∧
s.name = “startPersistentComps” ∧ s.actions = [a1] ∧ s.convIDs = {} ∧ t ∈ st’.transitions ∧ t.from = s ∧
t.receive = null ∧ t.receiveEvent = null ∧ t.guard = “NOT started” ∧ t.to = es ∧ t.actions = [] ∧ t.sends = {} ∧
t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧ t.AgentID = null)

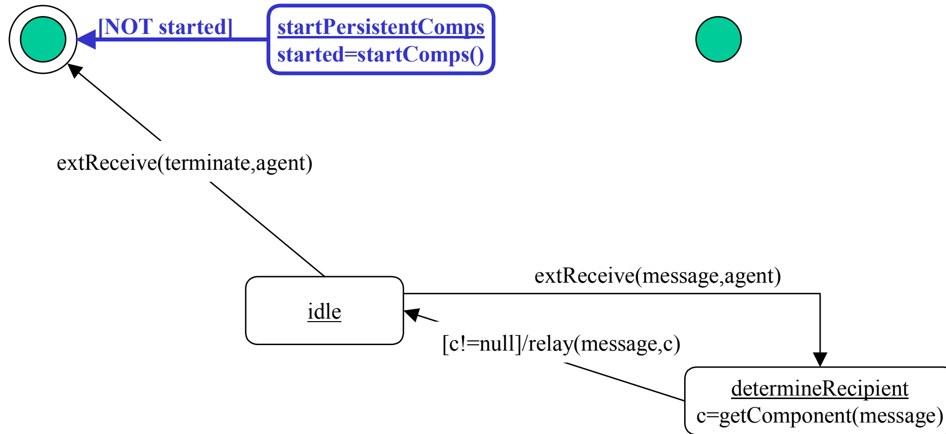


Figure 39. Agent Component Diagram After ACT5 Transformation

3.1.2.2.6 Agent Component Transform 6

Agent Component Transform 6 (ACT6) adds the logic, as presented in Section 2.2.1.3, to start all persistent components to the agent component. This transformation is executed on agents that only contain persistent components. Figure 40 shows the agent component diagram after the ACT6 transform is complete.

ACT 6

∀ ag : **Agent**, c : **Component**, st : **StateTable**, es : **State** •
(c = ag.component ∧ st = c.stateTable ∧ es ∈ st.states ∧ es.name = "end" ∧ ¬hasTransientComponent(ag))
⇒
(∃ s : **State**, t : **Transition**, a1, a2 : **Action**, f1, f2 : **FunctionCall** •
f1.name = "startComps" ∧ f1.parameters = [] ∧ f2.name = "getSleepTime" ∧ f2.parameters = [] ∧
a1.lhs = "started" ∧ a1.rhs = f ∧ a2.lhs = "time" ∧ a2.rhs = f2 ∧ s ∈ st'.states ∧
s.name = "startPersistentComps" ∧ s.actions = [a1] ∧ s.convIDs = {} ∧ t ∈ st'.transitions ∧ t.from = s ∧
t.receive = null ∧ t.receiveEvent = null ∧ t.guard = "NOT started" ∧ t.to = es ∧ t.actions = [] ∧ t.sends = {} ∧
t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧ t.AgentID = null)

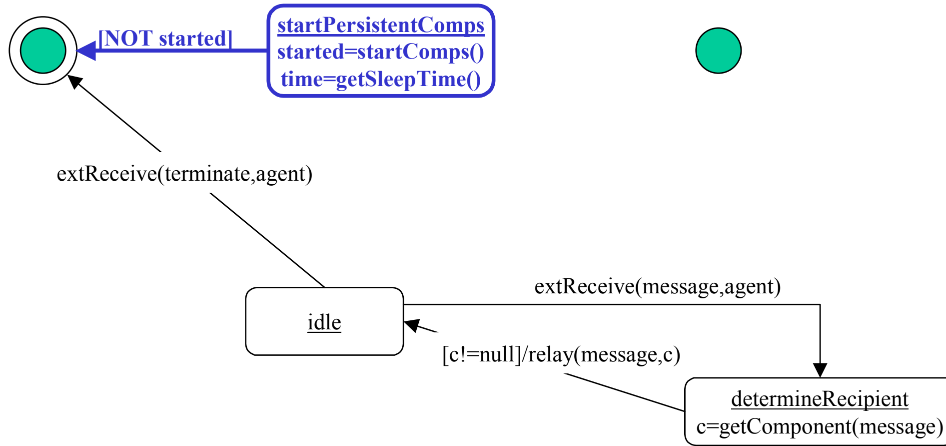


Figure 40. Agent Component Diagram After ACT6 Transformation

3.1.2.2.7 Agent Component Transform 7

Agent Component Transform 7 (ACT7) adds the logic, as presented in Section 2.2.1.3, to periodically check the persistent components to see if they are still alive to the agent component. This transform is executed on agents with only persistent components. Figure 41 shows the agent component diagram after the ACT7 transform is complete.

<u>**ACT 7**</u>

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, es : **State** •
(c = ag.component ∧ st = c.stateTable ∧ is ∈ st.states ∧ is.name = “idle” ∧ es ∈ st.states ∧
es.name = “end” ∧ spcs ∈ st.states ∧ spcs.name = “startPersistentComps” ∧ ¬hasTransientComponent(ag))
⇒
(∃ s : **State**, t1, t2, t3, t4 : **Transition**, a1, a2, a3 : **Action**, f1, f2, f3 : **FunctionCall**, p1, p2 : **Parameter** •
p1.name = “time” ∧ p2.name = “time” ∧ f1.name = “checkActiveComps” ∧ f1.parameters = [] ∧
f2.name = “setTimer” ∧ f2.parameters = [p1] ∧ f3.name = “setTimer” ∧ f3.parameters = [p2] ∧
a1.lhs = “active” ∧ a1.rhs = f1 ∧ a2.lhs = “t” ∧ a2.rhs = f2 ∧ a3.lhs = “t” ∧ a3.rhs = f3 ∧ s ∈ st’.states ∧
s.name = “checkComps” ∧ s.actions = [a1] ∧ s.convIDs = {} ∧ t1 ∈ st’.transitions ∧ t1.from = is ∧
t1.receive = null ∧ t1.receiveEvent = null ∧ t1.guard = “timeout(t)” ∧ t1.to = s ∧ t1.actions = [] ∧ t1.sends = {} ∧
t1.sendEvents = {} ∧ t1.start = false ∧ t1.end = false ∧ t1.convIDs = {} ∧ t1.AgentID = null ∧
t2 ∈ st’.transitions ∧ t2.from = s ∧ t2.receive = null ∧ t2.receiveEvent = null ∧ t2.guard = “active” ∧ t2.to = is ∧
t2.actions = [a2] ∧ t2.sends = {} ∧ t2.sendEvents = {} ∧ t2.start = false ∧ t2.end = false ∧ t2.convIDs = {} ∧
t2.AgentID = null ∧ t3 ∈ st’.transitions ∧ t3.from = s ∧ t3.receive = null ∧ t3.receiveEvent = null ∧
t3.guard = “NOT active” ∧ t3.to = es ∧ t3.actions = [] ∧ t3.sends = {} ∧ t3.sendEvents = {} ∧ t3.start = false ∧
t3.end = false ∧ t3.convIDs = {} ∧ t3.AgentID = null ∧ t4 ∈ st’.transitions ∧ t4.from = spcs ∧ t4.receive = null ∧
t4.receiveEvent = null ∧ t4.guard = “started” ∧ t4.to = is ∧ t4.actions = [a] ∧ t4.sends = {} ∧
t4.sendEvents = {} ∧ t4.start = false ∧ t4.end = false ∧ t4.convIDs = {} ∧ t4.AgentID = null))

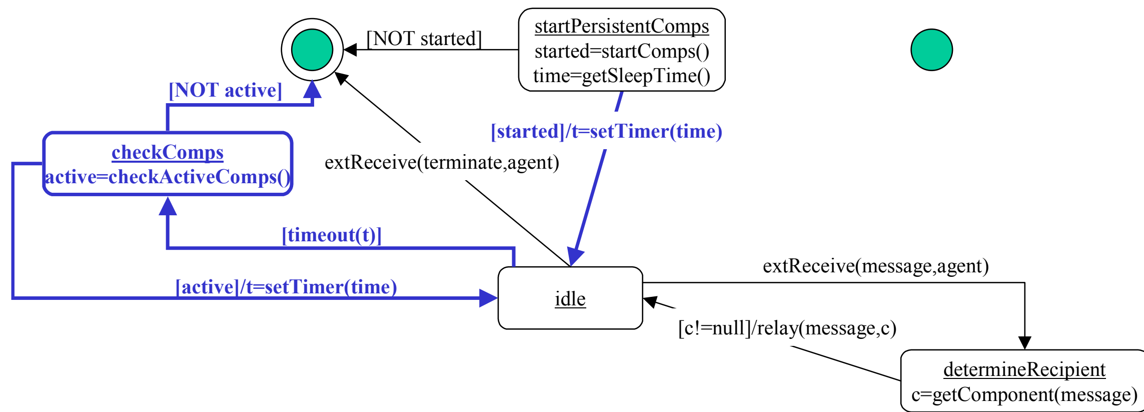


Figure 41. Agent Component Diagram After ACT7 Transformation

3.1.2.2.8 Agent Component Transform 8

Agent Component Transform 8 (ACT8) adds, as presented in Section 2.2.1.4, the transition from the startPersistentComps state to the idle state without adding the functionality to periodically check the status of the components as in the ACT7 transform. This transform is executed on agents with transient and persistent components. Figure 42 shows the agent component diagram after the ACT8 transform is complete.

ACT 8

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, spcs : **State** •
(c = ag.component ∧ st = c.stateTable ∧ is ∈ st.states ∧ is.name = "idle" ∧ spcs ∈ st.states ∧
spcs.name = "startPersistentComps" ∧ hasTransientComponent(ag) ∧ hasPersistentComponent(ag))
⇒
(∃ t : **Transition** •
t ∈ st'.transitions ∧ t.from = spcs ∧ t.receive = null ∧ t.receiveEvent = null ∧ t.guard = "started" ∧ t.to = is ∧
t.actions = [] ∧ t.sends = {} ∧ t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧
t.AgentID = null)

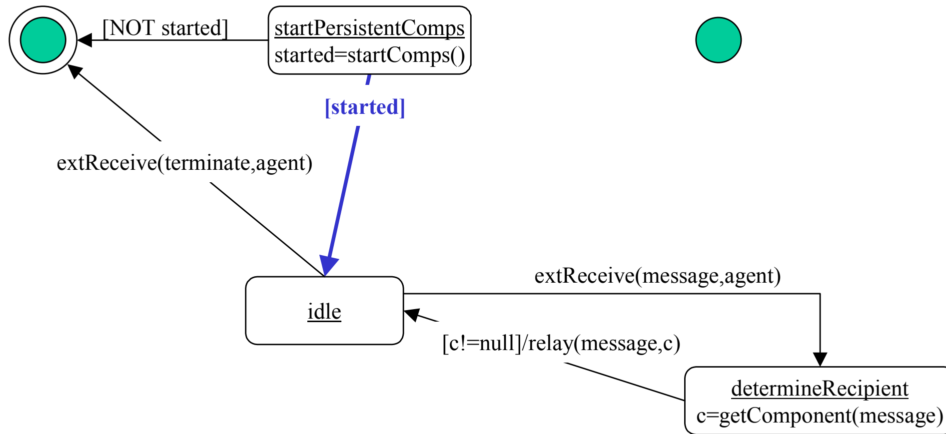


Figure 42. Agent Component Diagram After ACT8 Transformation

3.1.2.2.9 Agent Component Transform 9

Agent Component Transform 9 (ACT9) adds, as presented in Section 2.2.1.4, the transition from the start state to the startPersistentComps state, if the agent class contains at least one persistent component and no move activities, to an agent component. Figure 43 shows the agent component diagram after the ACT9 transform is complete.

ACT 9

∀ ag : **Agent**, c : **Component**, st : **StateTable**, spcs, ss : **State •**
(c = ag.component ∧ st = c.stateTable ∧ spcs ∈ st.states ∧ spcs.name = “startPersistentComps” ∧ ss ∈ st.states ∧ ss.name = “start” ∧ hasPersistentComponent(ag) ∧ ¬isMobility_Specified(ag))
⇒
(∃ t : **Transition •**
t ∈ st’.transitions ∧ t.from = ss ∧ t.receive = null ∧ t.receiveEvent = null ∧ t.guard = null ∧ t.to = spcs ∧ t.actions = [] ∧ t.sends = {} ∧ t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧ t.AgentID = null)

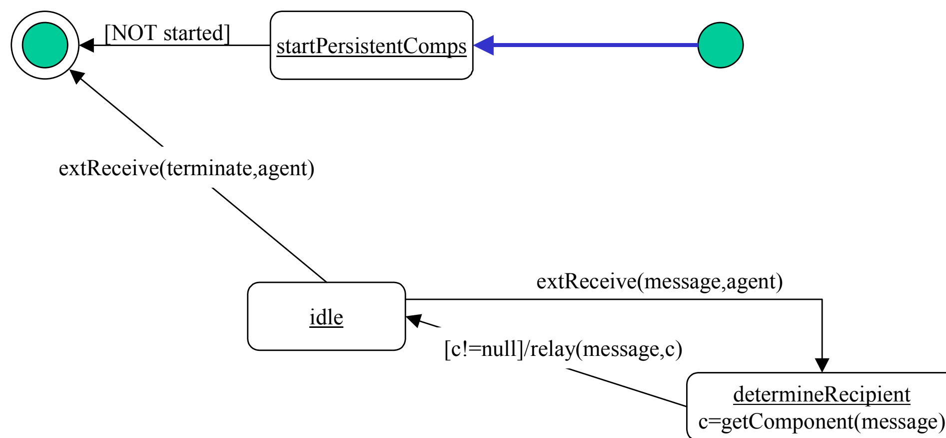


Figure 43. Agent Component Diagram After ACT9 Transformation

3.1.2.3 Summary

Analyses to design transformations were defined that prevented a move activity from being placed into a conversation and created an agent component for every agent class in the system. The transformations to create the agent component were executed based upon whether or not mobility was specified and the types of components contained within each agent class. Now that all of the analysis-to-design transformations are complete, the process to define the generic design model to mobile design model transformations can be completed.

3.1.3 Design to Design

After all of the necessary transformations are executed on the analysis models, the next set of transformations deals with adding mobility functionality to all of the components including the agent component of a mobile agent class. The first group of transformations incrementally adds mobility functionality to the agent component. Then, the transformations for adding mobility functionality to the other components belonging to a mobile agent class are defined.

3.1.3.1 Agent Component Mobility

The following sections cover the transformations that add the mobility functionality to the agent component as was described in Section 2.2.4.3. Table 2 defines which agent component mobility

transformations (ACMT) are executed for each agent class. The type of components contained within an agent class determines which ACMT transformations will be executed on that class. As shown in Table 2, the first four transformations are executed automatically while transformations 5 and 6 are executed depending on component type. Since transformations 1 through 4 are executed automatically, they could have been combined into one transformation. They were separated to ease understanding and readability.

Table 2. Agent Component Mobility Transformations

Type of Components	ACMT Transformations Needed
Transient	1, 2, 3, 4, 5
Persistent	1, 2, 3, 4, 6
Transient and Persistent	1, 2, 3, 4, 6

3.1.3.1.1 Agent Component Mobility Transform 1

Agent Component Mobility Transform 1 (ACMT1) adds, as presented in Section 2.2.4.3, the states needed for mobility to the agent component. Figure 44 shows the agent component diagram after the ACMT1 transform is complete.

ACMT 1

∀ ag : **Agent**, c : **Component**, st : **StateTable** •
(c ∈ ag.components ∧ c.name = “AgentComponent” ∧ st = c.stateTable ∧ isMobility_Specified(c))
⇒
(∃ s1, s2, s3, s4, s5, s6 : **State**, a1, a2, a3, a4, a5, a6 : **Action**, f1, f2, f3, f4, f5, f6 : **FunctionCall**,
p1, p2, p3, p4, p5, p6 : **Parameter** •
p1.name = “stateInfo” ∧ p2.name = “moved” ∧ p3.name = “reason” ∧ p4.name = “dest” ∧ p5.name = “list” ∧
p6.name = “comp” ∧ f1.name = “restore” ∧ f1.parameters = [p1,p2,p3] ∧ f2.name = “decision” ∧
f2.parameters = [] ∧ f3.name = “getCompList” ∧ f3.parameters = [] ∧ f4.name = “move” ∧
f4.parameters = [p4] ∧ f5.name = “saveState” ∧ f5.parameters = [p1] ∧ f6.name = “remove” ∧
f6.parameters = [p5,p6] ∧ a1.lhs = “compsStarted” ∧ a1.rhs = f1 ∧ a2.lhs = “reason,denied” ∧ a2.rhs = f2 ∧
a3.lhs = “list” ∧ a3.rhs = f3 ∧ a4.lhs = “reason,moved” ∧ a4.rhs = f4 ∧ a5.lhs = “stateInfo” ∧ a5.rhs = f5 ∧
a6.lhs = “list” ∧ a6.rhs = f6 ∧ s1 ∈ st’.states ∧ s1.name = “reestablish” ∧ s1.actions = [a1] ∧ s1.convIDs = {} ∧
s2 ∈ st’.states ∧ s2.name = “moveDecision” ∧ s2.actions = [a2] ∧ s2.convIDs = {} ∧ s3 ∈ st’.states ∧
s3.name = “buildComponentList” ∧ s3.actions = [a3] ∧ s3.convIDs = {} ∧ s4 ∈ st’.states ∧
s4.name = “tryMove” ∧ s4.actions = [a4] ∧ s4.convIDs = {} ∧ s5 ∈ st’.states ∧ s5.name = “wait” ∧
s5.actions = [] ∧ s5.convIDs = {} ∧ s6 ∈ st’.states ∧ s6.name = “update” ∧ s6.actions = [a5,a6] ∧
s6.convIDs = {})

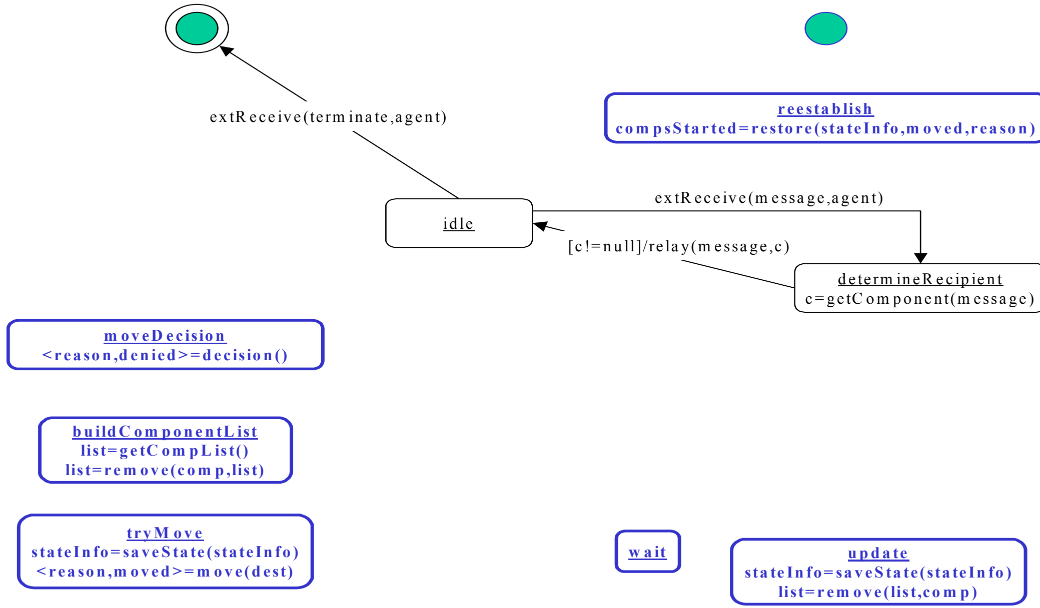


Figure 44. Agent Component Diagram After ACMT1 Transformation

3.1.3.1.2 Agent Component Mobility Transform 2

Agent Component Mobility Transform 2 (ACMT2) adds four transitions, as presented in Section 2.2.4.3, between the idle, moveDecision, buildComponentList and tryMove states to the agent component. Figure 45 shows the agent component diagram after the ACMT2 transform is complete.

ACMT 2

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, mds, bcls, tms : **State** •
(c ∈ ag.components ∧ c.name = "AgentComponent" ∧ st = c.stateTable ∧ is ∈ st.states ∧ is.name = "idle" ∧ mds ∈ st.states ∧ mds.name = "moveDecision" ∧ bcls ∈ st.states ∧ bcls.name = "buildComponentList" ∧ tms ∈ st.states ∧ tms.name = "tryMove" ∧ isMobility_Specified(c))
⇒
(∃ t1, t2, t3, t4 : **Transition**, e1, e2, e3, e4 : **Event**, p1, p2, p3, p4 : **Parameter** •
p1.name = "dest" ∧ p2.name = "comp" ∧ p3.name = "stateInfo" ∧ p4.name = "reason" ∧ e1.name = "reqMove" ∧ e1.parameters = [p1,p2,p3] ∧ e2.name = "moveDenied" ∧ e2.parameters = [p4] ∧ e3.name = "terminate" ∧ e3.parameters = [p2] ∧ e4.name = "saveAgent" ∧ e4.parameters = [p3] ∧ t1 ∈ st'.transitions ∧ t1.from = is ∧ t1.receive = e1 ∧ t1.receiveEvent = null ∧ t1.guard = null ∧ t1.to = mds ∧ t1.actions = [] ∧ t1.sends = {} ∧ t1.sendEvents = {} ∧ t1.start = false ∧ t1.end = false ∧ t1.convIDs = {} ∧ t1.AgentID = null ∧
t2 ∈ st'.transitions ∧ t2.from = mds ∧ t2.receive = null ∧ t2.receiveEvent = null ∧ t2.guard = "denied" ∧ t2.to = is ∧ t2.actions = [] ∧ t2.sends = {e2} ∧ t2.sendEvents = {} ∧ t2.start = false ∧ t2.end = false ∧ t2.convIDs = {} ∧ t2.AgentID = null ∧ t3 ∈ st'.transitions ∧ t3.from = mds ∧ t3.receive = null ∧ t3.receiveEvent = null ∧ t3.guard = "NOT denied" ∧ t3.to = bcls ∧ t3.actions = [] ∧ t3.sends = {e3} ∧ t3.sendEvents = {} ∧ t3.start = false ∧ t3.end = false ∧ t3.convIDs = {} ∧ t3.AgentID = null ∧
t4 ∈ st'.transitions ∧ t4.from = bcls ∧ t4.receive = null ∧ t4.receiveEvent = null ∧ t4.guard = "size(list)<=0" ∧ t4.to = tms ∧ t4.actions = [] ∧ t4.sends = {e4} ∧ t4.sendEvents = {} ∧ t4.start = false ∧ t4.end = false ∧ t4.convIDs = {} ∧ t4.AgentID = null)

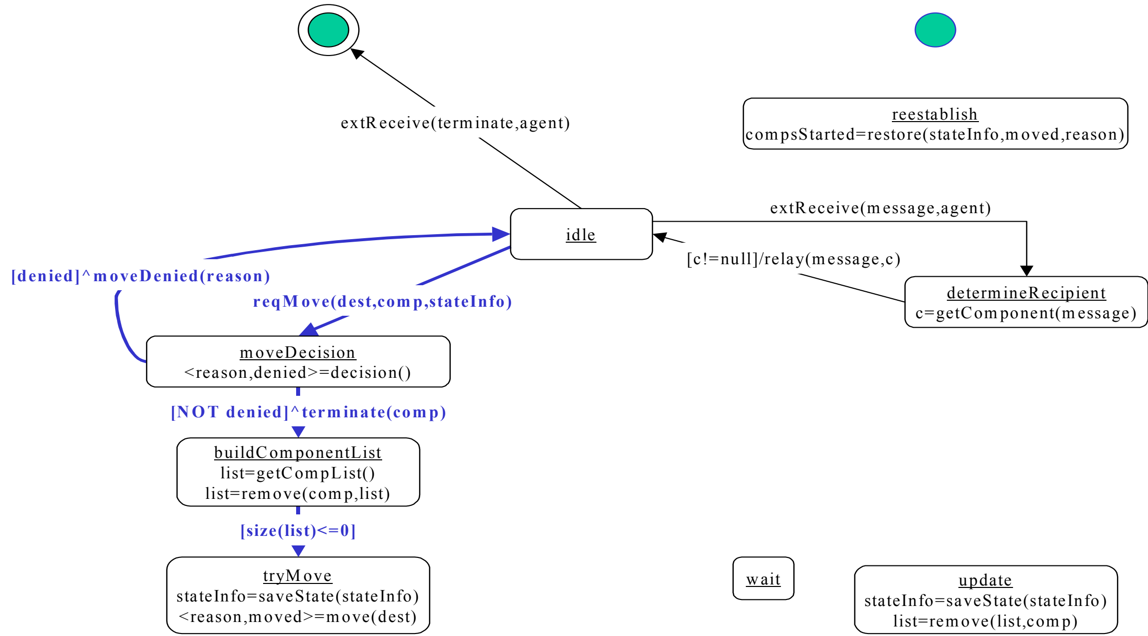


Figure 45. Agent Component Diagram After ACMT2 Transformation

3.1.3.1.3 Agent Component Mobility Transform 3

Agent Component Mobility Transform 3 (ACMT3) adds four transitions, as presented in Section 2.2.4.3, between the buildComponentList, tryMove, wait and update states to the agent component. Figure 46 shows the agent component diagram after the ACMT3 transform is complete.

<u>ACMT 3</u>

∀ ag : **Agent**, c : **Component**, st : **StateTable**, bcls, tms, ws, us : **State** •
(c ∈ ag.components ∧ c.name = “AgentComponent” ∧ st = c.stateTable ∧ bcls ∈ st.states ∧
bcls.name = “buildComponentList” ∧ tms ∈ st.states ∧ tms.name = “tryMove” ∧ ws ∈ st.states ∧
ws.name = “wait” ∧ us ∈ st.states ∧ us.name = “update” ∧ isMobility_Specified(c))
⇒
(∃ t1, t2, t3, t4 : **Transition**, a : **Action**, e : **Event**, f : **FunctionCall**, p1, p2, p3, p4 : **Parameter** •
p1.name = “moveReq” ∧ p2.name = “list” ∧ p3.name = “stateInfo” ∧ p4.name = “comp” ∧
f.name = “broadCast” ∧ f.parameters = [p1,p2] ∧ e.name = “ready” ∧ e1.parameters = [p3,p4] ∧ a1.lhs = null ∧
a1.rhs = f ∧ t1 ∈ st’.transitions ∧ t1.from = bcls ∧ t1.receive = null ∧ t1.receiveEvent = null ∧
t1.guard = “size(list)>0” ∧ t1.to = ws ∧ t1.actions = [a] ∧ t1.sends = {} ∧ t1.sendEvents = {} ∧ t1.start = false ∧
t1.end = false ∧ t1.convIDs = {} ∧ t1.AgentID = null ∧ t2 ∈ st’.transitions ∧ t2.from = ws ∧ t2.receive = e ∧
t2.receiveEvent = null ∧ t2.guard = null ∧ t2.to = us ∧ t2.actions = [] ∧ t2.sends = {} ∧ t2.sendEvents = {} ∧
t2.start = false ∧ t2.end = false ∧ t2.convIDs = {} ∧ t2.AgentID = null ∧ t3 ∈ st’.transitions ∧ t3.from = us ∧
t3.receive = null ∧ t3.receiveEvent = null ∧ t3.guard = null ∧ t3.to = ws ∧ t3.actions = [] ∧ t3.sends = {} ∧
t3.sendEvents = {} ∧ t3.start = false ∧ t3.end = false ∧ t3.convIDs = {} ∧ t3.AgentID = null ∧
t4 ∈ st’.transitions ∧ t4.from = ws ∧ t4.receive = null ∧ t4.receiveEvent = null ∧ t4.guard = “size(list)<=0” ∧
t4.to = tms ∧ t4.actions = [] ∧ t4.sends = {} ∧ t4.sendEvents = {} ∧ t4.start = false ∧ t4.end = false ∧
t4.convIDs = {} ∧ t4.AgentID = null)

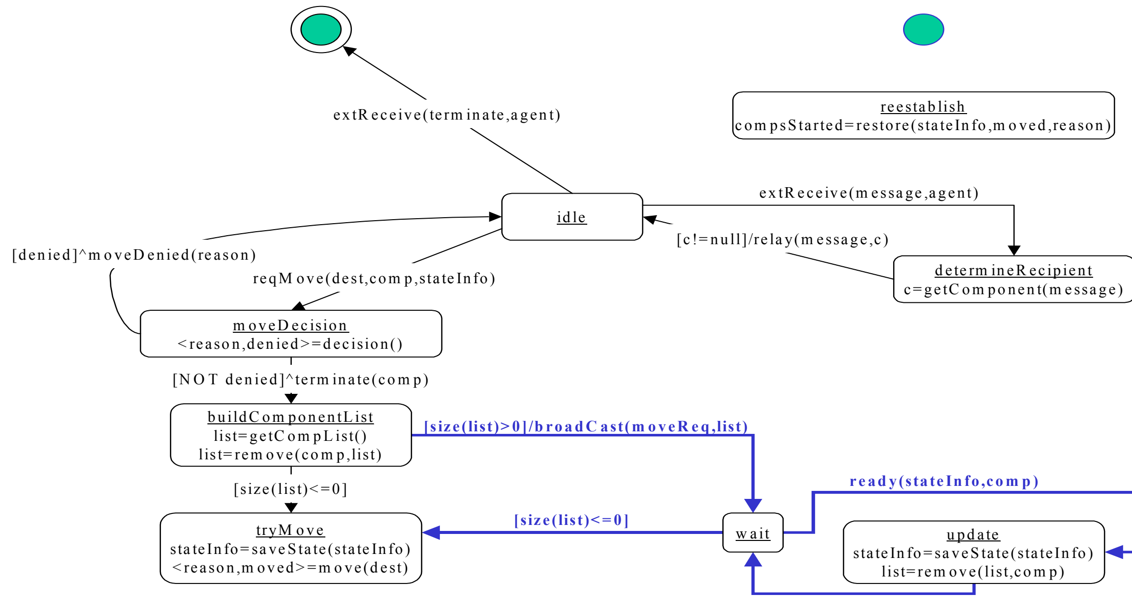


Figure 46. Agent Component Diagram After ACMT3 Transformation

3.1.3.1.4 Agent Component Mobility Transform 4

Agent Component Mobility Transform 4 (ACMT4) adds five transitions, as presented in Section 2.2.4.3, between the tryMove, reestablish, start, end and idle states to the agent component. Figure 47 shows the agent component diagram after the ACMT4 transform is complete.

ACMT 4

∀ ag : **Agent**, c : **Component**, st : **StateTable**, es, is, rs, ss, tms : **State** •
(c ∈ ag.components ∧ c.name = "AgentComponent" ∧ st = c.stateTable ∧ tms ∈ st.states ∧
tms.name = "tryMove" ∧ es ∈ st.states ∧ es.name = "end" ∧ ss ∈ st.states ∧ ss.name = "start" ∧ rs ∈ st.states ∧
rs.name = "reestablish" ∧ is ∈ st.states ∧ is.name = "idle" ∧ isMobility_Specified(ag))
⇒
(∃ t1, t2, t3, t4, t5 : **Transition** •
t1 ∈ st'.transitions ∧ t1.from = tms ∧ t1.receive = null ∧ t1.receiveEvent = null ∧ t1.guard = "moved" ∧
t1.to = es ∧ t1.actions = [] ∧ t1.sends = {} ∧ t1.sendEvents = {} ∧ t1.start = false ∧ t1.end = false ∧
t1.convIDs = {} ∧ t1.AgentID = null ∧ t2 ∈ st'.transitions ∧ t2.from = tms ∧ t2.receive = null ∧
t2.receiveEvent = null ∧ t2.guard = "NOT moved" ∧ t2.to = rs ∧ t2.actions = [] ∧ t2.sends = {} ∧
t2.sendEvents = {} ∧ t2.start = false ∧ t2.end = false ∧ t2.convIDs = {} ∧ t2.AgentID = null ∧
t3 ∈ st'.transitions ∧ t3.from = ss ∧ t3.receive = null ∧ t3.receiveEvent = null ∧ t3.guard = "stateInfo!=null" ∧
t3.to = rs ∧ t3.actions = [] ∧ t3.sends = {} ∧ t3.sendEvents = {} ∧ t3.start = false ∧ t3.end = false ∧
t3.convIDs = {} ∧ t3.AgentID = null ∧ t4 ∈ st'.transitions ∧ t4.from = rs ∧ t4.receive = null ∧
t4.receiveEvent = null ∧ t4.guard = "compsStarted" ∧ t4.to = is ∧ t4.actions = [] ∧ t4.sends = {} ∧
t4.sendEvents = {} ∧ t4.start = false ∧ t4.end = false ∧ t4.convIDs = {} ∧ t4.AgentID = null ∧
t5 ∈ st'.transitions ∧ t5.from = rs ∧ t5.receive = null ∧ t5.receiveEvent = null ∧
t5.guard = "NOT compsStarted" ∧ t5.to = es ∧ t5.actions = [] ∧ t5.sends = {} ∧ t5.sendEvents = {} ∧
t5.start = false ∧ t5.end = false ∧ t5.convIDs = {} ∧ t5.AgentID = null)

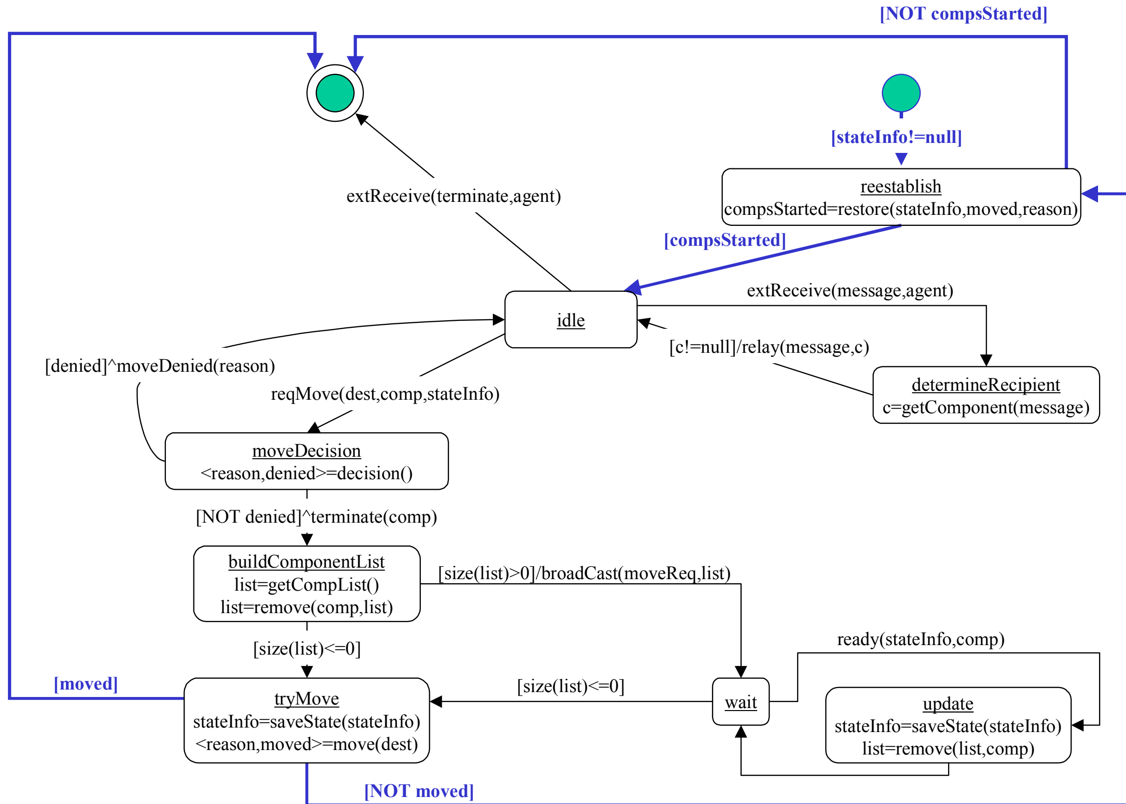


Figure 47. Agent Component Diagram After ACMT4 Transformation

3.1.3.1.5 Agent Component Mobility Transform 5

Agent Component Mobility Transform 5 (ACMT5) adds, as presented in Section 2.2.4.3, a final transition from the start state to the idle state, if the agent class contains only transient components, to the agent component. Figure 48 shows the agent component diagram after the ACMT5 transform is complete.

<u>ACMT 5</u>

∀ ag : **Agent**, c : **Component**, st : **StateTable**, is, ss : **State** •
(c = ag.component ∧ st = c.stateTable ∧ ss ∈ st.states ∧ ss.name = “start” ∧ is ∈ st.states ∧ is.name = “idle” ∧ isMobility_Specified(c) ∧ ¬hasPersistentComponent(ag))
⇒
(∃ t : **Transition** •
t ∈ st’.transitions ∧ t.from = ss ∧ t.receive = null ∧ t.receiveEvent = null ∧ t.guard = “stateInfo==null” ∧ t.to = is ∧ t.actions = [] ∧ t.sends = {} ∧ t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧ t.AgentID = null)

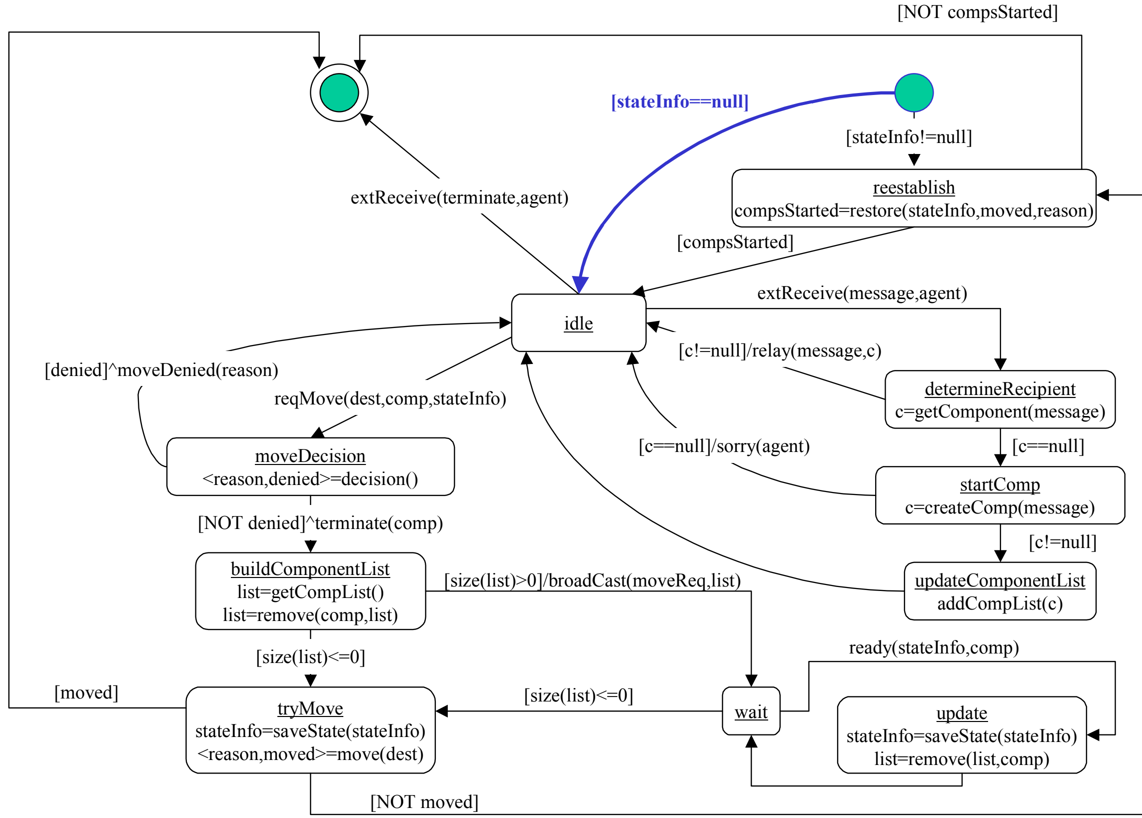


Figure 48. Agent Component Diagram After ACMT5 Transformation

3.1.3.1.6 Agent Component Mobility Transform 6

Agent Component Mobility Transform 6 (ACMT6) adds, as presented in Section 2.2.4.3, a final transition from the start state to the startPersistentComps state, if the agent class contains at least one persistent component, to the agent component. Figure 49 shows the agent component diagram after the ACMT6 transform is complete.

ACMT 6

∀ ag : **Agent**, c : **Component**, st : **StateTable**, spcs, ss : **State** •
(c = ag.component ∧ st = c.stateTable ∧ ss ∈ st.states ∧ ss.name = "start" ∧ spcs ∈ st.states ∧
spcs.name = "startPersistentComps" ∧ isMobility_Specified(c) ∧ hasPersistentComponent(ag))
⇒
(∃ t : **Transition** •
t ∈ st'.transitions ∧ t.from = ss ∧ t.receive = null ∧ t.receiveEvent = null ∧ t.guard = "stateInfo==null" ∧
t.to = spcs ∧ t.actions = [] ∧ t.sends = {} ∧ t.sendEvents = {} ∧ t.start = false ∧ t.end = false ∧ t.convIDs = {} ∧
t.AgentID = null)

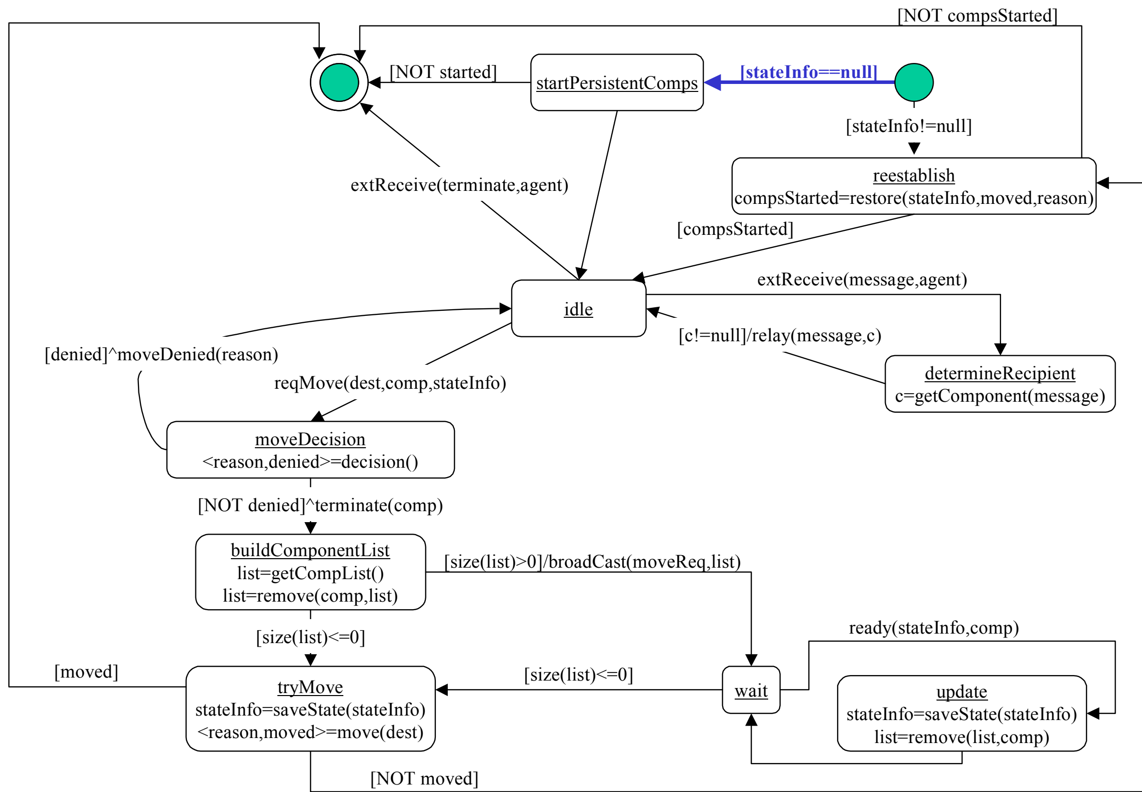


Figure 49. Agent Component Diagram After ACMT6 Transformation

3.1.3.2 Component Mobility

Finally, each non-agent component for a mobile agent class is transformed according to the table below. Even though most of the transformations are executed for each component type, they were broken into smaller pieces for ease of readability.

Table 3. Component Mobility Transformations

Component	Persistent	Transient	# of mobile	End	CMT Transformations Needed
Mobile	Yes	No	Yes	Yes	3, 4, 5, 6, 7, 8, 9, 10
|	|	|	↓	No	1, 3, 4, 5, 6, 7, 8, 9, 10
|	|	|	No	Yes	3, 4, 5, 6, 8, 9
|	↓	↓	↓	No	1, 3, 4, 5, 6, 8, 9
|	No	Yes	Yes/No	Yes	3, 4, 5, 6, 7, 8, 9, 10
↓	↓	↓	↓	No	1, 3, 4, 5, 6, 7, 8, 9, 10
Non-Mobile	Yes/No	Yes/No	Yes/No	Yes	2, 7, 9, 10
↓	Yes/No	Yes/No	Yes/No	No	1, 2, 7, 9, 10

3.1.3.2.1 Component Mobility Transform 1

Component Mobility Transform 1 (CMT1) adds an end state to any component that belongs to a mobile agent that does not already have an end state. This end state is unreachable at this point but is needed due to the fact that all components belonging to a mobile agent will need to terminate regardless of component type.

CMT 1

∀ ag : **Agent**, c : **Component**, st : **StateTable**, s : **State** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ isMobility_Specified(ag) ∧ ¬(∃ s : State • s ∈ st.states ∧ s.name ≠ "EndState"))
⇒
(∃ es : **State** • es ∈ st'.parameters ∧ es ∉ st.parameters ∧ es.name = "EndState")

3.1.3.2.2 Component Mobility Transform 2

Component Mobility Transform 2 (CMT2) adds a transition from the start state to the restore state for all non-mobile components belonging to a mobile agent class. The transition from the start state to the restore state is needed to ensure that the component is started correctly after an agent move has occurred. This transition is an internal message containing the stateInfo that the component saved before it was terminated for the move. The restart function in the restore state takes the stateInfo that is passed into the component upon creation and determines the starting state for the component. The value of the Boolean variable state is used to transition from the restore state to the proper state for continued execution. Figure 50 shows the restore state and transition that are added to a non-mobile component by CMT2.

CMT 2

∀ ag : **Agent**, c : **Component**, st : **StateTable**, ss : **State** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ ss ∈ st.states ∧
ss.name = "StartState" ∧ isMobility_Specified(ag) ∧ ¬isMobility_Specified_Component(c))
⇒
(∃ s : **State**, t : **Transition**, e : **Event**, a : **Action**, f : **FunctionCall**, p : **Parameter** •
p.name = "stateInfo" ∧ f.name = "restart" ∧ f.parameters = [p] ∧ a.lhs = "state" ∧ a.rhs = f ∧ e.name = "start" ∧
e.parameters = [p] ∧ s ∈ st'.states ∧ s ∉ st.states ∧ s.name = "restore" ∧ s.actions = [a] ∧ t ∈ st'.transitions ∧
t ∉ st.transitions ∧ t.from = ss ∧ t.receive = e ∧ t.receiveEvent = null ∧ t.guard = null ∧ t.to = s ∧ t.actions = [] ∧
t.sends = {} ∧ t.sendEvents = {} ∧ t.start = null ∧ t.end = null ∧ t.convIDs = {} ∧ t.AgentID = null)

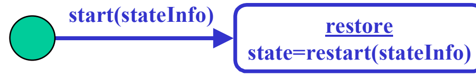


Figure 50. State and Transition Added to Non-Mobile Components by CMT2

3.1.3.2.3 Component Mobility Transform 3

Component Mobility Transform 3 (CMT3) adds with the same transition a different restore state to all mobile components in a mobile agent class. The checkLocation function in the restore state returns the current address for the agent. The result of this function is used in conjunction with the stateInfo to determine the starting state for the component. If the stateInfo is null then the agent has not moved. But if stateInfo is not null then the agent has moved and the state of the component was saved prior to that move. The Boolean variable state is used to transition from the restore state to the proper starting state for the component. Figure 51 shows the restore state and transition that are added to a mobile component by CMT3.

CMT 3

∀ ag : **Agent**, c : **Component**, st : **StateTable**, ss : **State** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ ss ∈ st.states ∧
ss.name = "StartState" ∧ isMobility_Specified(ag) ∧ isMobility_Specified_Component(c))
⇒
(∃ s : **State**, t : **Transition**, e : **Event**, a1, a2 : **Action**, f1, f2 : **FunctionCall**, p1, p2 : **Parameter** •
p1.name = "stateInfo" ∧ p2.name = "currentLocation" ∧ f1.name = "checkLocation" ∧ f1.parameters = [] ∧
f2.name = "restart" ∧ f2.parameters = [p1,p2] ∧ a1.lhs = "currentLocation" ∧ a1.rhs = f1 ∧
a2.lhs = "state,reason" ∧ a2.rhs = f2 ∧ e.name = "start" ∧ e.parameters = [p1] ∧ s ∈ st'.states ∧ s ∉ st.states ∧
s.name = "restore" ∧ s1.actions = [a1,a2] ∧ t ∈ st'.transitions ∧ t ∉ st.transitions ∧ t.from = ss ∧ t.receive = e ∧
t.receiveEvent = null ∧ t.guard = null ∧ t.to = s ∧ t.actions = [] ∧ t.sends = {} ∧ t.sendEvents = {} ∧
t.start = null ∧ t.end = null ∧ t.convIDs = {} ∧ t.AgentID = null)

Figure 51. State and Transition Added to Mobile Components by CMT3

3.1.3.2.4 Component Mobility Transform 4

Component Mobility Transform 4 (CMT4) adds a moveCalled state and wait state to every mobile component belonging to a mobile agent class. The saveCompState function in the moveCalled state saves the state of the component into the variable stateInfo when a move is called by that component. The

function getCompName returns the name of the component that is passed to the agent when the component calls for a move. The wait state is used to suspend the component until it receives either a terminate message which means that the move request was accepted or a move denied message. Figure 52 shows the moveCalled and wait states that are added to a mobile component by CMT4.

CMT 4

∀ ag : **Agent**, c : **Component**, st : **StateTable** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ isMobility_Specified(ag) ∧ isMobility_Specified_Component(c))
⇒
(∃ s1, s2 : **State**, a1, a2 : **Action**, f1, f2 : **FunctionCall** •
f1.name = "saveCompState" ∧ f1.parameters = [] ∧ f2.name = "getCompName" ∧ f2.parameters = [] ∧
a1.lhs = "stateInfo" ∧ a1.rhs = f1 ∧ a2.lhs = "comp" ∧ a2.rhs = f2 ∧ s1 ∈ st'.states ∧ s1 ∉ st.states ∧
s1.name = "moveCalled" ∧ s1.actions = [a1,a2] ∧ s2 ∈ st'.states ∧ s2 ∉ st.states ∧ s2.name = "wait" ∧
s2.actions = [])

Figure 52. States Added by CMT4 for Mobile Components

3.1.3.2.5 Component Mobility Transform 5

Component Mobility Transform 5 (CMT5) adds the transitions that link the states created in CMT4 and the end state to every mobile component belonging to a mobile agent class. The reqMove internal message contains the destination address, the component name and stateInfo of the component and is sent to the agent component when a move is needed by a component. Once the move has been requested the component waits for a confirmation or denial message. The moveDenied message is the denial message while the terminate message is the confirmation message. Figure 53 shows the reqMove, moveDenied and terminate messages that are added to a mobile component by CMT5.

CMT 5

∀ ag : **Agent**, c : **Component**, st : **StateTable**, mcs, ws, es : **State** •
(c ∈ ag.components ∧ c.name ≠ “AgentComponent” ∧ st = c.stateTable ∧ mcs ∈ st.states ∧
mcs.name = “StartState” ∧ ws ∈ st.states ∧ ws.name = “wait” ∧ es ∈ st.states ∧ es.name = “end” ∧
isMobility_Specified(ag) ∧ isMobility_Specified_Component(c))
⇒
(∃ t1, t2, t3 : **Transition**, e1, e2, e3, e4 : **Event**, p1, p2, p3, p4 : **Parameter** •
p1.name = “dest” ∧ p2.name = “comp” ∧ p3.name = “stateInfo” ∧ p4.name = “reason” ∧ e1.name = “reqMove” ∧
e1.parameters = [p1,p2,p3] ∧ e2.name = “moveDenied” ∧ e2.parameters = [p4] ∧ e3.name = “sorry” ∧
e3.parameters = [p4] ∧ e4.name = “terminate” ∧ e4.parameters = [p2] ∧ t1 ∈ st’.transitions ∧ t1 ∉ st.transitions ∧
t1.from = mcs ∧ t1.receive = null ∧ t1.receiveEvent = null ∧ t1.guard = null ∧ t1.to = ws ∧ t1.actions = [] ∧
t1.sends = {e1} ∧ t1.sendEvents = {} ∧ t1.start = null ∧ t1.end = null ∧ t1.convIDs = {} ∧ t1.AgentID = null ∧
t2 ∈ st’.transitions ∧ t2 ∉ st.transitions ∧ t2.from = ws ∧ t2.receive = e2 ∧ t2.receiveEvent = null ∧
t2.guard = null ∧ t2.to = es ∧ t2.actions = [] ∧ t2.sends = {e3} ∧ t2.sendEvents = {} ∧ t2.start = null ∧
t2.end = true ∧ t2.ConvIDs = {} ∧ t2.AgentID = null ∧ t3 ∈ st’.transitions ∧ t3 ∉ st.transitions ∧ t3.from = ws ∧
t3.receive = e4 ∧ t3.receiveEvent = null ∧ t3.guard = null ∧ t3.to = es ∧ t3.actions = [] ∧ t3.sends = {e4} ∧
t3.sendEvents = {} ∧ t3.start = null ∧ t3.end = true ∧ t3.ConvIDs = {} ∧ t3.AgentID = null)

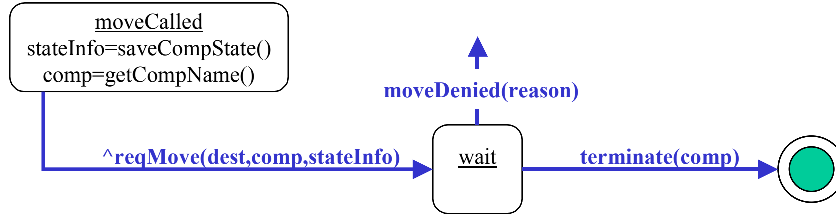


Figure 53. Transitions Added to Mobile Components by CMT5

3.1.3.2.6 Component Mobility Transform 6

Component Mobility Transform 6 (CMT6) modifies the From state and guard condition for all transitions that have a From state that is a state containing a move activity in a mobile component. The From state is changed to the restore state that was created by CMT3 and the guard condition is changed to a concatenation of the string “state==” and the name of the To state. Figure 54 shows the transitions before and the transitions after modification by CMT6.

CMT 6

∀ ag : **Agent**, c : **Component**, st : **StateTable**, rs, s : **State** •
(c ∈ ag.components ∧ c.name ≠ “AgentComponent” ∧ st = c.stateTable ∧ rs ∈ st.states ∧ rs.name = “restore” ∧
s ∈ st.states ∧ isMobility_Specified_State(s) ∧ t ∈ st.transitions ∧ t.from = s ∧ isMobility_Specified(ag) ∧
isMobility_Specified_Component(c))
⇒
(t’.to = rs ∧ t’.guard = “state==”+(t.to).name)

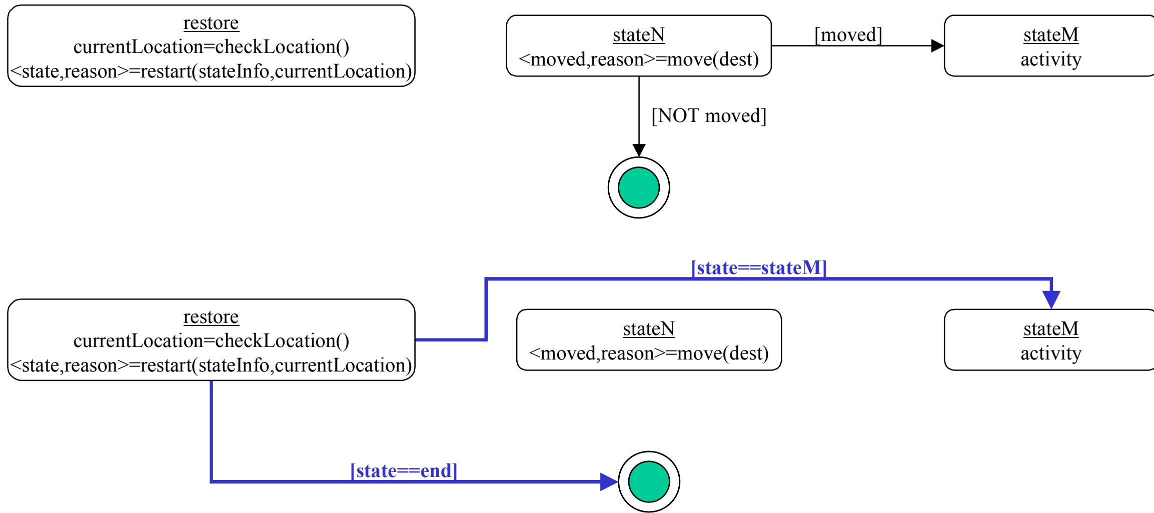


Figure 54. Transitions Altered in Mobile Components by CMT6

3.1.3.2.7 Component Mobility Transform 7

Component Mobility Transform 7 (CMT7) adds a state and transition to partially handle the process of components, belonging to a mobile agent, receiving move required messages from the agent component. There are three cases for modifying the components of a mobile agent to be able to receive move-required messages from the agent component:

1. If there is only one mobile component and that component is a persistent component, then only the non-mobile components are transformed.
2. If there is only mobile component and that component is a transient component, then all of the components are transformed.
3. If there are two or more mobile components, then all the components are transformed.

The moveReceived state is necessary to ensure that each component of a mobile agent can respond to an internal moveReq message from the agent component. Finally, the ready transition is needed to inform the agent that the component is terminating and to pass the components' current state to the agent component. This state information is used to restart the component in the proper state at the new address.

Figure 55 shows the moveReceived state and transition that are added to components of a mobile agent class by CMT7.

<u>CMT 7</u>

∀ ag : **Agent**, c : **Component**, st : **StateTable**, es : **State** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ es ∈ st.states ∧
es.name = "EndState" ∧ isMobility_Specified(ag) ∧ (isTransientComponent(c) ∨
!isMobility_Specified_Component(c) ∨
((mobileComponentCount(ag) > 1) ∧ isMobility_Specified_Component(c))))
⇒
(∃ s : **State**, t : **Transition**, e : **Event**, a1, a2 : **Action**, f1, f2 : **FunctionCall**, p1, p2 : **Parameter** •
p1.name = "stateInfo" ∧ p2.name = "comp" ∧ f1.name = "saveCompState" ∧ f1.parameters = [] ∧
f2.name = "getCompName" ∧ f2.parameters = [] ∧ a1.lhs = "stateInfo" ∧ a1.rhs = f1 ∧ a2.lhs = "comp" ∧
a2.rhs = f2 ∧ e.name = "ready" ∧ e.parameters = [p1,p2] ∧ s ∈ st'.states ∧ s ∉ st.states ∧
s.name = "moveReceived" ∧ s.actions = [a1,a2] ∧ t ∈ st'.transitions ∧ t ∉ st.transitions ∧ t.from = s ∧
t.receive = null ∧ t.receiveEvent = null ∧ t.guard = null ∧ t.to = es ∧ t.actions = [] ∧ t.sends = {e} ∧
t.sendEvents = {} ∧ t.start = null ∧ t.end = null ∧ t.convIDs = {} ∧ t.AgentID = null)

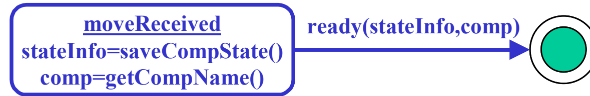


Figure 55. State and Transition Added to a Component by CMT7

3.1.3.2.8 Component Mobility Transform 8

Component Mobility Transform 8 (CMT8) adds a null transition from the state containing a move activity to the moveCalled state that was created in CMT4. Figure 56 shows the null transition added to a mobile component by CMT8.

<u>CMT 8</u>

∀ ag : **Agent**, c : **Component**, st : **StateTable**, mcs, s : **State** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ mcs ∈ st.states ∧
mcs.name = "moveCalled" ∧ s ∈ st.states ∧ isMobility_Specified_State(s))
⇒
(t ∈ st'.transitions ∧ t ∉ st.transitions ∧ t.from = s ∧ t.to = mcs)

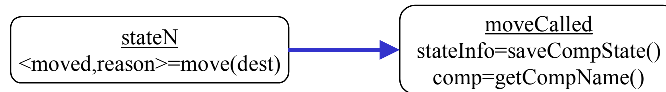


Figure 56. Transition Added to Mobile Components by CMT8

3.1.3.2.9 Component Mobility Transform 9

Component Mobility Transform 9 (CMT9) modifies the transitions originating from the start state. Every transition from the start state is modified to start from the restore states added in either CMT2 or CMT3. The guard conditions on those transitions are changed to the following string, “state==”, concatenated with the name of the To state of the transition. Figure 57 shows a transition modified by CMT9.

CMT 9

$\forall$ ag : **Agent**, c : **Component**, st : **StateTable**, ss, rs : **State** $\bullet$
(c $\in$ ag.components $\wedge$ c.name $\neq$ “AgentComponent” $\wedge$ st = c.stateTable $\wedge$ ss $\in$ st.states $\wedge$
ss.name = “StartState” $\wedge$ rs $\in$ st.states $\wedge$ rs.name = “restore” $\wedge$ t $\in$ st.transitions $\wedge$ t.from = ss $\wedge$ isMobility_Specified(ag) $\wedge$ isMobility_Specified_Component(c))
$\Rightarrow$
(t’.from = rs $\wedge$ t’.guard = “state==”+(t.to).name)

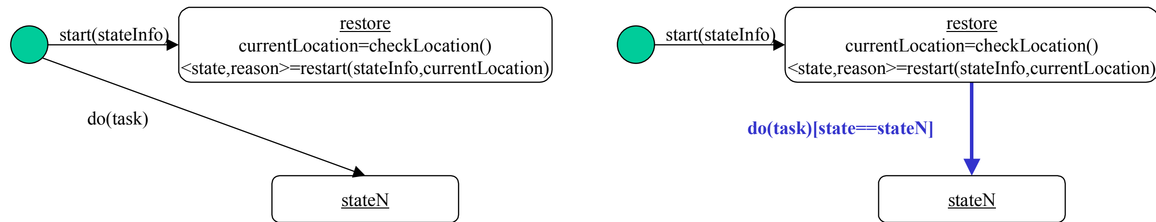


Figure 57. Transition Altered in Components by CMT9

3.1.3.2.10 Component Mobility Transform 10

Component Mobility Transform 10 (CMT10) adds transitions to handle the process of a component receiving a “move required” message from the agent component. Every valid state in the component is available to be selected by the designer for the CMT10 transform. After the designer has selected the states in the component where a move required message has to be received, each of those selected states will have a transition added from that state to the moveReceived state created in CMT7. Then a transition is added from the restore state to those selected states with the guard condition, “state==” concatenated with the name of each selected state. Figure 58 shows a transition added by CMT10.

CMT 10

∀ ag : **Agent**, c : **Component**, st : **StateTable**, t : **Transition**, s, rs, mrs : **State** •
(c ∈ ag.components ∧ c.name ≠ "AgentComponent" ∧ st = c.stateTable ∧ s ∈ st.states ∧
s.selected = true ∧ rs ∈ st.states ∧ rs.name = "restoreState" ∧ mrs ∈ st.states ∧ mrs.name = "moveReceived" ∧
t ∈ st.transitions ∧ t.from = rs ∧ t.to ≠ s ∧ (isTransientComponent(c) ∨ !isMobility_Specified_Component(c) ∨
((mobileComponentCount(ag) > 1) ∧ isMobility_Specified_Component(c))))
⇒
(∃ t1, t2 : **Transition**, e : **Event** •
e.name = "moveReq" ∧ e.parameters = [] ∧ t1 ∈ st'.transitions ∧ t1 ∉ st.transitions ∧ t1.from = s ∧
t1.receive = e ∧ t1.receiveEvent = null ∧ t1.guard = null ∧ t1.to = mrs ∧ t1.actions = [] ∧ t1.sends = {} ∧
t1.sendEvents = {} ∧ t1.start = null ∧ t1.end = null ∧ t1.convIDs = {} ∧ t1.AgentID = null ∧
t1.receiveEvent = null ∧ t2 ∈ st'.transitions ∧ t2 ∉ st'.transitions ∧ t2.from = s ∧ t2.receive = e ∧
t2.receiveEvent = null ∧ t2.guard = "state==" + s.name ∧ t2.to = mrs ∧ t2.actions = [] ∧ t2.sends = {} ∧
t2.sendEvents = {} ∧ t2.start = null ∧ t2.end = true ∧ t2.ConvIDs = {} ∧ t2.AgentID = null)

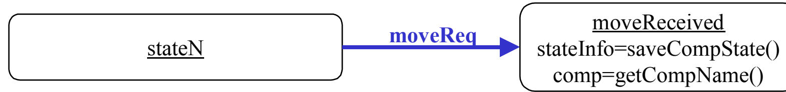


Figure 58. Transition Added to a Component by CMT10

3.1.3.2.11 Component Mobility Transform 11

Component Mobility Transform 11 (CMT11) removes all move activities within the mobile components of every agent class. Since the agent component has been given the responsibility of requesting a move from the agent platform, the move activities within the mobile components are no longer needed.

CMT 11

∀ s : **State**, a : **Action**, f : **FunctionCall** •
(isMobility_Specified_State(s) ∧ a ∈ s.actions ∧ a.rhs = f ∧ f.name = "move")
⇒
(a ∉ s'.actions)

3.2 Summary

This chapter described the process used to incorporate mobility in the MaSE methodology. Formal predicate logic equations were used to present the transformations that finished the process of generating the generic MaSE design models from the analysis models and also generated the mobile design models from the generic design models. The generic design models were finished by making sure that the move activities remained in the components during the transformation process defined by Sparkman [18]

and by adding an agent component to every agent class. Transforming the generic design models into mobile design models was accomplished by adding mobility to the new agent component and then adding required mobility functionality to all components belonging to a mobile agent class. Chapter IV demonstrates the operation of these transformations, as well as taking generating mobile design models for an example system and converting those models into software code to be executed within the Carolina mobile agent platform.

IV. Demonstration

Demonstrating the solution presented in Chapter III involved taking an example problem through the entire MaSE process that included mobility functionality. First, the problem was taken through the MaSE analysis phase, which showed that mobility was included only as activities within concurrent task diagrams. Second, the analysis models were transformed into design models using the transformation process defined by Sparkman [18] and the analysis-to-design transformations defined in Chapter III. Third, the generic design models were transformed into mobile design models by the design-to-design transformations defined in Chapter III. Finally, the mobile and non-mobile design models were translated into software code and the resulting program was executed within the Carolina mobile agent system.

Section 4.1 describes the problem. This problem had to be detailed enough to allow for a mobile agent with two mobile components. Section 4.2 presents the MaSE analysis of the problem while Section 4.3 presents the MaSE design that incorporates the mobility changes made to agentTool. Finally, Section 4.4 presents the Carolina implementation with screen captures that show the actual execution of the software solution.

4.1 Travel Planning System (TPS)

People are constantly traveling either for business or leisure reasons. Much of this travel is through the air. Normally people use travel agencies to handle the arrangements for their trip. But with the explosion of the Internet, more and more people are handling their own travel arrangements. These arrangements can include not just airline reservations but also reservations for a hotel and rental car. Rather than a person having to put all these arrangements together themselves, a software system using mobile agents could handle this task.

All the user would have to do is input the basic travel information for the trip. Then, a mobile agent could be dispatched that would make all of the required reservations automatically and report back the completed itinerary to the user. The mobile agent would travel to places containing the necessary agents to make the reservations.

4.2 MaSE/AgentTool Analysis

Since analysis option 1 was chosen, the MaSE analysis for the TPS system is shown starting with the Refining Roles step. The role model diagram shown in Figure 59 displays all of the roles, tasks and protocols in the TPS system. The UserInterface role is the bridge between the user and the system. Once the user has input the basic flight and rental car information, the reserveFlight and reserveRentalCar roles attempt to make the proper reservations with the airline and rentalCarAgency roles respectively. The reservation or failure information is displayed to the user through the UserInterface role.

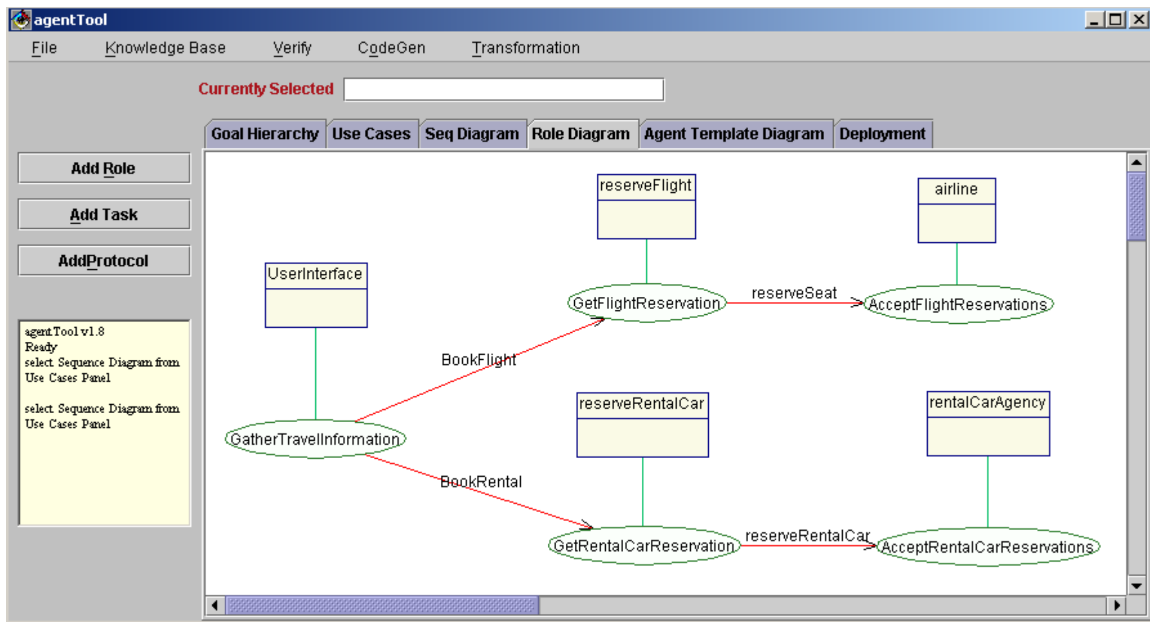


Figure 59. MaSE Role Model for TPS System

Each of the tasks diagrams is presented and explained except for the GetRentalCarReservation and AcceptRentalCarReservations tasks. These tasks are virtually identical to the GetFlightReservation and AcceptFlightReservations tasks respectively, so only the GetFlightReservation and AcceptFlightReservations tasks will be presented.

The GatherTravelInformation task under the UserInterface role is the catalyst for starting the system so it will be presented first. Figure 60 shows the GatherTravelInformation task diagram. It is a persistent heterogeneous task because it starts with a null transition to a beginning state and then generates

travel-planning requests once it receives input from the user. This input is separated into flight and rental car information and is sent to the reserveFlight and reserveRentalCar roles respectively. A counter is also started in order to determine whether all the results have been received. Reservations or failures are then received and the final compiled results are displayed to the user. After displaying the results the GatherTravelInformation task transitions back to the idle state to wait for another travel planning request from the user.

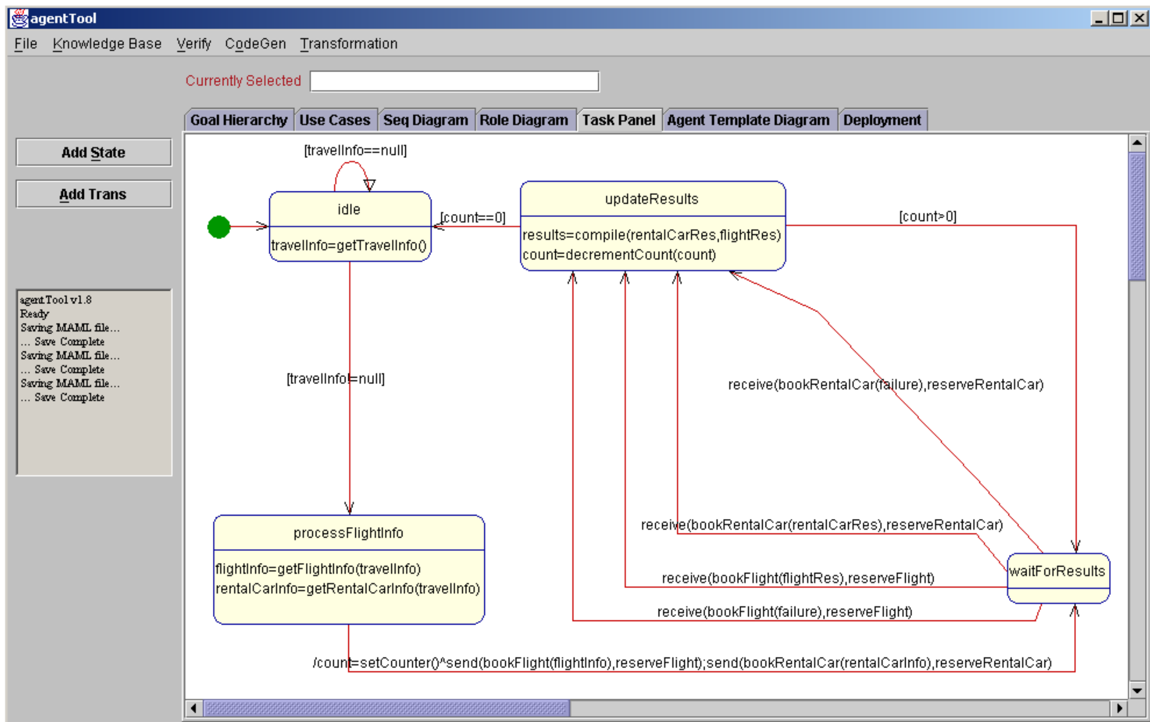


Figure 60. Gather Travel Information Task for TPS System

The GetFlightReservation task, shown in Figure 61, is started by a flight information message received from the UserInterface agent. Along with information pertaining to reserving a flight, the flight information also contains a list of places containing airline agents. This list is used in the getDestination activity in the moveNeeded state to determine whether the agent needs to move to another place. If the agent is required to move, the move activity in the tryMove state is executed. Once the agent has moved the seat request is prepared and sent to the airline agent. Once the approved reservation has been received a

reservation message is sent to the UserInterface agent for display. If the airline agent cannot make the reservation, a failure message is received and the task transitions back to the moveNeeded state to see if the list of places is not empty. If the list is not empty, the reserveFlight agent attempts to move to the next place and make a reservation. This process continues until either a reservation has been made or the list of places is empty. Once the list is empty a failure message is sent to the UserInterface agent for display.

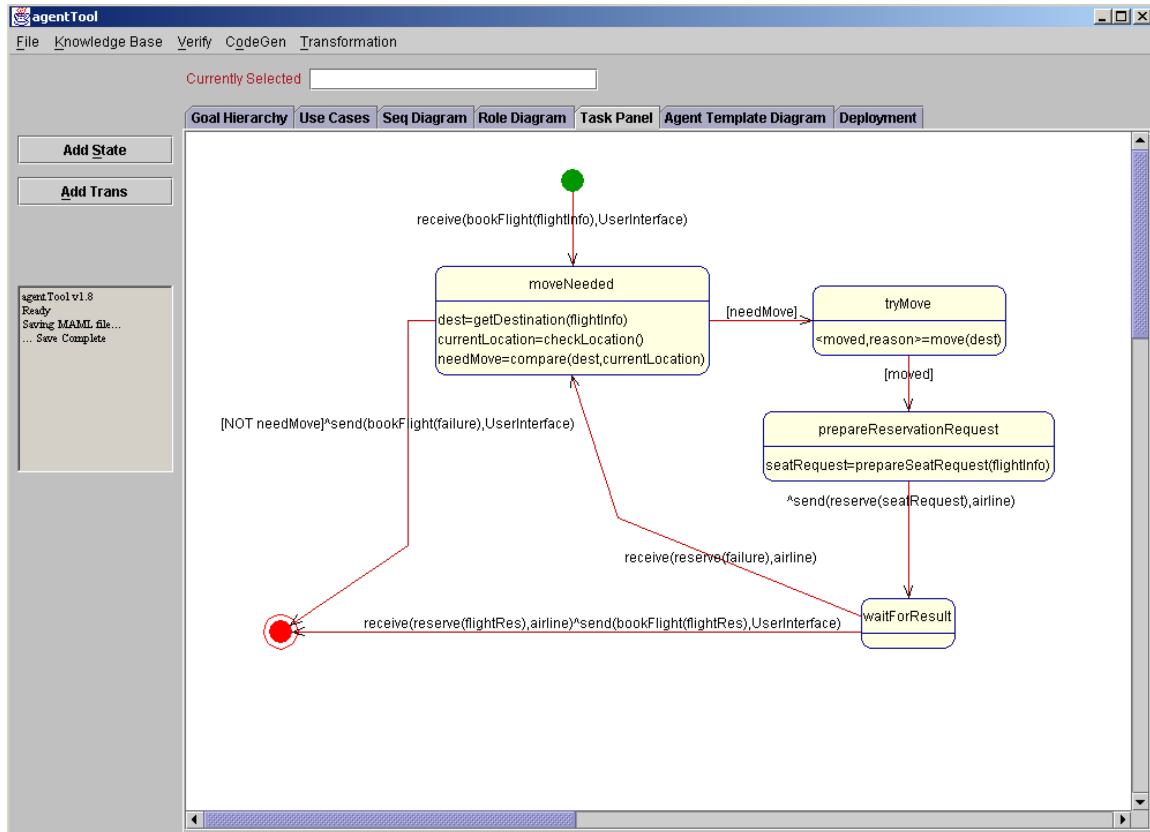


Figure 61. GetFlightReservation Task for TPS System

The AcceptFlightReservations task is the final analysis phase task to be presented and explained and is shown in Figure 62. This task is a persistent reactive task that starts with a null transition into an idle state and remains there until a reserve message is received. If a reservation is made, a reserve message is sent to the GetFlightReservation agent who forwards the reservation to the UserInterface agent. If, however, a reservation is not made, a failure message is sent to the GetFlightReservation agent.

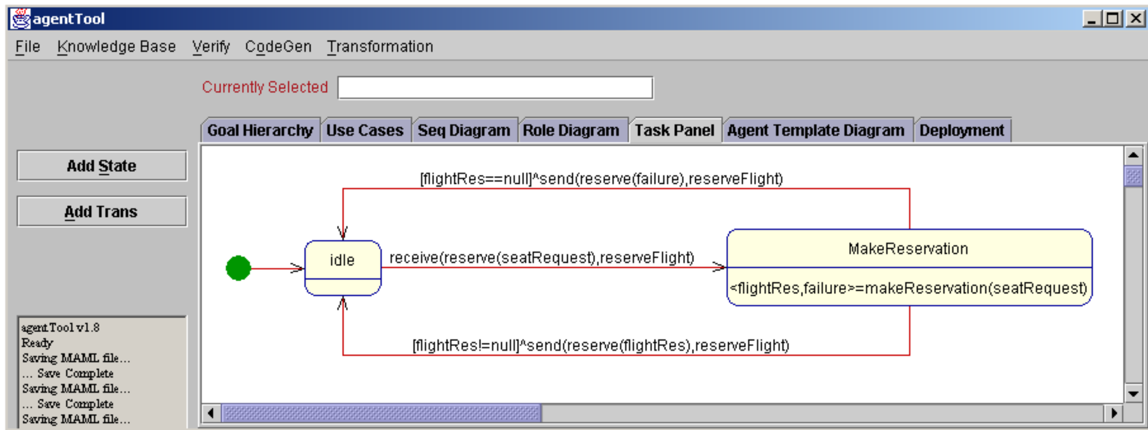


Figure 62. AcceptFlightReservations Task for TPS System

This concludes the analysis of the TPS system within MaSE. As was shown, the concept of mobility was included only as an activity within the tryMove state in the GetFlightReservation and GetRentalCarReservation task diagrams. In the next section that covers the MaSE design phase, these analysis models are transformed into design models. Then, an agent component is created for each of the agents in the system according to the transformations defined in Section 3.2.2.2. Finally, the agent and other components belonging to any mobile agent classes in the TPS are transformed to handle mobility according to the transformations defined in Section 3.2.3.

4.3 MaSE/AgentTool Mobile Design

Design of the TPS system using MaSE consisted of three steps. First, the agent classes were identified in the Creating Agent Classes step. Secondly, in the Construction Conversations and Assembling Agent Classes steps, the transformations defined by Sparkman [18] in conjunction with the analysis-to-design transformations defined in Chapter III, were executed on the TPS analysis models, that included the agent classes from the first step, to construct the components from the tasks and harvest the conversations from those components. Finally, the mobility transformations defined in Chapter III were executed to finish the Assembling Agent Classes design step.

4.3.1 Creating Agent Classes

The first step was to create the agent classes for the TPS system. Figure 63 shows the agent classes. A UserInterface agent class was created that contains the InterfaceUser role. The reserveFlight and reserveRentalCar roles were combined into the PlanTravel agent class. The airline role was placed into the Airliner agent class and the rentalCarAgency role was placed into the CarRental agent class.

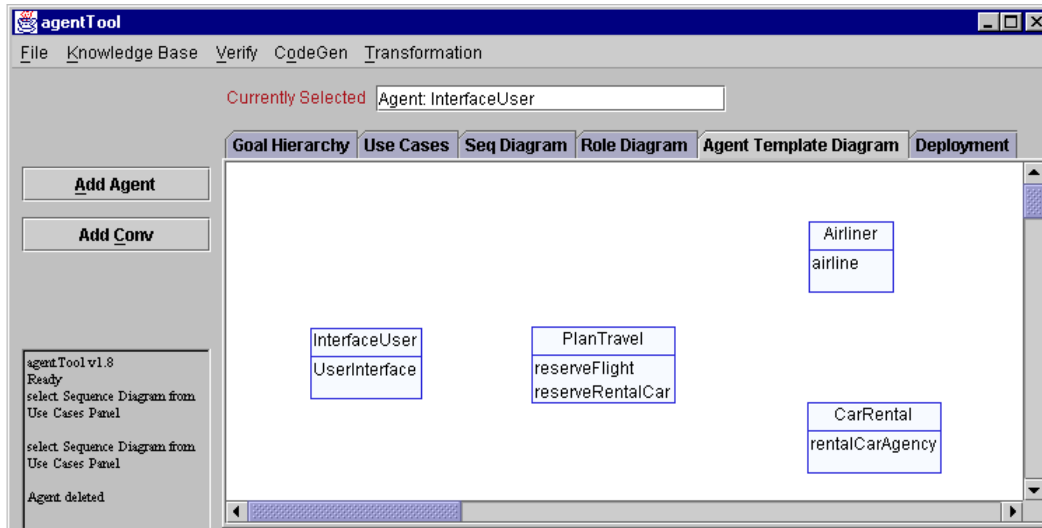


Figure 63. Agent Classes for TPS System

4.3.2 Constructing Conversations/Assembling Agent Classes

Next, the transformations defined by Sparkman [18], including the transformation defined in 3.1.2.1, were executed on the analysis models and agent class definitions. Then, the mobility transformations defined by Chapter III were executed on the resulting analysis models. All of the transformations defined in Chapter III are available on the main agentTool menu bar under the Transformations heading. As shown in Figure 64, the analysis to design transformations defined in Section 3.2.2 are grouped together under the menu heading of Create Agent Component, the agent component mobility transformations defined in Section 3.2.3.1 are grouped under the menu heading of Agent Component Mobility and the component mobility transformations defined in Section 3.2.3.2 are grouped under the menu heading of Component Mobility.

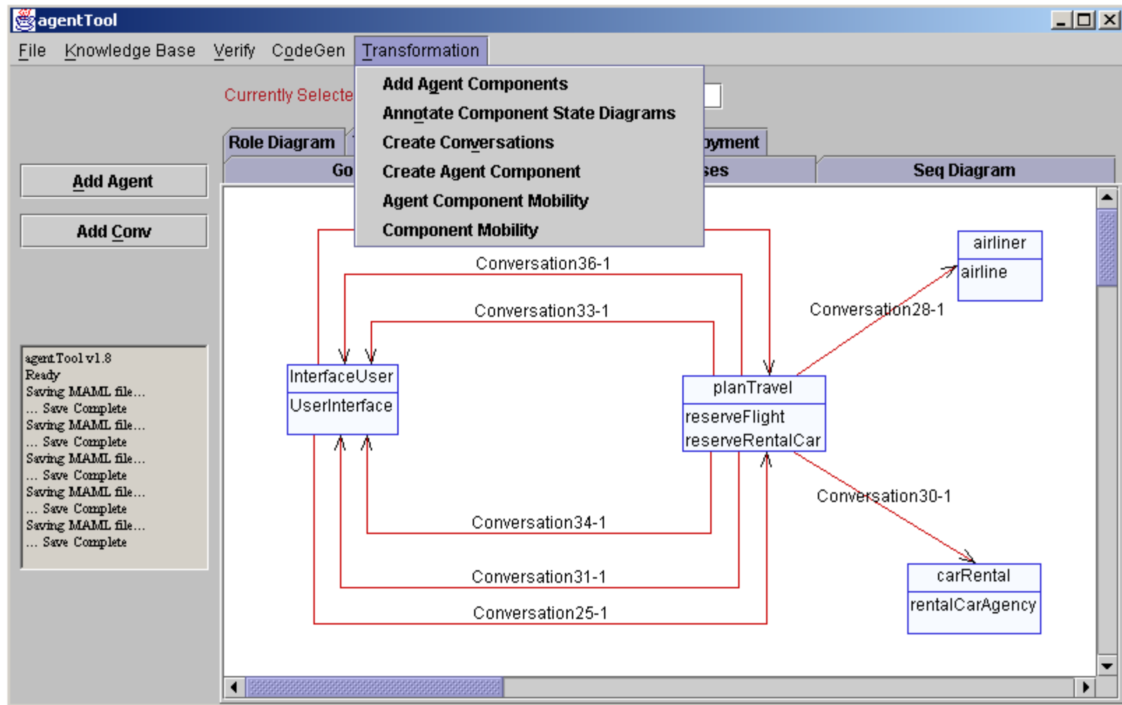


Figure 64. Transformation Menu in agentTool

4.3.2.1 Create Agent Component

After selecting the Create Agent Component option on the Transformation menu bar, the agent component transformations were executed, which created an agent component for every agent class in the system. A high level view of the agent component created for the InterfaceUser agent class that plays the UserInterface role is shown in Figure 65. The GatherTravelInformation, AcceptFlightReservations and AcceptRentalCarReservations components are all non-mobile persistent proactive components. The agent component diagram created for these components, as described in Section 2.2.1.1.3, is the same for all three agent classes and is shown in Figure 66.

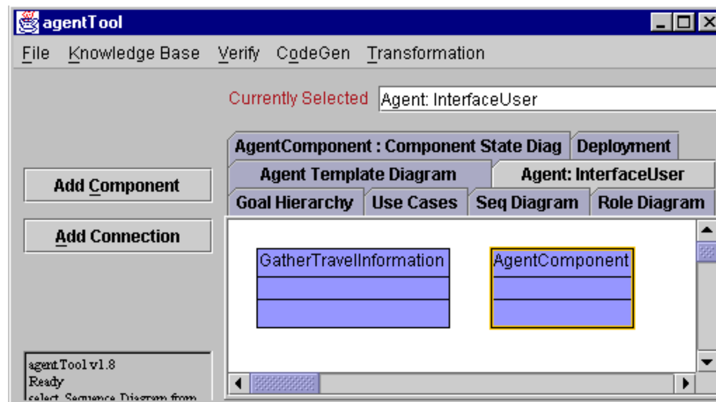


Figure 65. Components of InterfaceUser Agent Class for TPS System

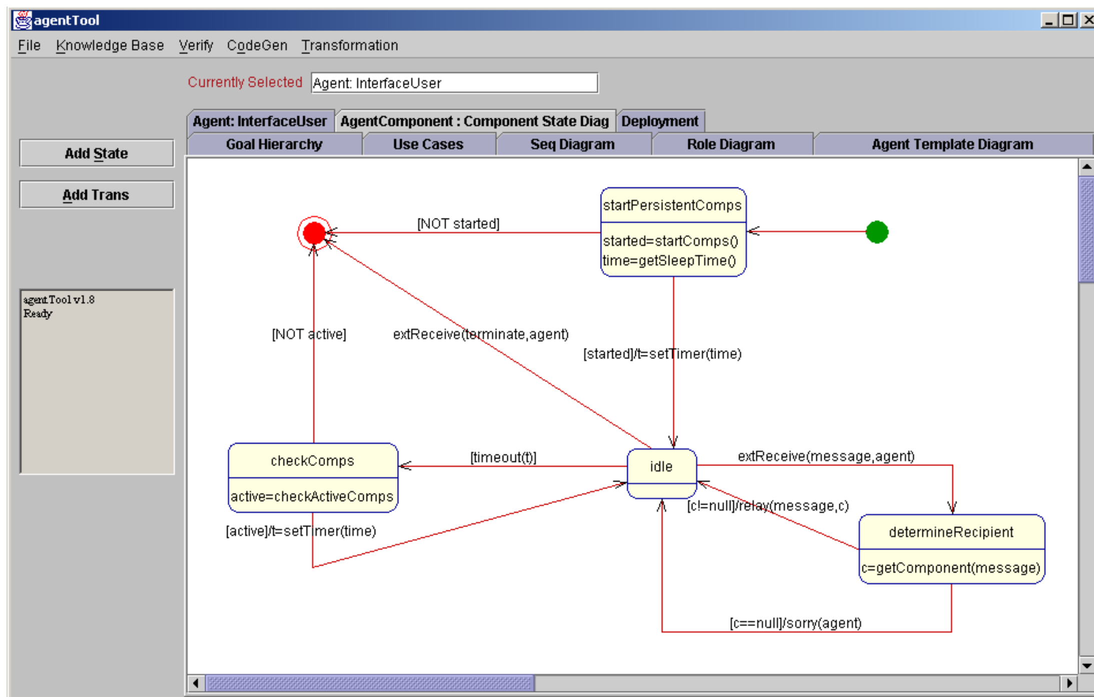


Figure 66. Agent Component for InterfaceUser Agent Class

An agent component was also created for the PlanTravel agent class by the agent component transformations. This component is shown in Figure 67. This is a transient agent component, as described in Section 2.2.1.1.2, since the GetFlightReservation and GetRentalCarReservation components are transient reactive components. Also, there is no transition from the start state. Since the PlanTravel agent class has

mobility specified in its components, the agent component will be transformed by the agent component mobility transformations, which will add a transition from the start state to idle state.

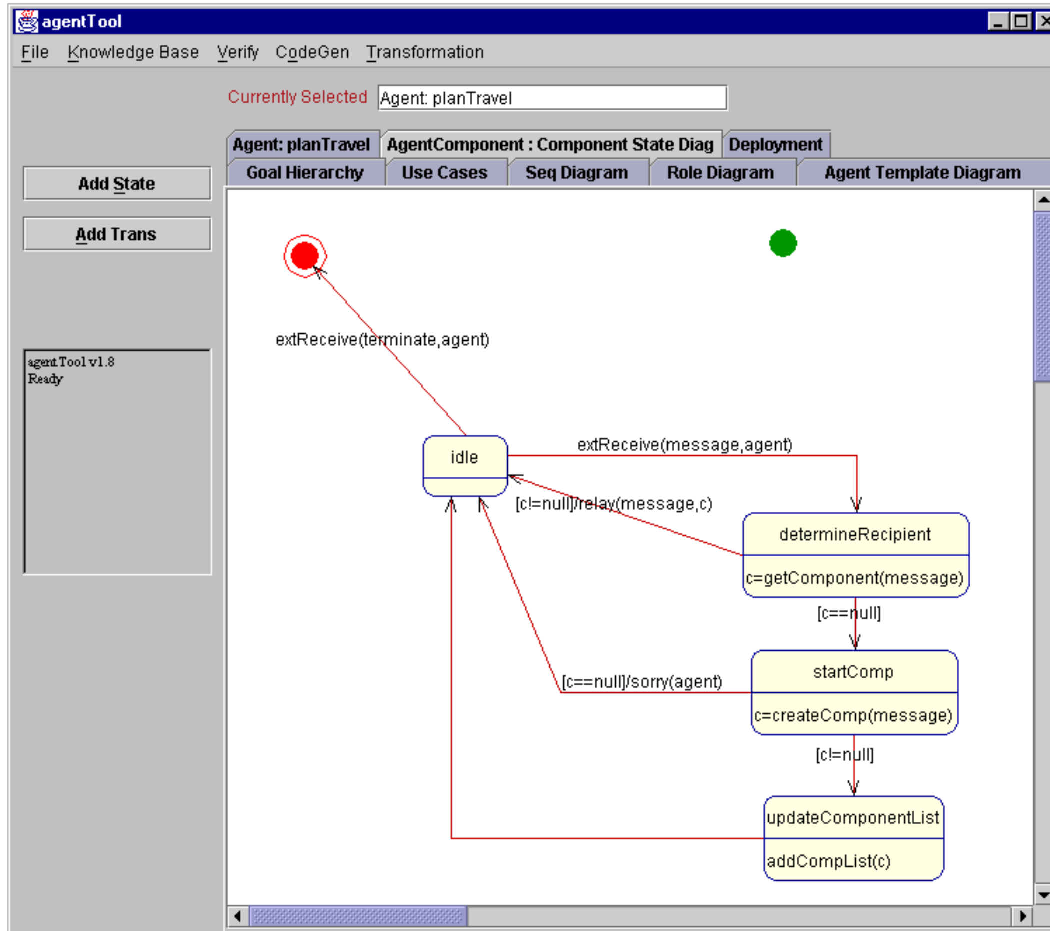


Figure 67. Agent Component for Plan Travel Agent Class

This concludes the changes to the analysis models by the agent component transformations. Next, mobility is added to the agent components of all mobile agent classes in the system by the agent component mobility transformations.

4.3.2.2 Agent Component Mobility

In the TPS system, only the PlanTravel agent class is mobile. After selecting the Agent Component Mobility option on the Transformation menu bar, the agent component mobility

transformations were executed to add mobility functionality to the agent component of the PlanTravel agent class. The resulting agent component diagram, as described in Section 3.2.3.1.5, is shown in Figure 68.

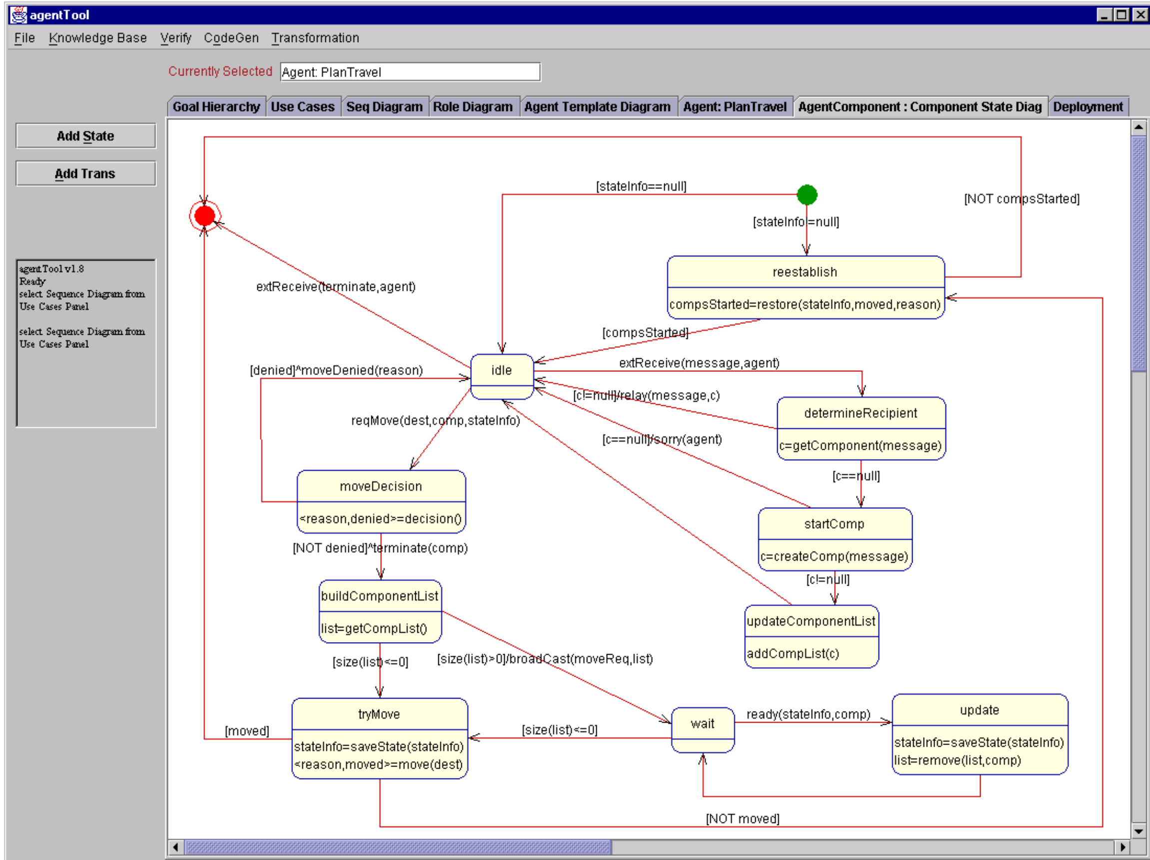


Figure 68. Agent Component for PlanTravel Agent Class in TPS System After Agent Component Mobility Transformations

This concludes the changes made to the agent components by the agent component mobility transformations. Finally, mobility is added to the non-agent components of all mobile agent classes in the system by the component mobility transformations.

4.3.2.3 Component Mobility

In the TPS system only the GetFlightReservation and GetRentalCarReservation components have mobility specified. After selecting the Component Mobility option on the Transformation menu bar,

mobility functionality was added to these two components by the component mobility transformations. Before showing the resulting GetFlightReservation component, Figure 69 shows the GetFlightReservation component after the transformations defined in [18] were executed. Notice that the move activity and state containing that activity have not been changed whatsoever. Only the agent component has been created based upon the type of the components contained within the agent class. The GetRentalCarReservation component will not be shown because it is essentially the same as the GetFlightReservation component.

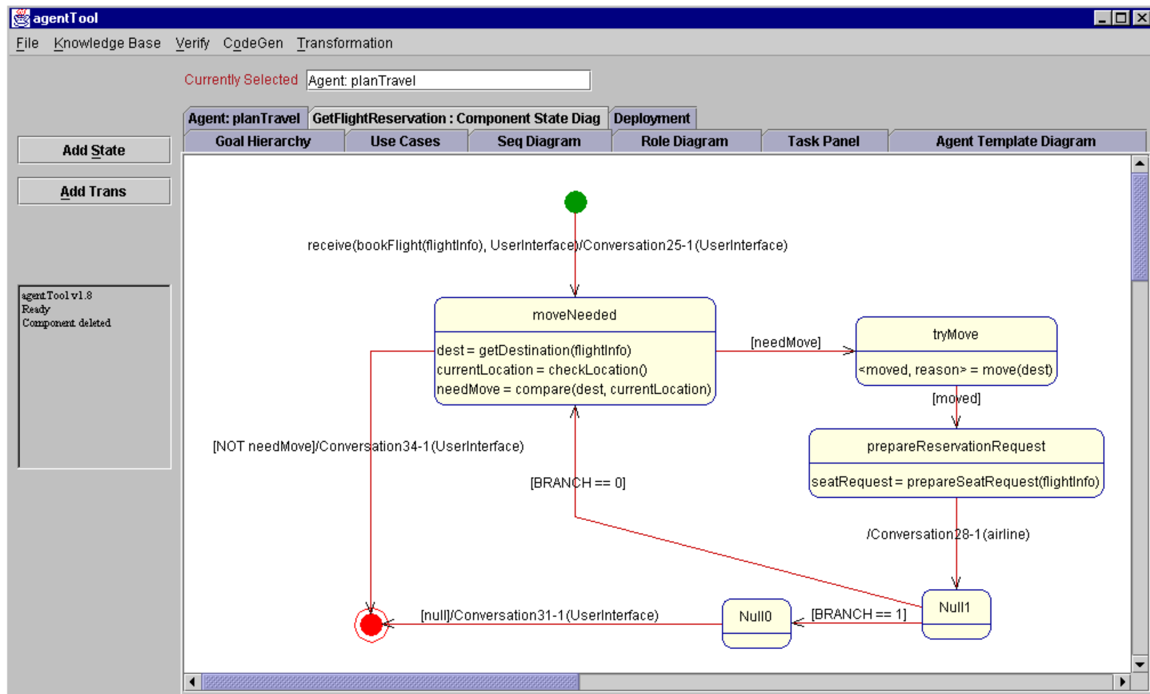


Figure 69. GetFlightReservation Component after Transformations Defined by [18]

The agent component and agent component mobility transformations require no input from the designer of the system. But during the component mobility transformations, the designer could be asked to select states that can receive a move-required message. The rules governing whether the designer will be asked were presented in Section 3.2.3.2.7. The dialog box shown in Figure 70 informs the designer of the need to select states for a given component.

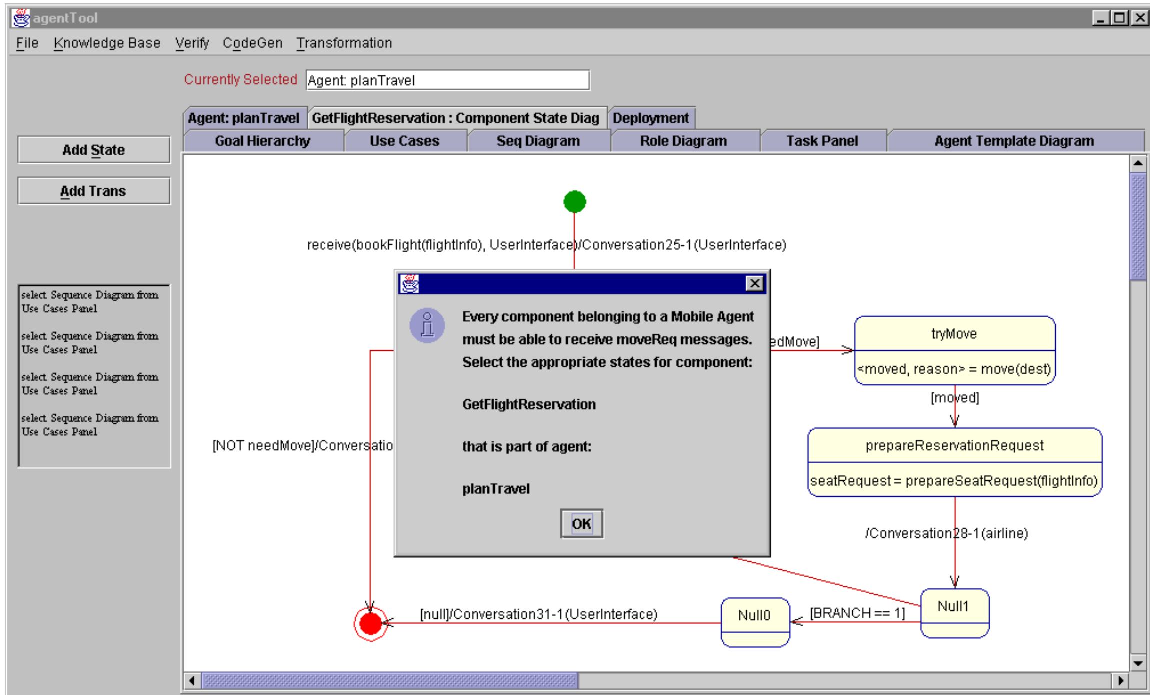


Figure 70. Informational Dialog before Selecting States to Receive Move Required Messages

Once the designer has pressed the "OK" button on the dialog box the state selection dialog box is shown with the states that are allowed to receive a move-required message. The designer must select at least one state. Figure 71 shows the state selection window for selecting states for the GetFlightReservation component. For demonstration purposes, the prepareReservationRequest state was chosen to receive a move-required message. Figure 72 shows the completed GetFlightReservation component after all the component mobility transformations have been executed.

This concludes the analysis of the TPS system using the MaSE methodology that incorporated mobility. In the next section, this design is transformed into software code that executed within the Carolina mobile agent platform.

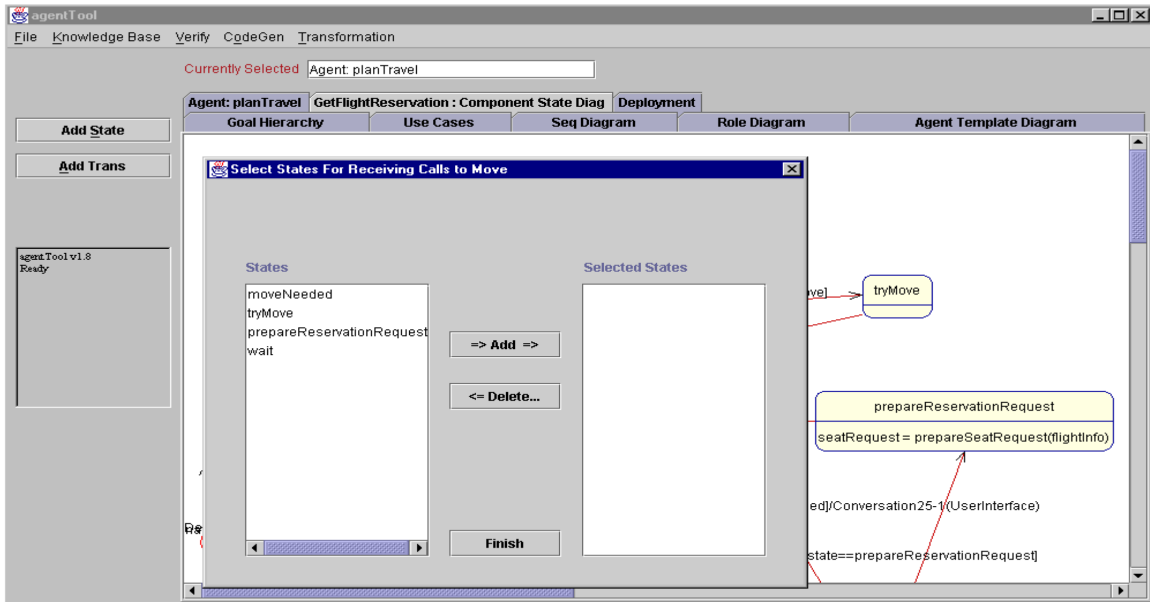


Figure 71. State Selection Window

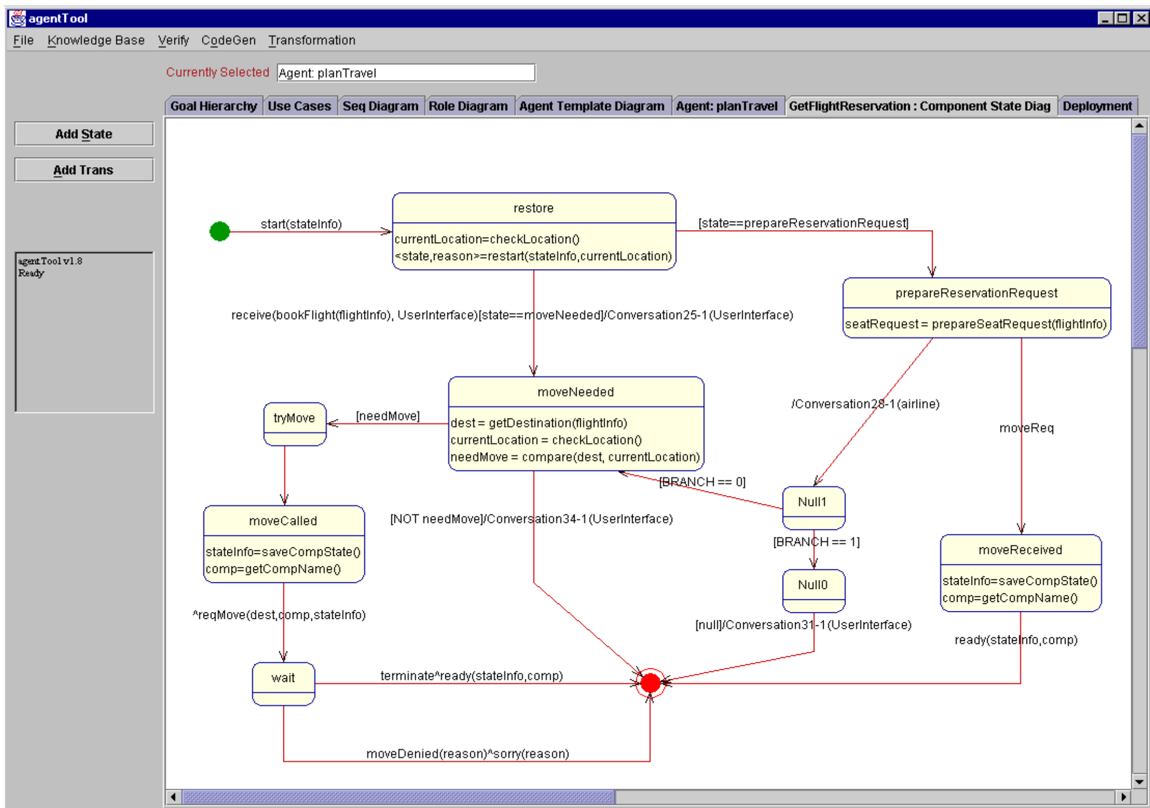


Figure 72. GetFlightReservation Component After Component Mobility Transformations

4.4 Carolina Implementation

For the Carolina implementation, the mobile design models generated by the transformations defined in Chapter III for the TPS system were converted into software code. The agent component for each agent class in the system was an extension of the Carolina base agent class. A special component class was created for the other components in an agent class. The conversations, however, were generated automatically by agentTool but required slight modifications to correct errors in the code generation process.

On startup of the TPS system, the InterfaceUser and PlanTravel agents are started, on a Carolina agent platform, at the same address. The GatherTravelInformation component belonging to the InterfaceUser agent is also started on the same address as the InterfaceUser agent. The Airline and CarRental agents are started on Carolina agent platforms throughout the network. All of these agents transition to their respective idle states upon creation and wait for a travel request to plan.

The InterfaceUser agent incorporated a graphical user interface (GUI) in order to get the necessary requirements for travel from the user. Figure 73 shows the GUI for the TPS system. The user inputs the start city, end city, start time, end time, day of travel, airline choices and rental car agency choices and then clicks the Plan Itinerary button to start the planning.

Once the button has been pressed the InterfaceUser agent sends the travel information to the PlanTravel agent. The PlanTravel agent starts the GetFlightReservation and GetRentalCarReservation components based on the information received from the InterfaceUser agent. These components, based on the information received, starts the planning processes.

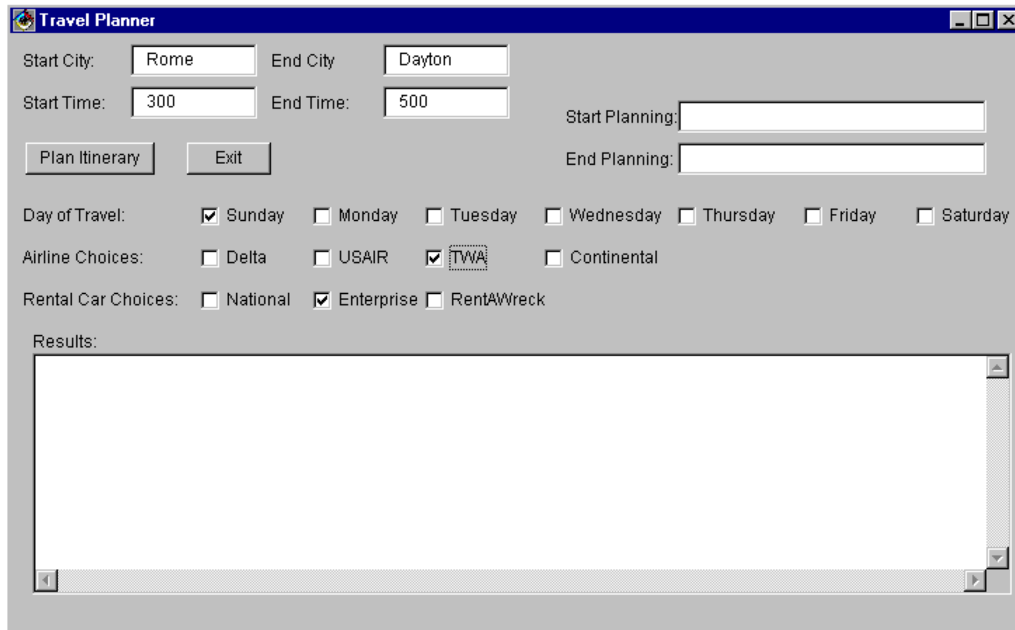


Figure 73. Travel Planner Graphical User Interface for TPS System

In this example, the Airline and CarRental agents are located on different addresses. So, the first component to decide to move (either GetFlightReservation or GetRentalCarReservation) sends that move request to the PlanTravel agent. The PlanTravel agent automatically accepts the move request, informs the requesting component to terminate and informs the other component to save state and terminate.

The output on the user's machine is shown in Figure 74. In this case, the GetRentalCarReservation component requested a move first so the PlanTravel agent is moving to the network address specified by that component. The PlanTravel agent ignores the move request by the GetFlightReservation component since the PlanTravel agent received the message after the move request message sent by the GetRentalCarReservation component. Both components terminate gracefully and the PlanTravel agent moves to the location specified.

Console
File Edit Workspace Programs Window Help
Output

```
Component GatherTravelInformation is alive.

InterfaceUser agent is alive on address ENNT30AD.

Agent PlanTravel is alive on address ennt30ad.

GatherTravelInformation component has received message with performative travelInfo.

Component GatherTravelInformation is starting the bookFlight and bookRentalCar conversations.

GetFlightReservation component is alive.

GetFlightReservation component is in StartState
GetFlightReservation component is in restoreState
GetRentalCarReservation component is alive.

GetRentalCarReservation component is in StartState
GetRentalCarReservation component is in restoreState
GetRentalCarReservation component is in moveNeededState
GetRentalCarReservation component is in moveCalledState
GetRentalCarReservation component is requesting a move to address EN2K31EJ
PlanTravel agent has received a request to move from GetRentalCarReservationto address EN2K31EJ
GetRentalCarReservation component is in waitForResultState
Component GetRentalCarReservation is dying...

GetFlightReservation component is in moveNeededState
GetFlightReservation component is in moveCalledState
GetFlightReservation component is requesting a move to address EN2K31EK
GetFlightReservation component is in waitForResultState
GetFlightReservation component is in moveReceivedState
Component GetFlightReservation is dying...

Plan Travel agent is moving to EN2K31EJ.
```

Figure 74. TPS System Output Showing Agents Starting the Travel Planning Process

Figure 75 shows the output of the PlanTravel agent at the first address. The CarRental agent and the AcceptRentalCarReservations component were already running at this address before the PlanTravel agent arrived. Upon arrival, the PlanTravel agent restarted the GetFlightReservation and GetRentalCarReservation components with the state information for each component that was saved at the previous address. The GetRentalCarReservation component determines that the agent did indeed move to the location it specified, so it begins the bookRentalCar conversation with the AcceptRentalCarReservations component. The AcceptRentalCarReservations component receives the rental car information from the GetRentalCarReservation component and begins to make a reservation. On the other hand, the GetFlightReservation component determines that it is still not at an address where a Airline agent is located so it sends a message to the PlanTravel agent with a new destination. The PlanTravel agent receives the message, automatically approves the move, sends a terminate message to the

GetFlightReservation component and sends a move required message to the GetRentalCarReservation component. Both components shutdown and the PlanTravel agent attempts to move to the location specified by the GetFlightReservation component. The GetRentalCarReservation component does not have to tell the AcceptRentalCarReservations component where the agent is moving to because the Carolina agent platform keeps track of where agents move and forwards the messages to the new address. The PlanTravel agent moves before the AcceptRentalCarReservations component has sent the reservation to the GetRentalCarReservation component.

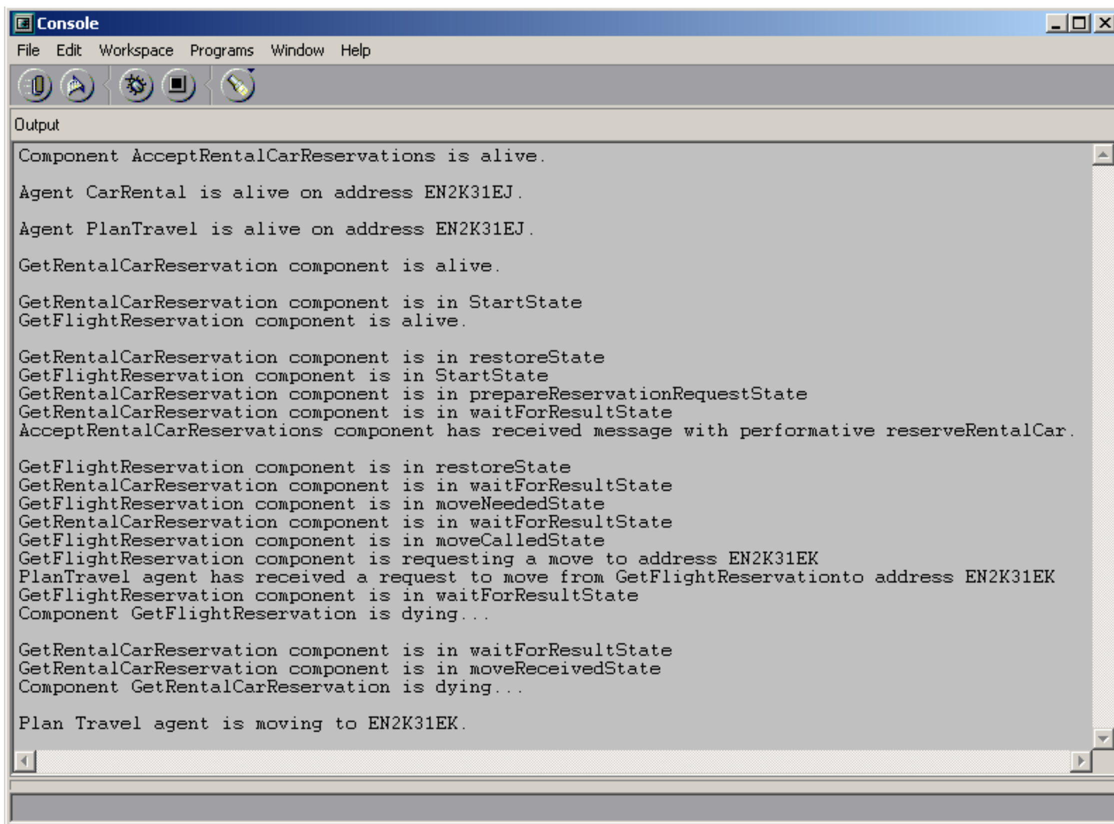


Figure 75. PlanTravel Agent at First Address

Figure 76 shows the output of the TPS system at the second and final address. The Airline agent and the AcceptFlightReservations component were already running at this address before the PlanTravel agent arrived. Upon arrival, the PlanTravel agent restarted the GetFlightReservation and GetRentalCarReservation components with the state information for each component that was saved at the

previous address. The GetFlightReservation component determines that the agent did indeed move to the location it specified, so it begins the bookFlight conversation with the AcceptFlightReservations component. The AcceptFlightReservations component receives the flight information from the GetFlightReservation component and begins to make a reservation. The GetRentalCarReservation component determines that it has moved to a new address and that it was waiting for the reservation from the AcceptRentalCarReservations component so it enters a wait for results state. The rental car reservation message arrives and the GetRentalCarReservation component forwards the result to the GatherTravelInformation component on the users machine. It then terminates because it has fulfilled its goals. The GetFlightReservation component receives the flight reservation from the AcceptFlightReservations component and forwards the result to the GatherTravelInformation component on the users machine. The GetFlightReservation component also terminates because its goals have been fulfilled as well. However, the PlanTravel remains alive because it is a reactive agent.

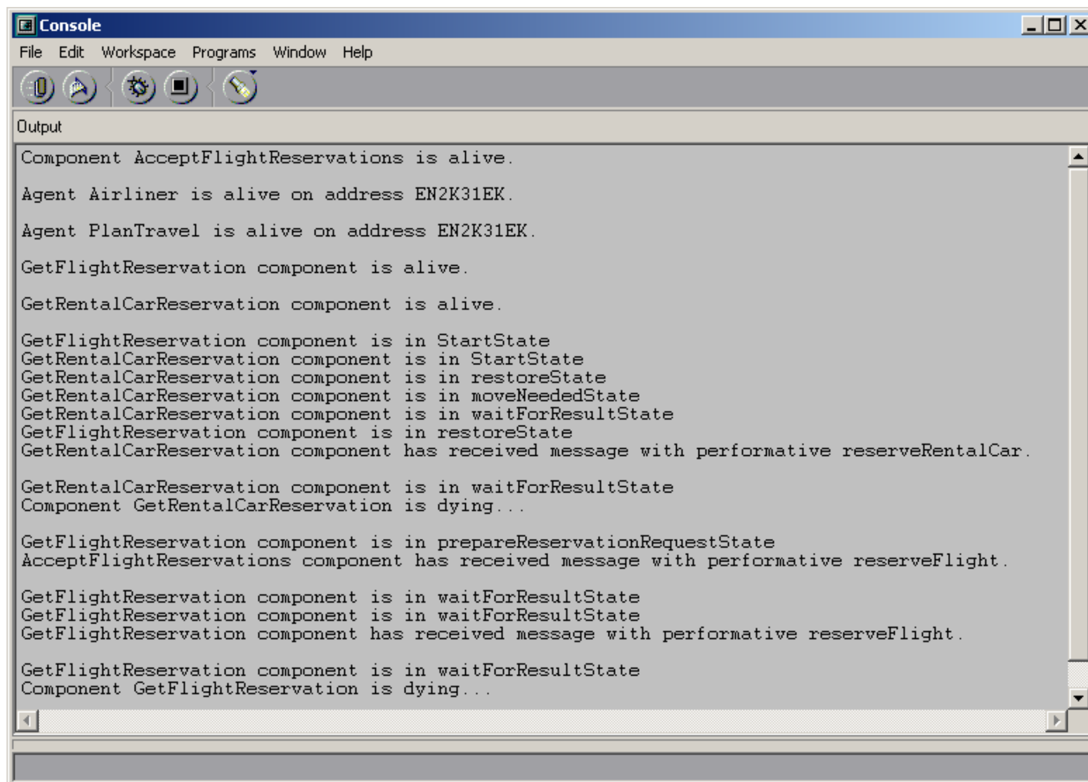


Figure 76. PlanTravel Agent at Second and Final Address

Figure 77 shows the final results from the TPS system. The GatherTravelInformation component collected the result messages from the GetFlightReservation and GetRentalCarReservation components and displayed them to the user.

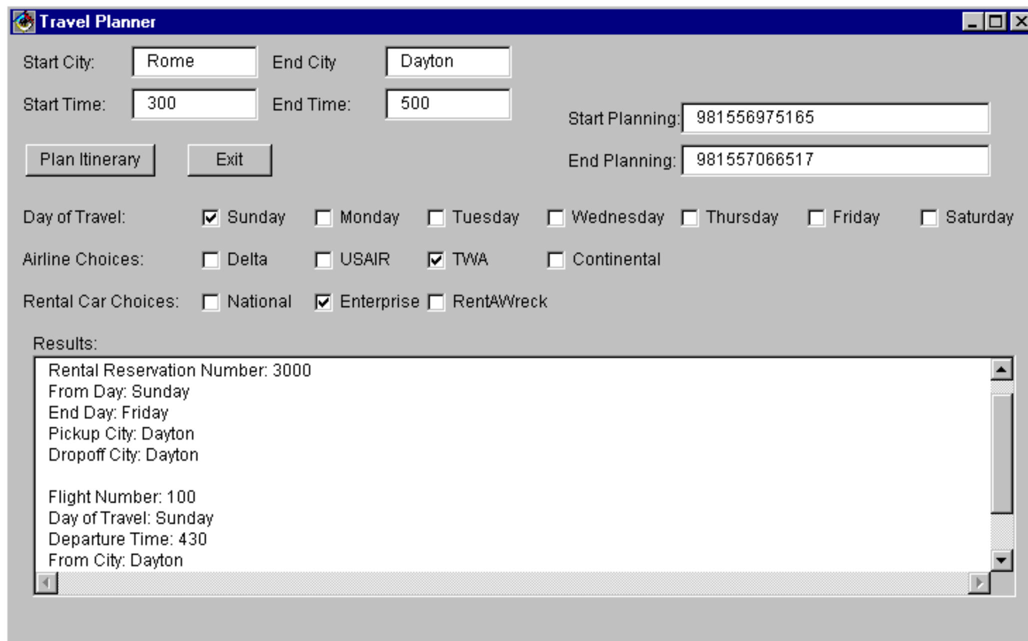


Figure 77. Final Results from TPS System

4.5 Summary

This chapter demonstrated that the solution presented in Chapter III, to the problem described in Chapter I, was feasible. An example system was analyzed and designed using the MaSE methodology with the additional functionality defined in Chapters II and III. Once the mobile design models were created in MaSE by the semi-automated transformations, the models were converted into software code for execution on the Carolina mobile agent platform. Only the software code for the conversations was automatically generated by agentTool for the Carolina environment. The travel planning system (TPS) was executed in the Carolina environment successfully, thus demonstrating the feasibility of the solution presented in this thesis. Chapter V summarizes the conclusions from previous chapters and also discusses future research for extending the work begun in this thesis.

V. Conclusions and Future Work

Incorporating mobility into the MaSE methodology turned out to be more intensive than was originally thought, especially the many enhancements that were required in the design phase of the methodology in order to support mobility. This chapter summarizes the research that was accomplished through this thesis effort and describes future research that could augment or broaden this research.

5.1 Conclusions

Additions to the MaSE methodology as described in previous chapters did indeed satisfy the end goal of this research. Analyzing a variety of options for incorporating mobility into the analysis and design phases of MaSE provided great insight into the requirements for agent mobility even though only one option was selected for each phase. Once the requirements were incorporated into the methodology, defining the analysis-to-design and design-to-design transformations was relatively straightforward.

Mobility was added to the MaSE analysis phase with minimal impact. Only the addition of a second predefined move activity was required in the Refining Roles step.

Before mobility could be specified in the design phase, an agent component was required in the new MaSE agent architecture to incorporate the behavior of the components that compose an agent class. The agent component controls the creation all components and handles the initial messages to start conversations or transient components within agent classes. It also ensures the continuous operation of a reactive agent, while also providing fault tolerance in case of persistent component failures. Transformations were defined to build an agent component for each agent class in the Assembling Agent Classes design step. These transformations build upon the analysis-to-design model transformations defined by Sparkman [18] and are considered part of that process.

Mobility was specified within all components in a mobile agent class. The agent component handles most of the moving responsibilities for the agent while the mobile components are responsible for determining the destination for agent movement. Non-mobile components are required to receive move-

required messages, save their current execution state and terminate. Automated transformations, for the most part, converted the generic design models into mobile design models, which included the agent component.

The mobile design models were then converted to Carolina agent platform software code but those models could have been converted into software code for any of the mobile agent platforms discussed in the background section of Chapter I or the mobile agent platforms section of Appendix A. The execution of the code within Carolina demonstrated the feasibility of this research.

Adding mobility to an agent-oriented software engineering methodology increases the power of that methodology to solve problems. This research provides software system designers that use MaSE an invaluable capability for analyzing and designing multiagent systems that take advantage of mobility. Also, the almost fully automated transformation process defined in this thesis allows for the addition of the basic underlying functionality for mobility, while also allowing for increased specification of the decision process to move, to a multiagent system with almost minimal effort.

5.2 Future Research

There are many possible ways for expanding and verifying the work done in this thesis but the following discussion will be limited to four different areas. Incorporating the other two dynamic agent properties namely cloning and instantiation, is the main area for expanding this thesis. Another area would be to create transformations that add security to the design models. A third area would be to define formal proofs of the transformations contained within this thesis to verify the validity and correctness of the transformations. And a final area, software code for the mobile design models (existing conversation code generation only needs to be modified) can be automatically generated.

5.2.1 Specifying Cloning and Instantiation

Cloning and instantiation add to the power of a dynamic agent system. Mobility is just a special case of cloning. Full cloning capability would bring the advantages of parallel computing and optimal

distribution of tasks to a multiagent system in order to solve performance bottlenecks. The decision to clone should probably be the responsibility of the components, as is the case with mobility, with the agent component carrying out the details.

Instantiation of other agents that have the capabilities to perform certain tasks ensures that agents within multiagent systems can fulfill all of their goals through agent collaboration. Instantiation might differ from mobility in the respect that the agent component could have the both the responsibility to decide on instantiating another agent and the responsibility for starting the process of instantiation.

5.2.2 Adding Security to Design Model

Security is a major concern in the area of computing with multiagent systems and mobile agents being no exception. Defining transformations that add security to a generic design model that may or may not contain mobility would be an important addition to the overall MaSE methodology. Since the capability to design a system containing mobile agents has been added to MaSE and there are more security concerns with using mobile agents than a multiagent system in general, the need for designing security into the MaSE design models has greatly increased.

5.2.3 Formal Proof of Transformations

Even though examples were used in this thesis to show that the transformations worked as designed, they do not cover every possible consideration. Formal proofs are required to make sure that the transformations are correct. In order to formally define the transformations, semantics of the MaSE models used within [18] would have to be formally specified.

5.2.4 Automatic Code Generation

Generating code automatically, that executes as designed, has been the goal of much research in the field of computer science. agentTool has automatic code generation for the conversations between agents using the Carolina agent platform (used in the demonstration system in Chapter IV), so modifying

and extending that to include the code for the components and code for other mobile agent systems would be a straightforward attainable goal.

5.3 Summary

Mobile agent technology is a relatively new and exciting field in the area of artificial intelligence and software engineering. This thesis starts to bridge the gap between agent-oriented software engineering methodologies and mobile agent systems. Merging these two areas provides more power to solve the complex problems facing the Air Force and Department of Defense in this new era of increasing mobile, distributed computing.

A. Background

In order to solve the problem of integrating dynamic agent concepts into a multiagent design methodology a good understanding of dynamic agents, multiagent systems, Agent-Oriented Software Engineering (AOSE) methodologies and Agent Platforms (AP) that support dynamic agent operation is required. Section A.1 is devoted to discussing dynamic agents while the advantages for using these specialized agents is covered in Section A.2. Next, current AOSE methodologies are described in Section A.3 and finally, an overview of the properties and capabilities of Mobile Agent Platforms (AP) is covered in Section A.4.

A.1 Dynamic Agents

Building high quality software for real world applications is one of the most difficult construction tasks facing humans today [8]. It is the number and complexity of real world components and the relationships between those components that makes modeling the real world so difficult. Real world problems are also distributed, dynamic and heterogeneous. Many different software engineering approaches have been tried to tackle the task of modeling complex, distributed systems including object-oriented methodologies.

This methodology models the real world by using an abstraction called an object. Objects represent real world entities such as cars, airplanes or rooms. They encapsulate attributes of those objects such as size, weight, dimensions, location, capacity and speed limits. Objects also encapsulate methods that can access or change attributes describing that object. These methods can also perform some action on an object. Some method examples are: `set_location(3,4)`, `get_location()`, `get_speed()`, `set_speed(21)`, `fly()`, `land()`, `etc`.

Interactions between objects consist of one object calling a method from another object. These interactions are precisely defined during the design phase and are static for the life of the system. This means that no object will call a method from another object if that call was not modeled in the original design of the system. Also, the objects are placed into a structured hierarchy or organization during the

design phase as well. This organization is again static and will never change during the lifetime of the system.

It is precisely these two aspects of OO that are inadequate for developing complex distributed systems [8]:

(i) Interactions between computational entities are all defined a priori and do not change during execution of the system

(ii) Organizational structures are defined a priori and do not change during execution of the system

Interactions between entities may need to occur that were never planned and different organizational structures are needed at different times to ensure that system goals are accomplished. With the explosion of the Internet and client-server architecture, distributed computing environments have begun to dominate the computer world. Approaches to developing applications to operate in these environments are critically needed.

Researchers in the discipline of artificial intelligence have started to develop approaches to solve problems in these complex, distributed environments. Current and past work in artificial intelligence has been focused mainly on developing software systems that appear intelligent to humans using agents. An *agent* is a software system that displays the following traits or properties [24]:

- Autonomous – not controlled directly by humans or other agents
- Cooperative – agents communicate amongst themselves to help achieve goals
- Perceptive – agents perceive, react and can make changes to their environment.
- Pro-active – agents are not passive entities like objects. They exhibit goal-directed behavior.

Past work in artificial intelligence consisted mainly of a single agent [19]. A single agents' capacity to solve problems is limited by its knowledge, perspective and computing resources. Real world problems are generally too large, complex and unpredictable to be handled by a single agent. Some AI researchers have realized the limitations of a single agent and have focused work on multiagent systems.

These systems can be executed on one machine or can be distributed across multiple networked machines. Distributed multiagent systems bring the power of modularity in the form of flexible agent-based organizations to handle the complexity and size of real world problems. Multiagent systems can also provide efficient solutions to real world problems where resources or expertise is distributed.

The agents that make up these systems can be either static or dynamic. Dynamic agents are agents that possess the following properties:

- Cloning – the ability of an agent to create another instance of itself
- Instantiation – the ability of an agent to create instances of another type or class of agent other than itself
- Mobility – the ability of an agent to move from machine to machine in a network

These three agent properties, or traits, have been the focus of new research in the DAI arena with the property of mobility receiving the most attention.

Two specifications for these properties are found in current literature [6][17]. In the Foundation for Intelligent Physical Agents (FIPA) specification, cloning, instantiation and mobility are all grouped under mobility. Simple and full protocols have been defined for each property. An agent using a simple protocol relies on the agent platforms involved to perform most of the work while an agent using a full protocol will split the work between itself and the agent platforms. Shehory and others agree with the FIPA specifications but consider mobility to be a special instance of cloning and so group mobility under cloning. Their definition of mobility however, follows the FIPA simple mobility protocol exactly. Figure 78 shows the FIPA simple mobility protocol.

In this protocol an agent sends a request to move to the agent platform on which it is currently executing. The home platform terminates the agent and sends it to the destination platform. Finally, the agent is restarted on the destination platform. The agent by using this protocol is in effect delegating the mobility operation to the agent platform.

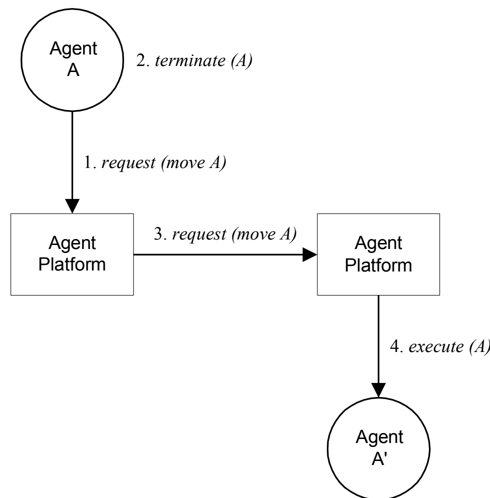


Figure 78. FIPA Simple Mobility Protocol [6]

A.2 Dynamic Agent Advantages

Dynamic agent systems have shown promise in being able to solve certain network related problems such as finding services and information on the World Wide Web (WWW). These systems have shown an advantage over other client-server interaction solutions, such as remote procedure calls (RPC), messaging and sockets, in terms of different application and network factors. Four advantages, if done properly, for using dynamic agents are [2][7][11]:

1) They can reduce communication and bandwidth costs
2) They can be hardware and operating system independent depending on the mobile agent platform they execute on
3) They can be fault tolerant
4) They can maintain optimal configuration

First, dynamic agents have an advantage in networks where communication costs increase as bandwidth decreases. Examples of this type of network include Internet connections using cellular phone networks and satellite-based networks. Current network communications rely heavily on RPC, which is a synchronous protocol. RPC is the protocol used when one network machine calls a procedure on another machine. The calling machine sends data over to the receiving machine and then waits for a response before sending any more data. All the messages passed in the network are either a request for service or an

acknowledgment for a received message. This approach requires a continuous connection between two machines, an established protocol for the message exchanges and multiple interactions in order to complete a task. Thus, a great deal of network bandwidth can be tied up because of the need for this type of connection and frequency of communication. Dynamic agents avoid this communications overhead by being able to migrate to another machine and perform data gathering on that machine. Once the data has been processed then the agent travels back to the original machine. Communication traffic in this case was reduced to just two agent moves and bandwidth was saved because only relevant information was sent across the network.

Second, current distributed systems including the Internet are not only heterogeneous in terms of the hardware that they run on but also the software that defines them. Since most dynamic agent platforms are written in Java or some other device independent software language they are a natural fit for widely dispersed heterogeneous execution environments. Most platforms rely only on their particular execution environment. Because of this independence these dynamic agent platforms are computer, and transport-layer independent.

Third, system crashing, loss of network connectivity and periodic reboots are just some of the problems that occur regularly in networks. Dynamic agents and systems are able to perceive changes to their environment and then have the functionality take an appropriate action. In the case of a system crash most dynamic agent systems have persistent storage of agents so that recovery of those agents is straightforward. If the network is down either the agent or system will keep trying to migrate until the connection is restored. Finally, if a machine needs to be rebooted, the agents would be notified by the agent system. The agents then have the option to either migrate to other available machines in the network or be stored until the machine is operating again.

Fourth, processor or agent overloading can occur in a multiagent system. When processor overloading occurs, dynamic agents can migrate to a machine in the network where the processor is not overloaded. And in the case of multiple dynamic agents, this functionality allows them to distribute themselves around a network in an optimal configuration in order to complete tasks. In the case of agent

overloading the dynamic agent either makes a clone to perform some of the tasks or, if the dynamic agent does not have the capability to perform, it can instantiate an agent that can perform those tasks.

A.2.1 Summary

Agents are a natural abstraction of objects from the object-oriented methodology. The properties of dynamic agents (mobility, cloning and instantiation) can be thought of as extensions to the general definition of an agent. As shown above, dynamic agents posses tremendous network functionality and could replace current network protocols such as RPC and messaging. In fact, if all the functionality provided by a dynamic agent system is taken together, there is no single alternative that can provide that same level of functionality [2]. However, there are three main reasons why dynamic agent technology has not replaced those protocols. The first deals with the lack of even one network function that can only be accomplished by using dynamic agents. Everything that a dynamic agent system can do can be done using other network protocols [2]. The second reason is that RPC and messaging have been in use for many years and are successful. And finally, dynamic agent technology is still in its infancy. Many issues still need to be worked out including security, standards, and interoperability. There are many dynamic agent systems (mostly focusing on the mobility aspect) already in industry but there is no interoperability to date. FIPA and the Object Modeling Group have begun to develop standards but they have not been widely accepted.

A.3 Agent-Oriented Software Engineering (AOSE) Methodologies

AOSE differs from object-oriented in terms of the agent-oriented abstractions used to model a system. These abstractions include agent-oriented decomposition, characterizing complex systems by subsystems, interactions, and organizational relationships to include building in mechanisms for agents to flexibly form, maintain, and disband these organizations. Some of the current AOSE methodologies include Agent-Oriented Analysis and Design (GAIA) [25], Multi Agent Scenario-Based (MASB) [13] and Multiagent Systems Engineering (MaSE) [23]. Each of these methodologies is discussed further in the following sections.

A.3.1 Agent-Oriented Analysis and Design (GAIA)

The GAIA methodology [25] is based on the view of a system as a computational organization consisting of various interacting roles. It has an analysis phase that is comprised of a roles model and an interaction model. GAIA also has a design phase that is comprised of an agent Model, a services model and an acquaintance model. The main models of the GAIA methodology are shown in Figure 79. The arrows represent the input to each model.

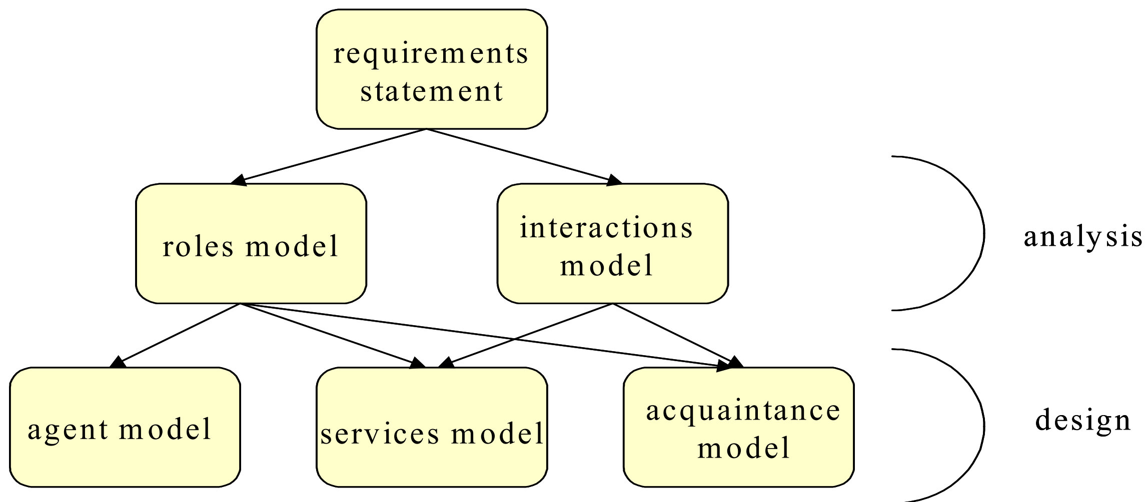


Figure 79. The GAIA Methodology [25]

The roles model schema identifies the key roles in the system where a role is an abstract definition of an agent's expected function. Each role has three types of attributes: Permissions/Rights, Responsibilities and Protocols. Permissions are used to identify resources (information or knowledge that an agent has) and the limits to which those resources can be used to carry out a role. Responsibilities define the functionality of a role. In GAIA there are two categories of responsibilities: liveness and safety. Liveness responsibilities imply that something will be done and refer to the actions that define a role. Safety responsibilities ensure that undesirable conditions do not arise as the liveness responsibilities are carried out. Finally, protocols are formally defined patterns of interactions that occur in the system between the various roles. These protocols are represented in the interaction model. An example roles model schema for a CoffeeFiller role is shown in Figure 80 while the Fill Protocol Definition for the interaction model between the CoffeeFiller and CoffeeMachine roles is shown in Figure 81.

ROLE SCHEMA: CoffeeFiller

DESCRIPTION:
This role involves ensuring the coffee is kept filled and informing the workers when fresh coffee has been brewed.

PROTOCOLS:
Fill, InformWorkers, CheckStock, AwaitEmpty

PERMISSIONS:

reads	supplied coffeeMaker	*//name of coffee maker*
	coffeeStatus	*//full or empty*
changes	coffeeStock	*//stock level of coffee*

RESPONSIBILITIES

LIVELINESS:
CoffeeFiller = $(\text{Fill.InformWorkers.CheckStock.AwaitEmpty})^{\omega}$

SAFETY:
- coffeeStock ≥ 0

Figure 80. Example Roles Model Schema [25]

Fill

CoffeeFiller | CoffeeMachine

Fill coffee machine

Supplied coffeeMaker

coffeeStock

Figure 81. Fill Protocol Definition [25]

The design phase of GAIA is concerned with transforming the analysis model into a sufficiently low level of abstraction so that traditional design techniques (including object-oriented) may be applied. Three models are generated in the design phase as shown in Figure 79. The purpose of the agent model is to identify the different *agent types* in the system and the *agent instances* that will represent the agent types at run-time. The services model identifies the *services* and main properties of those services that are associated with each agent role. Services are single, coherent blocks of activity (functions) that agents perform.

A.3.2 Multi Agent Scenario-Based (MASB)

Another analysis and design method for the development of multiagent systems is MASB [13]. MASB views multiagent systems as systems composed of software agents (SA's) playing one or more different roles in predetermined scenarios (much like protocols in GAIA). The analysis phase consists of the following 5 steps: Scenario description, Role functional description, Conceptual data modeling, Static and dynamic descriptions of the world and System-user interaction modeling. The design phase consists of

the following 5 steps: MAS architecture and scenario characterization, Object modeling, Agent modeling, Conversation modeling, and System design overall validation.

Several modeling techniques with user collaboration are used in the analysis phase of MASB to describe scenarios, agent's roles and their main knowledge structures. A scenario (or script) is a narrative, a graphical or a visual description of what people do when they perform tasks. Each scenario is specified as a set of roles that are played by agents involved in that scenario. Behavioral diagrams [14] are then used to precisely describe the scenarios. These diagrams outline the scenario in terms of agent roles and activities, local information or knowledge stores and their contributions to activities, and interactions with other agents playing specific roles. Then, the local information or knowledge stores described in the behavior diagrams are analyzed further in terms of attributes. A conceptual data structure is then created that features the main elements that will be included in the SA's local database. However, the conceptual data structure only specifies the static properties of an agent's fact base so object life-cycle diagrams [12] are used to reflect the changes that could occur in the attributes. Next, the world in which the agents act is described. This step consists mainly of identifying and structuring artifacts that the SA's create or manipulate. Finally, the interactions that take place between the users and the SAs are modeled using forms generated by hand or computer.

The objective of the design phase of MASB is to transform all the models from the analysis into formal specifications that are used as inputs to the SMAUL tool, which generates the actual code for the agents. First, scenarios from the first analysis step, the roles that make up those scenarios and which SAs will support those roles are identified. Behavior diagrams from the second analysis step are also used to assign the appropriate behaviors to the roles that were selected above. Second, structures (class hierarchies, attributes, procedural attachments, etc.) and behaviors of objects are specified from the models obtained during the fourth analysis step. Third, designers formally specify the knowledge structures characterizing SAs to include: beliefs, decision space, action space and reasoning space. Belief structures for agents are derived from the data conceptual structure created during the third analysis step. A decision space and action space are then created for each role that an agent can play using the scenario descriptions from the analysis phase as a starting point. Fourth, conversations are modeled and most of the code for the MAS is

generated. Finally, the system is validated by manually triggering the behaviors of the various SAs in the system.

A.3.3 Multiagent Systems Engineering (MaSE)

The Air Force Institute of Technology (AFIT) Agent Research Group has been focused on developing and maturing an AOSE methodology. The MaSE methodology [23] has been designed to cover the entire life cycle of developing and implementing a multiagent system. An overview of MaSE is shown in Figure 82, with the two phases, analysis and design, shown on the right hand side next to the seven steps. The boxes represent the different models used in those steps.

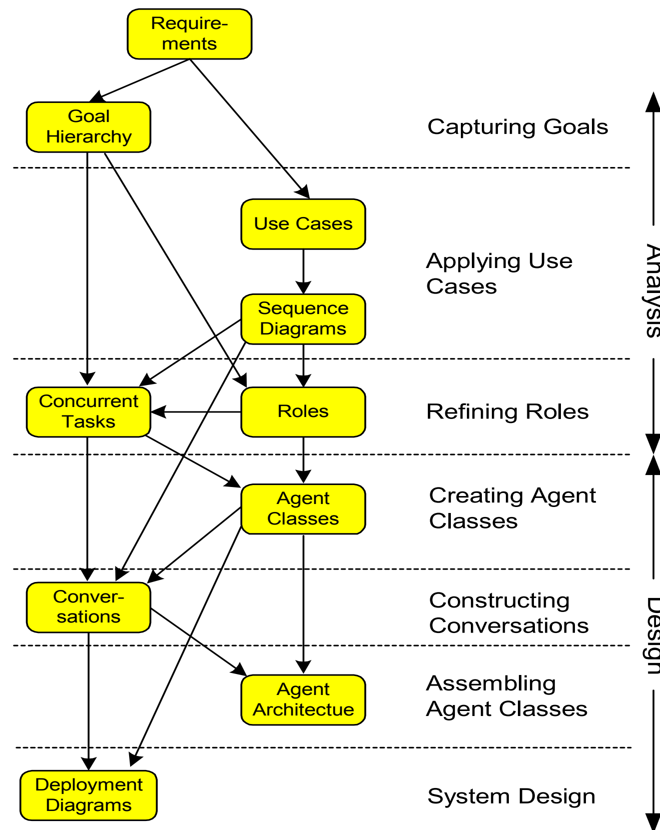


Figure 82. The MaSE Methodology [23]

A major difference between OO and MaSE analysis deals with how the problem is initially broken down. In the OO approach, the analyst evaluates an initial system specification or context in terms of the separate "objects" that exist within the system and creates an object diagram showing those objects.

However, in step 1 of the MaSE methodology, the overall goals of the system are identified from the specification and are structured into a goal hierarchy diagram as shown in Figure 83. Goals are used instead of identifying functions of the system because the goals are more stable and less likely to change during the development cycle.

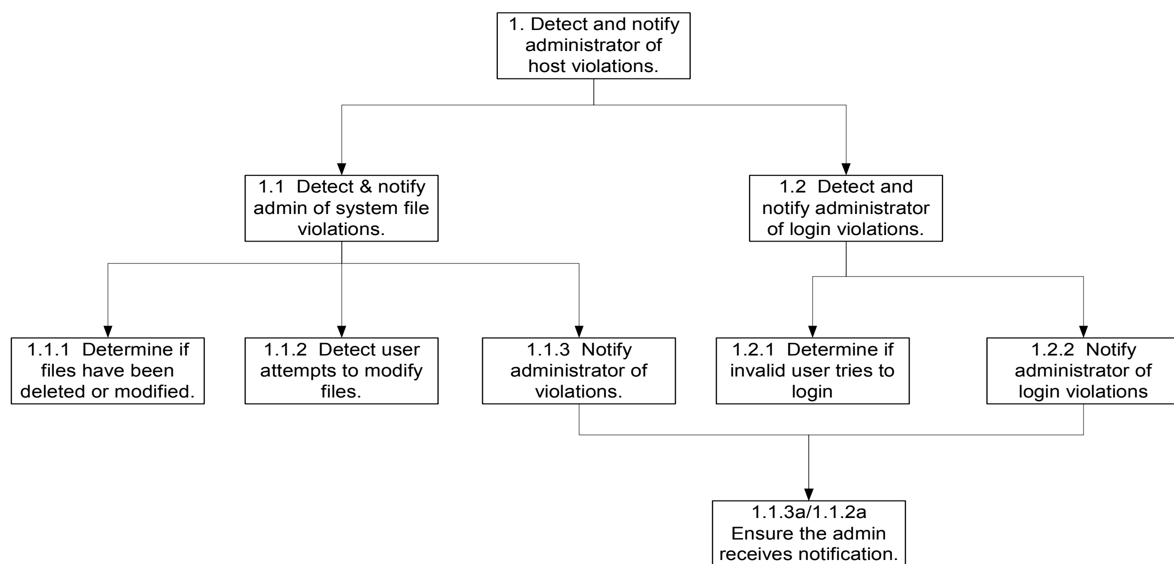


Figure 83. Goal Hierarchy Diagram [5]

Step 2 of MaSE then deals with creating use cases that help identify possible roles that will exist in the system and also outline initial communication paths between those roles. A *role* is an abstract description of an expected function that agents in the system will be required to perform and is made up of system goals that the role is expected to satisfy [5]. Roles can be thought of as particular positions within a company structure such as the President or Chief Information Officer. And like those positions in the company, an agent playing that role has certain responsibilities or goals.

Roles are partly defined by the sequence diagrams that are taken from use cases gathered either from the initial system context or from user stories. *Use cases* are sequences of events that define system behavior. There are two types of use cases needed to properly describe system behavior. A sequence of events between one or more roles is what defines a sequence diagram. Events correspond to the information passing between the individual roles in the use cases. One sequence diagram generally corresponds to one use case. An example of a MaSE sequence diagram is shown in Figure 84.

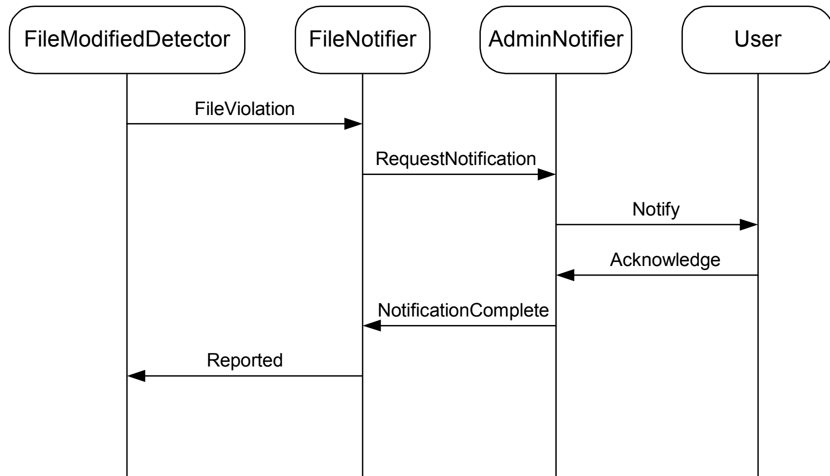


Figure 84. Sequence Diagram [5]

Once the goals have been captured and the system behavior has been modeled by use cases and sequence diagrams, the last analysis step in MaSE, Transforming Goals to Roles, can be accomplished. In this step the structured goals and sequence diagrams are transformed into roles and their associated tasks. Roles and tasks are the basic building blocks for developing multiagent systems. Every system goal will be accomplished if the goals are associated with roles and an agent class plays each role. There is generally a one-to-one mapping of goals to roles but in some cases goals are combined and placed into a single role. An example of a mapping of goals to roles in MaSE is shown in Figure 85.

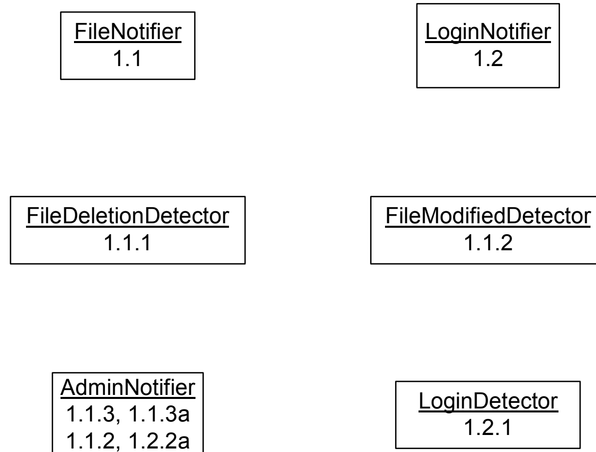


Figure 85. Goals Mapped to Roles [5]

Roles must exhibit certain behaviors in order to accomplish the goals that have been associated with them. These behaviors are modeled using concurrent tasks where each role may have many concurrent tasks that define its behavior. Sequence diagrams from step 2 of the methodology, Applying Use Cases, can be used to help initially define the tasks. *Concurrent tasks* are tasks that operate in their own thread of control and are graphically specified using *Concurrent Task Diagrams* that are built using finite state automata [4]. Actions in the states of those diagrams contain the functionality that the role is to perform. A transition between the states represents either internal coordination of tasks within a role or interactions between tasks of different roles. Figure 86 shows an example Concurrent task diagram and Figure 87 represents the different roles in a multiagent system along with their associated tasks.

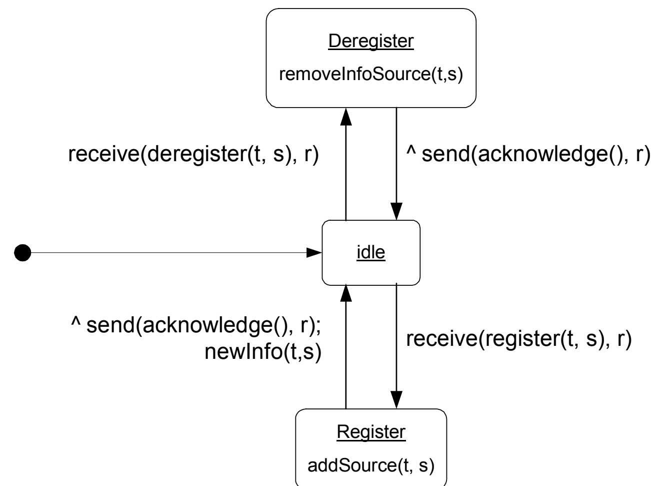


Figure 86. Registration Task [4]

Concurrent tasks have the goal in MaSE of defining the behavior of agents by tying together the internal reasoning processes, interactions with other internal processes and external interactions with other agents [3]. Tasks are categorized by their *life span* and *responsiveness*. Task life spans are either persistent or transient. Persistent tasks always have a *null* transition from the start state to the first state. These tasks are started when the agent is created and run until either the task or agent terminates. Transient tasks, however, always have a trigger event on the transition from the start state. These tasks are not started upon agent creation but only when the agent receives its trigger event. Transient tasks make it possible to have multiple concurrently executing tasks of the same type.

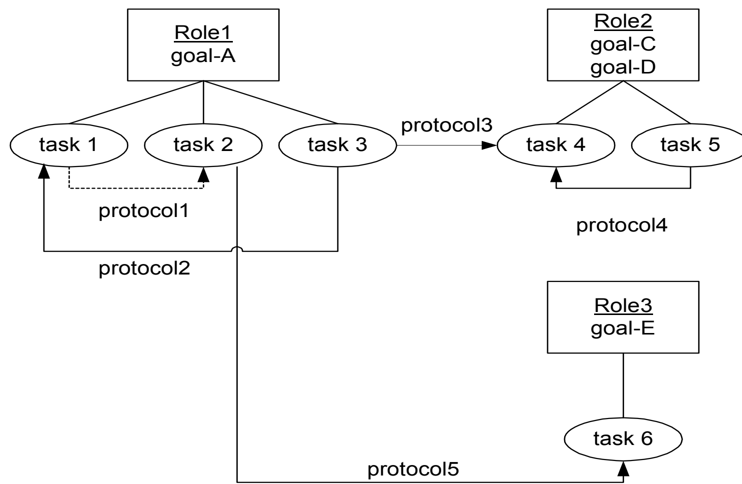


Figure 87. MaSE Role Model [5]

There are three types of task responsiveness: reactive, proactive or heterogeneous. Reactive tasks can be either persistent or transient. A *persistent reactive task* has a null transition from the start state to an idle state, where it remains until it receives a triggering event from the agent. On the other hand, a *transient reactive task* must receive a triggering event from the agent before it can begin processing. A *proactive task* continuously generates requests for other agent or tasks, is always persistent and does not contain any idle states. And finally, a *heterogeneous task* is a combination of a proactive and reactive task, but is always persistent, never starts in an idle state and must generate at least one request for another task or agent before entering an idle state.

After the analysis phase is complete, there are four steps in the design phase [23]. The first of these four steps is Creating Agent Classes. An *Agent Class* is a model for the types of agents that will exist in the system and is similar to an object class in object-oriented except that an agent class is defined by the roles it plays and not by attributes and methods. Agents are then just instantiations of these agent classes as is the case for objects in OO and can dynamically play one or more roles during execution of the system.

Along with assembling the agent classes in this step, conversations between those classes are identified. Conversations are the mechanism by which agents communicate within MaSE. A conversation is a coordination protocol defined between two agents. Agent classes and conversations are graphically

displayed in an Agent Class Diagram as shown in Figure 88. Agent class names and the roles that the agents play are contained in the boxes while the lines between the boxes represent the conversations. Lines point from the agent that initiates the conversation to the agent that responds.

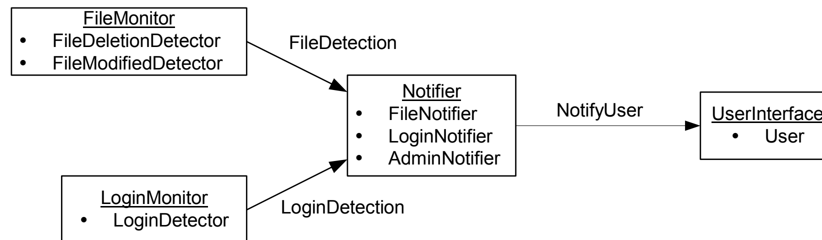


Figure 88. Agent Class Diagram [23]

Once the conversations have been identified, the details of the conversations are defined in the Constructing Conversations design step. Conversation details are derived from the concurrent tasks. Each event or message in the tasks and sequence diagrams must be translated into a send and receive transition on two complimentary communication class diagrams. *Communication Class Diagrams* define the states for every conversation and are modeled as finite state automaton. As stated above each conversation has an initiator agent and a responder agent. Conversations should be small and only support one goal if possible.

Next in the Design phase is the Assembling Agents step. This step is concerned with creating the internals of the agent classes and is accomplished by defining the agent architecture and components that make up that architecture. Examples of agent architectures are: Belief-Desire-Intention (BDI), reactive, planning and knowledge. These architectures are essentially composed of components and connectors.

In MaSE, a *component* is a storehouse of an agent's state and execution and is graphically represented much like a class is in an OO diagram with attributes and methods. Components are assembled to define an agent class by one of three different methods:

- Using a pre-defined architecture
- Combining one or more pre-defined architectures to form a new architecture

- Define an architecture and corresponding components from scratch

The first two methods allow for modification of the components in the architectures by adding, deleting or modifying the attributes and methods of those components.

As stated above the interactions between components are connectors. There are currently two types of connectors: inner agent and outer agent. An *Inner Agent Connector* connects two components with the same agent class, giving one or both of the components access to the attributes and methods of the other component. An *Outer Agent Connector* connects a component from one agent class to an external entity, which could be a component from another agent class or an entity in the agent's environment. Both inner agent and outer agent connectors are represented using one-way or two-way arrows, but the arrows for the outer agent connectors are thicker and dashed. The directions of the arrows denote which entity has access to the other entity. An example component diagram in MaSE with inner and outer agent connectors is shown in Figure 89.

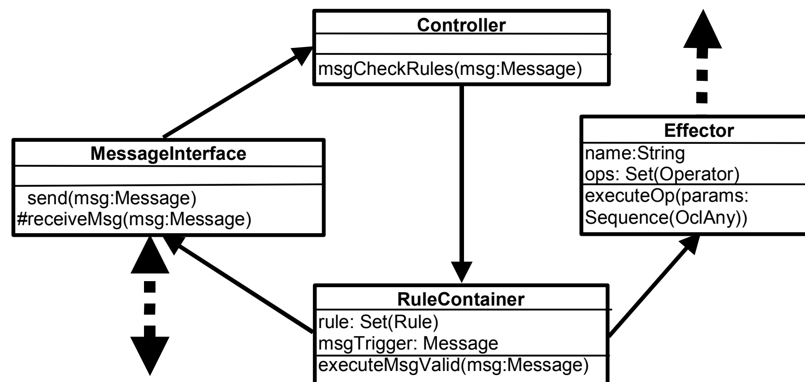


Figure 89. MaSE Component Diagram [15]

System deployment is the final step in the design phase of MaSE and is the step where the previously defined agent classes are instantiated. Numbers, types and locations of agents within the system are shown in a deployment diagram. Boxes on the diagram represent the different agents and the lines between them represent conversations between the agents. A dashed line around a group of boxes means that all those agents reside on the same machine. A deployment diagram is primarily used to identify the

machines that the agents will be using as well as being used to tune the system to maximize processing power and network bandwidth. An example deployment diagram is shown in Figure 90.

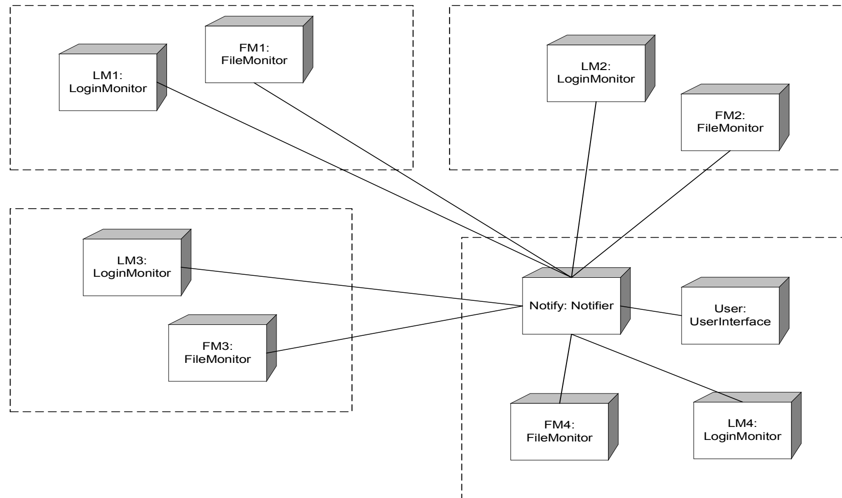


Figure 90. Deployment Diagram [23]

A.3.4 Summary

As shown above the GAIA, MASB and MaSE methodologies all cover the entire lifecycle of multiagent systems design. All three methodologies are similar because they are based on the view of a multiagent system being a computational organization consisting of various interacting roles. In each methodology an agent can play one or many roles at any given time. The agents in these methodologies can also be involved in more than one conversation at a time.

These methodologies differ mainly in terms of the amount of detail they provide and functionality not found in the others. MaSE and MASB provide much more detail for defining conversations than GAIA. MASB is the only methodology that emphasizes human/agent interactions. Only MASB and MaSE provide models for building the internal knowledge structures for agents and MaSE is the only methodology that specifies the locations of the agents in the final system. One major shortfall that exists in all three of the methodologies described above is the ability to analyze and design a multiagent system where the agents can be dynamic.

A.4 Mobile Agent Platforms

Once a multiagent system has been analyzed and designed utilizing dynamic agents then an agent platform is required to handle the execution of those agents. An *agent platform*, by general definition, is a software system that has the ability to create, name, dispatch and terminate agents [1]. The Foundation for Intelligent Physical Agents (FIPA) [6] has defined an agent management reference model, which outlines the following logical components that can be included in an agent platform:

- *Agent Platform (Agent Execution Environment)* – Environment in which the agent can exist.
- *Agent Management System* – Agent that manages the use of and access to the AP.
- *Agent Communication Channel* – Allows agents to exchange information between one another concerning services and communication messages.
- *Directory Facilitator* – A "yellow pages"-like directory service that advertises the services the agents provide within the system.
- *Agent Platform Security Manager* – Agent that maintains the security policies for the AP. An APSM is also responsible for negotiating access requests for agents with other APSMs.
- *Agent Resource Broker* – Agent that maintains and brokers software services provided by non-agents.
- *Wrapper Agent* – Agent that communicates with non-agent software, allowing agents to interact with that non-agent software.

Most current agent platforms that support mobility, including Concordia [22], Telescript/Odyssey [21], Aglets Workbench [10] and Carolina [16], provide a variation of these capabilities. Each of these mobile agent platforms is discussed in the following sections.

A.4.1 Concordia

Concordia [22] is a mobile agent system currently under development at Mitsubishi Electric Horizon Systems Laboratory. Concordia provides all of the logical components listed by FIPA. Figure 91 shows the current Concordia Server Architecture.

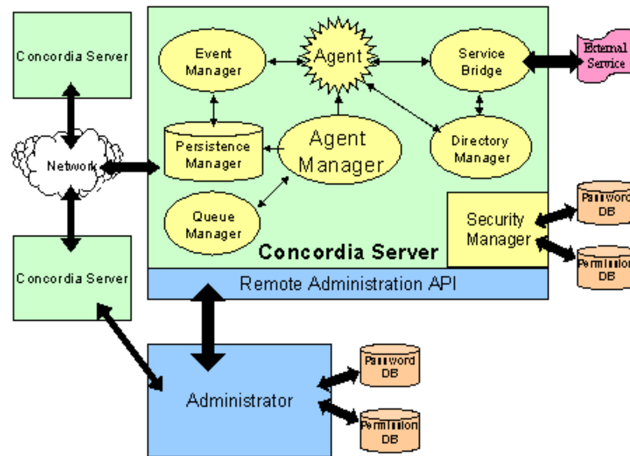


Figure 91. Concordia Architecture [22]

The Agent Manager handles the transfer of the agents to and from the Concordia server and provides the place where agents execute. The Persistence Manager and Queue Manager work in tandem with the Agent Manager to ensure reliable agent migration and survivability. The Queue Manager is used to store the state of agents before they are transferred to other Concordia servers. If other servers are down, the Queue Manager will periodically attempt to resend agents until the node is back online. The Persistence Manager is used to store the state of agents on disk so recovery from system crashes is possible.

Agents have an itinerary that controls their movement from server to server. The itinerary consists of a server address and method. Once an agent is started it travels to the first place listed in the itinerary and runs the specified method for that location. Once that method is finished the agent travels to the next location specified in the itinerary. This process continues until the itinerary is finished. The itinerary can be modified during agent execution.

Agents are created by authorized users and carry the users' identity wherever they travel. Thus, the Security Manager verifies new agents by the users who created them. The Security Manager provides protection is the following three areas: agent storage, agent transmission and server resources.

The Directory Manager provides the directory facilitator functionality for Concordia while the Event Manager provides the agent communication channel. Asynchronous distributed events and agent collaboration are the two forms of inter-agent communication within the Concordia server. Events are basically messages that are sent to agents. Agents can get on event mailing lists where events of a certain type are automatically sent to subscribing agents. Agents can also join event groups where each agent in a group receives an event sent by one of the members of that group. It is these event groups, which form the basis for agent collaboration.

A.4.2 Telescript/Odyssey

Telescript or Odyssey (Java version of Telescript) [21] was the first commercial implementation of a mobile agent system. Telescript implements all of the FIPA agent management reference model logical components except the agent resource broker and wrapper agent. This technology is focused on the concept of an electronic marketplace, which is a public network of service and goods providers and consumers that find each other and transact business electronically. This network does not exist currently but its beginnings can be seen on the Internet.

Telescript is based on the following concepts: places, agents, travel, meetings, connections, authorities and permits. Telescript models a network of computers as a collection of places. These places offer service to mobile agents and some examples from the electronic marketplace are: a directory place, a ticket place, a flower place, etc. Places are equivalent to the Agent Platform, Agent Manager and Directory Facilitator components in the FIPA model. Agents represent either a provider or a consumer and can either be stationary or can travel to other places. An agent travels to a new place by calling the `go ()` instruction. Meetings occur between agents located in the same place and are where agents conduct their business. Connections allow two agents on different machines to communicate. Connections and meetings

correspond to the Agent Communication Channel component in the FIPA model. Authorities deal with verifying the identity of agents for security reasons while permits control the capabilities of places and agents. Examples of permits for an agent would be the permission to execute certain instructions, the maximum lifetime for the agent, and the maximum amount of computation resources that the agent can use. Authorities and permits correspond to the Agent Platform Security Manager component in the FIPA model.

A.4.3 Aglets Workbench

Aglets Workbench [10] is the mobile agent architecture designed by IBM. Aglets Workbench implements only the Agent Platform, Agent Management System, Security Manager and Agent Communication Channel components of the FIPA agent management reference model. Aglet is the name for the mobile agents executing on this workbench.

Agent Platform functionality is provided by the workbench. Each workbench provides a toolkit for creating aglets and also the place where the aglets execute. A *context* is a stationary object on a workbench that acts as the agent manager for one or many aglets. There can be many contexts managing one or more aglets running on a given workbench at any given time. Once a context has been created then aglets can be created within that context. Aglets move from context to context by calling a `dispatch()` method.

Aglets communicate by message passing only. *Messages* are simply objects that are passed between aglets and can be synchronous or asynchronous. Multicasting is also supported but only within a given context. However, all messages passed between aglets must go through proxies. A *proxy* is an object that represents an aglet and provides the communication interface for the aglet, security and location transparency. Any given message is first passed to a proxy object and that proxy object sends the message to the intended aglet whether the aglet is local or on a remote host.

Contexts and proxies jointly handle security for aglets. Contexts provide security by protecting the host system and its resources from the aglets that are operating within a given context. Also, contexts protect aglets from interfering with each other within a given context. Proxies protect aglets by ensuring

that all communication is handled by message passing and not by one aglet invoking a method within another aglet.

A.4.4 Carolina

Carolina [16] is currently under development at the University of Connecticut as part of the Multi-Agent Distributed Goals Satisfaction project. AFIT, the University of Connecticut, and Wright State University are conducting this research jointly. Figure 92 shows the current Carolina server architecture.

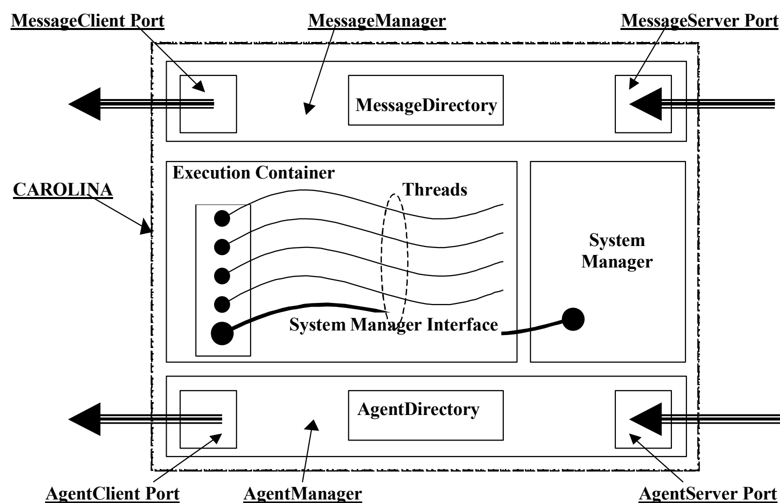


Figure 92. Carolina Architecture [16]

Three of the logical components listed by FIPA are provided by Carolina: the Agent Execution Environment, the Agent Management System and an Agent Communication Channel. An agent execution environment is common to all APs and is the place where the agent code executes. An AgentManager and AgentDirectory handle agent management in Carolina. The AgentManager handles all agents that are executed locally or arrive on the AgentServer Port. The AgentDirectory is a repository of information about all the agents that the server has seen, thus making it one of the key components for communication and system consistency. Information stored in the directory about agents includes: a unique name, agent type, creation machine address, current machine address, goal, and pointer to the messages stored for the agent.

Finally, the MessageManager and MessageDirectory make up the Agent Communication Channel in Carolina. All communication between agents running in the Carolina environment is handled through the server due to the problems of keeping track of mobile agents and being able to deliver messages in a timely manner. The decision against agent-to-agent communication constitutes a departure from most multiagent systems. This decision was made in order to ensure the following system conditions [16]:

1) Agents do not increase in size even with an increase in interactions between agents
2) Each node in the system has a consistent view
3) Network overhead is minimized
4) No central point of failure
5) Communications between agents are completed within a reasonable time frame

First, keeping agent size from increasing is accomplished by requiring agents to keep just the unique names of other agents they communicate with and not location information. Second, information sharing between the nodes is what maintains a consistent view. This sharing is accomplished by the use of two mobile agents, the communication agent and the wanderer agent. The communication agent keeps the AgentDirectory information current on each Carolina server. This agent continuously moves between the servers merging its information with each server's AgentDirectory to reflect the most accurate information. The wanderer agent keeps track of where all the servers are physically located. On startup of a server this agent is sent to notify the other Carolina servers about the new server and to give the locations of the other servers to the new server. On shutdown of a server the wanderer for that server moves and tells the other servers of the shutdown.

Third, the need for broadcast messages has been virtually eliminated because of the communication protocol used. This conclusion follows directly from the above discussion about consistent views. Agent locations are stored in each server's AgentDirectory; therefore, an agent does not have to broadcast a message to find a particular agent.

Fourth, there are no centrally located lookup tables or directories for agent locations; hence there is no single point of failure. Finally, since each node in the system has a consistent view, communications can be completed within a reasonable time.

Communication between agents running on Carolina servers is accomplished by passing serialized Java objects. Messages are posted to the server using the MessageManager. The MessageManager decides whether the message is addressed to a local agent or an agent that has moved to a remote host. If the agent is not local, the MessageManager forwards the message to the last known destination of the agent. However, if the agent is local, the message is put into the MessageDirectory where it remains until the agent it is addressed to reads it from the server. This means that Carolina agents must constantly poll the server to see if they have any messages waiting for them.

Messages can be addressed by agent ID or by agent type with the former allowing for interaction between specific agents, while the latter allows an agent to communicate with any other agent of a particular type. For messages sent by agent ID, the message is sent to the server where the agent is active and put in the MessageDirectory until the agent with the correct ID reads it. However, when messages are sent by agent type, any agent of that type can read the message. Once the message is read it is removed from the MessageDirectory and no other agents of that type can read the message. Service brokering is a good example of how these two addressing methods work. When a new agent comes into the system, it can announce itself and the service it provides to a broker agent. The agent ID would be one piece of information recorded by the broker. When another agent requests a list of service providers from the broker, it receives a list of agent IDs and hosts. The requesting agent can then send messages directly to the agent that provides the required service.

In Carolina, mobility is also handled by Java's object serialization capability, which takes the identity and state of a class and encapsulates it. Once serialized, the state and identity is passed to another agent platform and reconstructed. In Carolina, the program code for each type of mobile agent resides on each server so the code does not have to be passed when an agent moves. Each Carolina agent class has three basic methods called job, arrival and leave as shown in Figure 93. When an agent is created or

instantiated, execution starts in the birth method. If a destination machine (IP) has been specified before the end of the job method then the leave method is executed and the agent is passed to the specified machine. If a destination IP address has not been specified then the death method is executed and the agent dies.

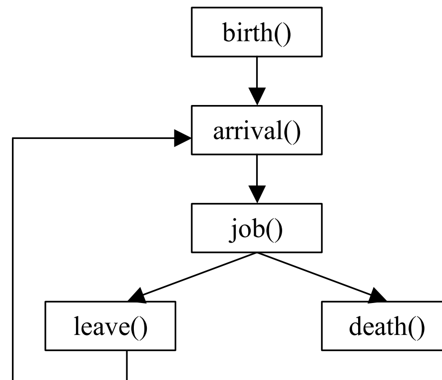


Figure 93. Agent States in Carolina [16]

New mobile agents arrive through the AgentServer port and are then processed by the AgentManager. If the intended destination is the local machine the AgentManager registers the agent in the AgentDirectory, deserializes it, and then hands it over to the ExecutionContainer, which provides a thread for it to execute. When an agent is created and its intended destination is not local, the AgentManager registers the appropriate information with the AgentDirectory and the agent is serialized and sent through the AgentClient port.

A.4.5 Summary

Dynamic agents need certain capabilities provided by their execution environment. FIPA has defined an agent management reference model that defines many of these capabilities or logical components. The dynamic agent systems discussed above, Concordia, Telescript/Odyssey, Aglets Workbench and Carolina, have at least the following three components: Agent Platform, Agent Management System and Agent Communication Channel. All four platforms were written in Java to provide hardware and operating system independence.

Most of the differences between the platforms are either in functionality provided, the way in which agents communicate or how agent mobility is handled. Concordia provides the most functionality in terms of the FIPA logical components but takes away some of the autonomous ability of the agents by using an itinerary. Carolina does not support direct agent-to-agent communication for reasons outlined above. Telescript/Odyssey is the only platform where the agent moves by calling a command within the agent system itself. The other platforms have "job" methods that once they are finished executing the agent will move if required.

A.5 Summary

This chapter provided background information on all the critical pieces of agent-based systems as defined by researchers in the field of Artificial Intelligence. These critical pieces included: the definition of agents, the definition of dynamic agents, evaluation of current Agent-Oriented Software Engineering (AOSE) methodologies and an evaluation of current dynamic agent platforms. Agents are a natural extension of objects from the object-oriented paradigm but are autonomous, cooperative, perceptive and pro-active. Dynamic agents are agents with the following additional abilities: cloning, instantiation and mobility. AOSE differs from object-oriented in terms of the agent-oriented abstractions used to model a system. These abstractions include agent-oriented decomposition, characterizing complex systems by subsystems, interactions, and organizational relationships to include building in mechanisms for agents to flexibly form, maintain, and disband these organizations. Finally, special agent platforms are required to handle the execution of dynamic agents. Agent platforms are software systems with the ability at a minimum to provide places for dynamic agents to execute and employ the dynamic abilities described above. A thorough understanding of each of these pieces was essential in order to correctly integrate dynamic agents into an existing AOSE methodology.

B. Transformation Models

This appendix is a summation of design models defined in Chapter II of Sparkman's thesis [18]. The models described here will be the models used for the transformations for this thesis in Chapter III and are a subset of the design models defined by Sparkman. These models are formally defined and the semantics of the models are clarified to ensure predicable behavior of the transformations.

B.1 Design Model Definitions

This section defines the types that make up the MaSE design models. Each type in the models is defined by using the object format exhibited in Figure 94. Curly brackets **{** and **}** represent sets and square brackets **[** and **]** denote attributes are sequences of a certain type. The Unified Modeling Language (UML) syntax is used in graphical class diagrams that show how the types in the models are related to each other. The UML class diagram in Figure 95 shows the Agent Class, Component Class and StateTable Class types used to define these models. Aggregate components in the class diagrams become an attribute for the type that represents the parent class in the aggregate relationship [18].

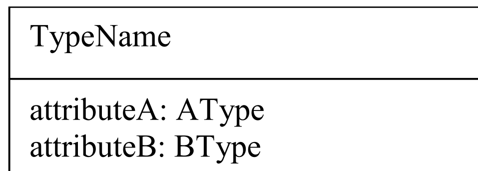


Figure 94. Example Graphical Representation of a Type [18]

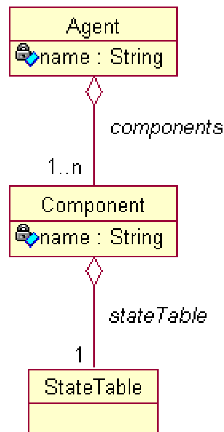


Figure 95. Class Diagram for the Types Used in the Design Phase of MaSE [18]

B.1.1 Agents

The Agent Class Diagram is the first model in the MaSE design phase. It shows the agent classes in the system, the roles those agents play, and the conversations defined between the agent classes. An agent type (Figure 96) represents the agents defined in the Agent Class Diagram and is defined by a name, its roles, its components, and the conversations it participates in. Each agent type has a unique name that distinguishes it from any other agent type in the system. The roles and conversations attributes [18] will not be used in the transformation definitions in this thesis.

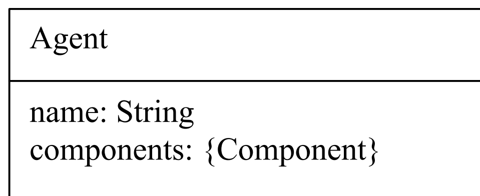


Figure 96. Agent Type [18]

B.1.2 Components

A name and a state table, as used within this thesis, define a component. Figure 97 shows the component type. Agent types cannot have two components with the same name. Also, the name "AgentComponent" is reserved for the agent component of an agent class.

Component
name: String stateTable: StateTable

Figure 97. Component Type [18]

B.1.3 State Tables

StateTables are used to define the behavior of concurrent tasks and components. Figure 98 shows the StateTable type. StateTables are composed of states and transitions. The states attribute represents the set of all possible states that a task or component can be in at any given time while the transitions attribute represents the set of transitions between the states. The UML class diagram in Figure 99 gives a graphical overview of the different types used to define a StateTable and shows the different relationships between them.

StateTable
states: {State} transitions: {Transition}

Figure 98. StateTable Type [18]

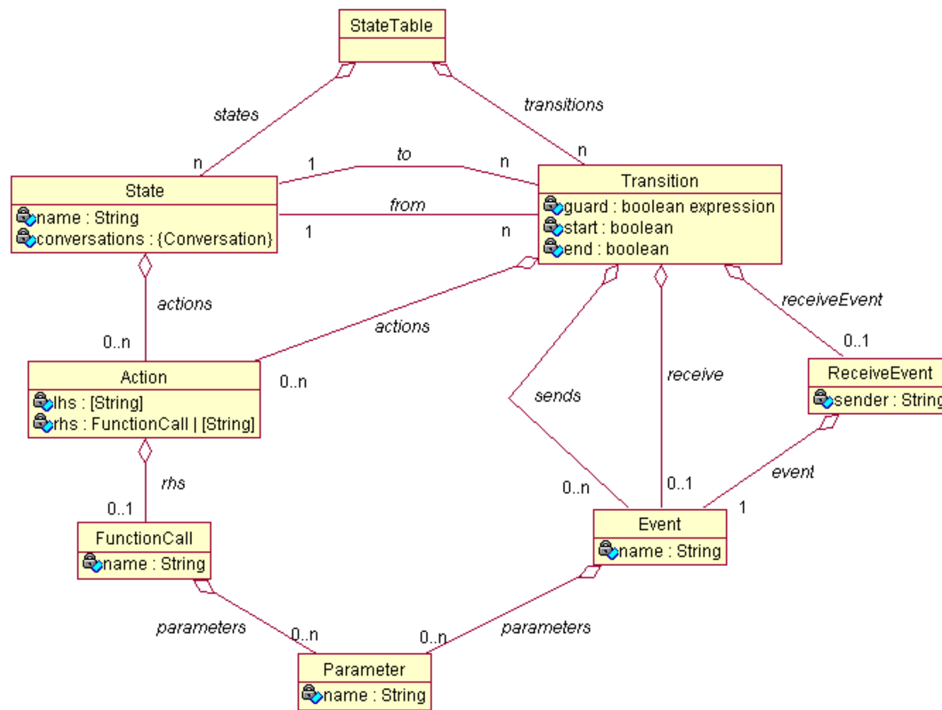


Figure 99. StateTable Class Diagram [18]

B.1.3.1 States

As mentioned in the previous section, state tables are composed of states and transitions. Figure 100 displays the state type that is used within the transformations defined in this thesis. States represent the internal processing of components. They contain a name attribute that must be unique within a given state table and also a sequence of actions that are executed upon state entry. Every state table contains a *start* state with the name "start", which is the beginning of the state table. A state table may contain an *end* state with the name "end", which is the ending of the state table if applicable.

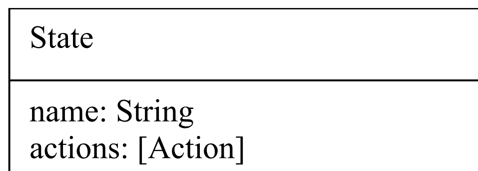


Figure 100. State Type [18]

B.1.3.2 Transitions

Transitions are the other elements that make up a state table. They define the communication within the system and are the connectors between the states in a state table. Transitions are defined by the following: start state, end state, receive event, guard condition, actions and transmission events. The syntax for a transition in a state diagram is:

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

Transitions are assumed to occur instantaneously and either progress an entity from one state to a different state or from one state back to the same state. Actions are executed in the given order before any transmissions are sent. 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 applicable) has been generated.
3. The transition's guard condition (if applicable) 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.

The same transition type is used for both the Concurrent task and Component models and is shown in Figure 101. The difference in the semantics for transitions between the two models is that in the Component model three of the attributes shown for a transition (*receiveEvent, start*, *end*) are not used. The *receiveEvent* is not used because the events on the transitions in the state table for the components are defined to be internal events to other components of the same agent instance. And, the *start* and *end* attributes were only used within [18] to mark the start and end of conversations, which are not are part of the transformations defined within this thesis.

Transition
from: State receive: Event receiveEvent: ReceiveEvent guard: Boolean Expression to: State actions: [Action] sends: {Event} start: Boolean end: Boolean

Figure 101. Transition Type [18]

B.1.3.3 Actions

Actions (or activities) are used to represent reading a percept from sensors, internal reasoning or activating effectors that make changes in the environment. They represent the actual processing within states in a state table. Actions can be defined in the form of functions that can have any number of input parameters and results in the form of a single or tuple value. They can also be used for tuple assignments as well. The syntax for actions is shown below:

```
result = name-of-action(parameter1, parameter2, ... ,parameterN)

<x,y> = position(object)

<a,b> = <x,y>
```

Figure 102 displays the Action type. The attribute, *lhs*, represents the left-hand-side of an assignment statement. It represents the result of the action as a sequence of strings. The attribute, *rhs*, represents the right-hand-side of the assignment statement and is either a sequence of strings or a FunctionCall.

Action
lhs: [String] rhs: FunctionCall | [String]

Figure 102. Action Type [18]

FunctionCalls are defined by a name attribute and a sequence of input parameter types. The FunctionCall type is shown in Figure 103. The Parameter type is show in Figure 104 and is defined by a name attribute.

FunctionCall
name: String parameters: [Parameter]

Figure 103. FunctionCall Type [18]

Parameter
name: String

Figure 104. Parameter Type [18]

B.1.3.4 Events

Events represent messages that are passed, either internally or externally, in a system. Figure 105 displays the Event type. The performative or intent of the message is represented by the *name* attribute while the content of the message is represented by a sequence of parameters as described in the previous section.

Event
name: String parameters: [Parameter]

Figure 105. Event Type [18]

Internal events are distinguished from external events in Concurrent Task Diagrams. External events are represented by a ReceiveEvent type, which is shown in Figure 106. The message being received is represented by an Event type, while the sender of the message is represented as a string.

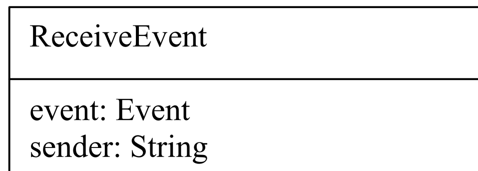


Figure 106. ReceiveEvent Type [18]

ReceiveEvents trigger transitions in Concurrent Task Diagrams. The syntax for ReceiveEvents on a transition is shown below.

```
receive(event, sender).
```

B.2 Summary

This Appendix described the models used within MaSE and the relationships between those models. Those models are made up of different types as defined in Chapter II of Sparkman's thesis [18]. Each type, which was used within this thesis to define the transformations in Chapter III, was defined by attributes and relationships with the other types.

Bibliography

1. Carrez, F., "Agent technology and its applications," Alcatel Telecommunications Review, 1999(1): p. 67-77.

2. Chess, D., C. Harrison, and A. Kershenbaum, "Mobile Agents: Are They a Good Idea?," Mobile Object Systems. Towards the Programmable Internet. Second International Workshop, MOS '9, 1996. Linz, Austria: Springer-Verlag.

3. DeLoach, S.A., "Specifying Agent Behavior as Concurrent Tasks: Defining the Behavior of Social Agents," Autonomous Agents 2001.

4. DeLoach, S.A., "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.

5. DeLoach, S.A., M. Wood and C. Sparkman, "Multiagent Systems Engineering," International Journal of Software Engineering and Knowledge Engineering, 2001.

6. FIPA, Agent Management Specification, 2000: www.fipa.org.

7. Griffel, F., et al, "Electronic Contract Negotiation as an Application Niche for Mobile Agents," in First International Enterprise Distributed Object Computing Workshop. 1997. Gold Coast, Qld., Australia: IEEE Comput. Soc, Los Alamitos, CA.

8. Jennings, N.R., "On agent-based Software Engineering," Artificial Intelligence, 2000(117): p. 277-296.

9. Joint Chiefs of Staff, Joint Vision 2020, 2000: www.dtic.mil.

10. Lange, D.B., "Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2," IBM Tokyo Research Laboratory, 1997.

11. Lange, D.B. and M. Oshima, "Introduction to mobile agents," Personal Technologies, 1998. 2: p. 49-56.

12. Moulin, B., "The use of EPAS/IPSO approach for integrating Entity Relationship concepts and Software Engineering techniques," Entity-Relationship Approach to Software Engineering, 1983: North Holland.

13. Moulin, B. and M. Brassard, "A Scenario-Based Design Method and an Environment for the Development of Multiagent Systems," in Proceedings of the First Australian workshop on Distributed Artificial Intelligence. 1996: Springer-Verlag.

14. Moulin, B. and L. Cloutier, "Collaborative work based on multiagent architectures: a methodological perspective," Soft Computing: Fuzzy Logic, Neural Networks and Distributed Artificial Intelligence, 1994. 261-296: Prentice Hall.

15. Robinson, D.J., A Component Based Approach to Agent Specification. MS thesis, AFIT/GCS/ENG/00M-22. School of Engineering, Air Force Institute of Technology, Wright Patterson Air Force Base, OH, 2000.

16. Saba, G.M. and E. Santos, "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.

17. Shehory, O., et al., "Agent Cloning: An Approach to Agent Mobility and Resource Allocation," IEEE Communications Magazine, 1998. 7: p. 58-67.

18. Sparkman, Clint, Transforming Analysis Models into Design Models for the Multiagent Systems Engineering (MaSE) Methodology, MS thesis, AFIT/GCS/ENG/01M-12. School of Engineering, Air Force Institute of Technology, Wright Patterson Air Force Base, OH, 2001.

19. Sycara, Katia, "Multiagent Systems," in AI Magazine, 1998. vol 10, no2:p. 79-93.

20. United States Air Force, Air Force Vision 2020, 2000: www.af.mil.

21. White, J.E., "Mobile Agents White Paper," General Magic, 1994.

22. Wong, D., et al, "Concordia: An Infrastructure for Collaborating Mobile Agents," Mobile Agents: First International Workshop, 1997.

23. Wood, M.F. and S.A. DeLoach, "An Overview of the Multiagent Systems Engineering Methodology," The First International Workshop on Agent-Oriented Software Engineering (AOSE2000), 2000. Limerick, Ireland.

24. Wooldridge, M. and N.R. Jennings, "Intelligent Agents: Theory and Practice," Knowledge Engineering Review, 1995. 10: p. 115-152.

25. Wooldridge, M., N.R. Jennings, and D. Kinney, "The Gaia methodology for Agent-Oriented Analysis and Design," Journal of Autonomous Agents and Multiagent Systems, 2000. 3(3):285-312.

Vita

Athie Lee Self was born in Washington D.C. in October 1969, and graduated from Rome Free Academy High School, New York in May 1987. He earned a Bachelor of Science degree in Computer Science from the State University of New York Institute of Technology at Utica/Rome in May 1993. He earned a commission as a 2nd Lt in the United States Air Force Reserve after completing Officer Training School in March 1994. His first assignment was with Communications Systems Center, Tinker AFB, OK, as a computer programmer/analyst from August 1994 to August 1997. His next assignment was with the Joint Staff Support Center, Pentagon, VA, as a Global Command and Control System (GCCS) Technical Database Manager (TDBM) from August 1997 to August 1999. Athie is married to the former Janet Joachim of Albany, New York and has two daughters, Maggie and Anna.

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)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
02-03-2001	Masters Thesis	APR 2000- MAY2001

4. TITLE AND SUBTITLE
DESIGN AND SPECIFICATION OF DYNAMIC, MOBILE, AND RECONFIGURABLE MULTIAGENT SYSTEMS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Self, Athie L., Captain, USAF

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 45433-7756

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/01M-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory -- Human Effectiveness Directorate
Attn: Scott M. Brown, scott.brown@wpafb.af.mil
2255 H Street
Wright-Patterson AFB, OH 45322
DSN: 785-8883

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
Advisor: Major Scott A. DeLoach, DSN 785-3636 x4266, scott.deloach@afit.edu

14. ABSTRACT
Multiagent Systems use the power of collaborative software agents to solve complex distributed problems. There are many Agent-Oriented Software Engineering (AOSE) methodologies available to assist system designers to create multiagent systems. However, none of these methodologies can specify agents with dynamic properties such as cloning, mobility or agent instantiation. This thesis starts the process to bridge the gap between AOSE methodologies and dynamic agent platforms by incorporating mobility into the current Multiagent Systems Engineering (MaSE) methodology. Mobility was specified within all components composing a mobile agent class. An agent component was also created that integrated the behavior of the components within an agent class and was transformed to handle most of the move responsibilities for a mobile agent. Those agent component and component mobility transformations were integrated into agentTool as a proof-of-concept and a demonstration system built on the mobility specifications was implemented for execution on the Carolina mobile agent platform.

15. SUBJECT TERMS
Agent-Oriented Software Engineering, Multiagent Systems Engineering, Mobile Agents

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Major Scott A. DeLoach, ENG
U	U	U	UU	154	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 x4716

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18