Formal Object State Model Transformations for Automated Agent System Synthesis

THESIS

David Wesley Marsh, Captain, USAF

AFIT/GCE/ENG/00M-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/00M-03

Formal Object State Model Transformations for Automated Agent System Synthesis

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

David Wesley Marsh, B.S.

Captain, USAF

March 2000

AFIT/GCE/ENG/00M-03

Formal Object State Model Transformations for Automated Agent System Synthesis

David Wesley Marsh, B.S.

Captain, USAF

Approved:

Thomas C. Hartrum	9 Mar 2000
Dr. Thomas C. Hartrum Committee Chair	Date
[illegible]	9 Mar 2000
Maj. Scott A. DeLoach Committee Member	Date
Robert P. Graham, Jr.	9 MAR 2000
Maj. Robert P. Graham, Jr. Committee Member	Date

Acknowledgements

I would first like to acknowledge God as the provider of life and intellect and thank Him for giving me the ability to rise to AFIT's challenges. Next I must express my deepest gratitude to Kristina for hanging in there and taking care of our home so that I could dedicate more of my time to school work; without her help both as proof-reader and as home-maker I could not have accomplished nearly as much. To Dr. Hartrum I extend special thanks for giving me the freedom to pursue my thesis in my own way and for providing insightful and challenging feedback along the way. I am also indebted to Maj DeLoach and Maj Graham for their assistance with the thesis development; their feedback was vital to a successful end product. Finally I thank my classmates for making the long days (and nights!) at school profitable and, dare I say, enjoyable.

David Wesley Marsh

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
AFIT	Air Force Institute of Technology
AgDL	Agent Design Language
AgML	Agent Modeling Technique
AI	Artificial Intelligence
AO	Agent Oriented
AST	Abstract Syntax Tree
AWSOME	AFIT Wide Spectrum Object Modeling Environment
COIL	Common Object-Oriented Imperative Language
DOM	Domain Specification Abstract Syntax Tree
GOM	Generic Object Model Abstract Syntax Tree
MARSH	Multi-Agent Relationships via Socket cHannels
MaSE	Multiagent System Engineering
OMT	Object Modeling Technique
OO	Object Oriented
UML	Unified Modeling Language

AFIT/GCE/ENG/00M-03

Abstract

Automated agent system synthesis is the process of generating code from a requirements specification with appropriate inputs from the software engineer. Object-oriented (OO) specifications are frequently used to model intelligent software agent systems and software requirements in general; formal representations capture precisely the intentions of the specifier. Portions of OO specifications can be classified as the structural, functional, and state (or dynamic) models; major strides have been taken in the development of transformations for creating code from formal OO specifications, specifically the structural and functional aspects, and are captured within the AFIT Wide-Spectrum Object Modeling Environment (AWSOME). This research creates a methodology for the automatic transformation of the dynamic model into structural and functional components which can then be exploited for the generation of executable code exactly reflecting the original intent of the requirements specification. The integration of agent communication protocols within this context is addressed, providing a methodology for the incorporation of various agent-to-agent and agent-to-human interaction schemes. Feasibility is demonstrated through the application of transformations to a formal requirements model within AWSOME resulting in executable code.

Formal Object State Model Transformations for Automated Agent System Synthesis

I. Introduction

A client from the maintenance analysis section walks in the door of the Base Computer Programs Support Branch and asks for a computer program that will use information in existing databases to identify all abnormally high break rates for the F-15s both on base and around the world. The program needs to forward those items to the appropriate on-base maintenance shop supervisors by e-mail and print "personalized" letters highlighting safety-critical items that appear on the list to the squadron commanders and appropriate staff members. The software engineer turns to the computer and types in a few lines of specification after clarifying a few more details with the client. A few minutes later the software is ready and the maintainers and their supervisors will soon have a "heads-up" for potential problem components or systems in the aircraft.

This is a simplistic example of what could become commonplace in the future: making use of software tools that generate executable code automatically from high-level specifications, statements of requirements, or graphical models. While much progress has been made in this area, the field is still quite young and additional work to realize the ultimate goal is necessary.

The past five years of research and development at the Air Force Institute of Technology (AFIT) have yielded a software implement referred to as the AFIT Wide-Spectrum Object Modeling Environment (AWSOME). Ultimately AWSOME will automatically transform entire program specifications into executable code. *Z* representation of the object oriented (OO) paradigm has shaped AWSOME's structure but this is of little import to the broader scope of what it encapsulates. AWSOME's basic function is to transform a formally correct representation of an object model into a domain abstract syntax tree (DOM) and then to transform the DOM into another abstract syntax tree (AST), the generic object model (GOM), through the use of formal rules. Finally, AWSOME generates code for any programming language whose grammar is defined in the system. The outlook is quite good for AWSOME to generate useful executable code from complete formal specifications of OO models in the near future.

The example that began this thesis will likely make use of an automated transformation system such as AWSOME as well as an area of research that is still in its infancy: artificial intelligence (AI) agents[1]. Exactly what a software agent is and how it interacts with its world is not much beyond purely conceptual representation. Models for representing and implementing agents are still immature, though recent AI research has begun to formalize an approach to agent creation. Developing the principles of automated agent system software generation from a formal system specification is one more step toward the goal of designing programs and not having to code them manually.

1.1 Problem

The problem currently being addressed is the development of a methodology for the automatic conversion of an agent system from specification to executable code. Feasibility is demonstrated through an implementation using AWSOME.

When specifying agent systems the first questions to ask are 1) what characteristics are common to the agent domain and 2) what model best represents those characteristics. Some qualities are undisputed, such as the ability to "remember" information and the capacity to perform actions. Other areas such as autonomicity are much more subjective, both in definition and in pertinence. Because AWSOME can represent *Z* specifications well and previous research shows that conversion from any OO design model into *Z* is straightforward [30], a likely continuing requirement is to capture necessary characteristics in an OO-type model.

Developing transformations that must take place between the DOM representation of software system analysis and the GOM specification of software design is the most daunting task in this research effort. Research and implementation are focused on the dynamic object model, but requires an evaluation of the existing transformations in AWSOME and

[1]also referred to in this document by variants such as "intelligent agents," "software agents," or simply "agents"

its predecessor, *AFIT*tool. *AFIT*tool has many portions of this transformation in place, with the overall approach shown in Figure 1. While much of the system has been addressed by others as identified in Section 1.2, the dynamic model is more thoroughly addressed in this research.

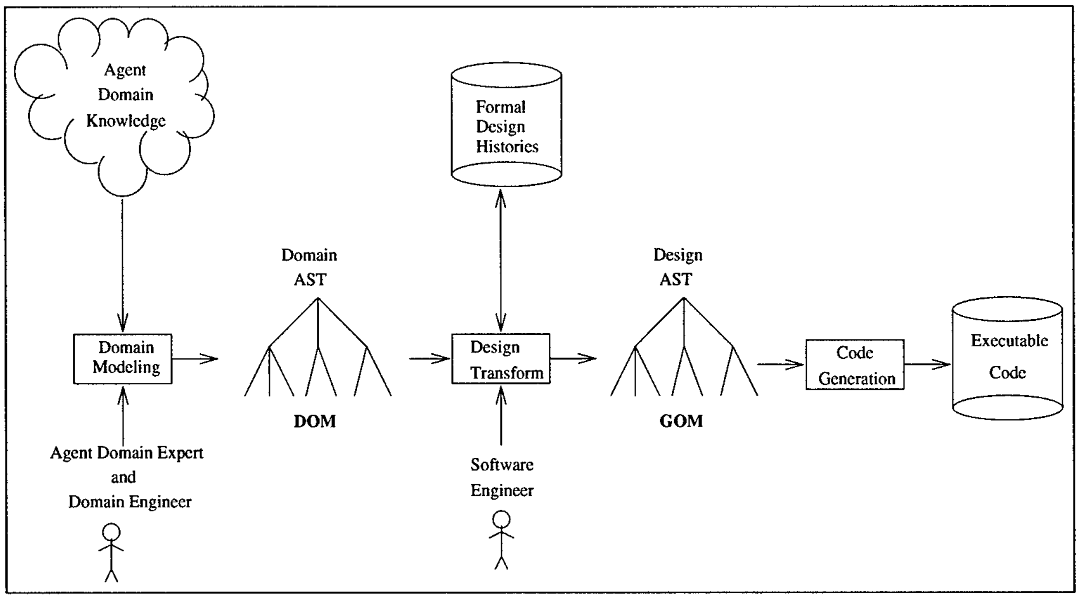


Figure 1. AWSOME and *AFIT*tool Process Model

1.2 Initial Assessment of Past Effort

*AFIT*tool can currently parse an OO specification into the DOM from a representation in LaTeX *Z*. It can then translate the structural and functional model representations into a GOM abstract syntax tree. While the concept has been proven for both primitive and aggregate OO classes, only the structural and functional OO models have been addressed to a detailed level [25, 39]; translation of the dynamic model remains. A system also exists for translating OO models from a Rational Rose representation into *Z*, which can then be parsed into *AFIT*tool [30]. The ease with which *Z* and the OO approach work together is a key reason these two have been implemented in *AFIT*tool.

Methodologies for describing agent systems exist with a varying degree of thoroughness as detailed in Chapter II. One such OO representation that provides a high-level approach to agent system design appears in Kendall's work [23]. A more formal approach dedicated to *Z* representations of agent specifications is presented by Luck and d'Inverno [9]. Other works also provide methods for agent system analysis and design that could conform nicely to OO and *Z* models [6,17,22,23,38].

The output from *AFIT*tool is currently Ada code, which has been shown to be an accurate implementation of the initial specification [25,39]. Again, this cannot currently be accomplished with the entire OO model but the concept has been proven and can be expanded by future research.

Because Refine [33] handles information transformations easily and has powerful operations for working with ASTs, this environment was chosen for *AFIT*tool implementation. Further modifications or additions to *AFIT*tool need not be in Refine and, in fact, may be more desirable in a more common language environment. Using the same concepts that are provided by the Refine implementation can lead to the development of a similar tool's instantiation in many other languages as well; AWSOME is a tool designed to do exactly that in the Java language.

Many different sources contributed to the DOM and GOM structures used in AWSOME. Rumbaugh's Object Modeling Technique (OMT) provides a general object-oriented domain model [35]. Sward developed the Generic Object Model (GOM), a general OO programming model [37]. The Common Object-Oriented Imperative Language (COIL), developed by Graham [14], provides a language-independent representation of program designs. Finally, the Unified Modeling Language (UML) has also influenced the AWSOME model [31]. All of these have had significant impact on the analysis and design representations now used in AWSOME.

1.3 Scope

As previously stated, the goal of this research is to produce a methodology for automatically converting an agent system from specification to executable code and to demonstrate its feasibility through AWSOME. This researcher develops a system for transforming generic specifications into various agent or non-agent implementations for many different applications. Because the diversity of approaches for defining and implementing agency is extremely broad, it is necessary to focus on a subset rather than the universe of intelligent software agents.

The AWSOME system has been altered through many different research efforts. Therefore some inconsistencies exist among semantics, methodologies, and implementation decisions. Further design and development of AWSOME within this research is focused only on those aspects relating directly to dynamic model manipulations in the context of agent system development.

1.4 Research Approach

This research creates a specification of a basic agent system and extends AWSOME to support the automatic generation of executable code for this agent system. The steps are: 1) develop or refine a model for capturing agent systems, 2) formally specify the model with a focus on the dynamic characteristics (previous research has focused on the structural and functional aspects [25, 39]), 3) represent the specification in the AWSOME analysis model, 4) transform the model from the analysis specification into a design representation, and 5) generate executable code. The analysis specification model is also referred to as the "DOM" and the design specification model as the "GOM" throughout this thesis.

The first step is to develop a model and specification for the agent system. An examination of existing methodologies for specifying agents and agent systems provides the basis for determining relative values of existing representations for this research. Once a methodology has been selected, an approach to agent system representation is developed and specified. Without a formal model, the demonstration of the results of this research would be impossible; therefore formal specifications are required. Since AWSOME handles OO representations well and previous research has developed the formal syntax (using *Z* specifications) and semantics for structural and functional OO components, specifications mirror the OO paradigm; syntax and semantics for the dynamic model, including states, events, and transitions is thoroughly developed in this thesis and representations are developed for the AWSOME DOM.

Transformation of the DOM into the GOM constitutes a significant portion of the work handled here. A set of rules and functions must maintain proper definitions of the interactions between classes/agents and their external connections. The final step requires the extension of the existing AWSOME system to generate executable code from the GOM. The last phase, code generation, is not the focus of this research and, while addressed, is not fully explored. After transformations are developed the theory is applied to an example system. Three communication protocols are used to demonstrate the adaptability of the automatically generated code to various inputs and outputs to the system.

1.5 Document Layout

This thesis is presented with an overview of the application of formal methods to dynamic model manipulations and various approaches to agent description and specification used for the creation of agent and agent system specifications Chapter II. Chapters III through V present the three key contributions of this research:

1. The formal specification of the syntax and semantics for the dynamic model within AWSOME provides for an unambiguous input model in Chapter III.
2. Chapter IV provides the definition of five dynamic model transformations to represent a model of states, events, and transitions within structural and functional components which can be harnessed directly for code generation. Mathematical expressions capturing the effects of the transformations provide the formality required for future proofs of correctness preservation within the transformation system.
3. The above two contributions are implemented within AWSOME and demonstrated in a simple system using three separate communication protocols in Chapter V.

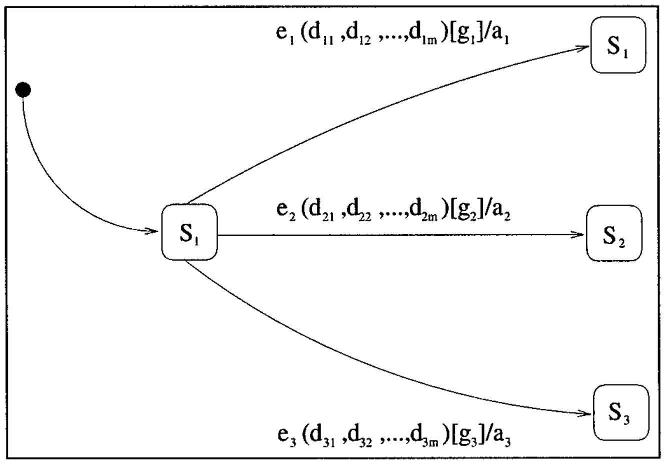
Chapter VI completes this thesis by providing a summary of these contributions, recommendations for further research, and other concluding comments.

II. Background

This chapter provides background information to assist the reader in understanding the concepts discussed in this thesis. Topics included are the application of formal methods to dynamic model manipulations within the object oriented paradigm (Section 2.1) and various approaches to agent description and specification used for the creation of agent and agent system specifications (Sections 2.2 and 2.3).

2.1 Formal Methods and the Dynamic Model

The dynamic model as used in the OMT [35] graphically depicts the behavior of a system by using a state diagram, demonstrated in Figure 2. The rounded boxes represent states an object may visit, while the text associated with each arrow provides information about the causal event ("e_x"), data items associated with the event ("(d_{xx})"), guarding conditions ("$[g_x]$"), and actions resulting from the event ("$/a_x$").



$e_1\,(d_{11}, d_{12}, \ldots, d_{1m})[g_1]/a_1$
S_1
$e_2\,(d_{21}, d_{22}, \ldots, d_{2m})[g_2]/a_2$
S_1
S_2
S_3
$e_3\,(d_{31}, d_{32}, \ldots, d_{3m})[g_3]/a_3$

Figure 2. Sample OMT State Diagram

Wang, et al., have developed a formalized syntax and semantics that, when applied to the state diagram, merge the formalisms required for application to automated processing and transformations with the simplicity of graphically-based design [40]. Their ongoing work is aimed at formulating methods for transforming an analysis specification into a design specification. The first step toward dynamic model formalization is the description of the semantics to be used. The behavior of an object, defined as the communications and operations that occur between the object and the environment, is fully specified within the dynamic model. Modeling of the environment is limited to the various objects; therefore, modeling of communications is limited to inter-object communications.

States are used when defining the various interaction sequences that are allowed within the system. *Events* represent inter-process communication. A *guard condition* describes the circumstances required for a state transition to occur and is represented by a set of predicates. *Transitions* are simply the state changes caused by some event. Each transition whose starting state differs from its ending state causes state changes. *Actions* and *activities* describe the operations performed by the object during a transition and upon entry into a state, respectively. The distinction between the two is somewhat fuzzy, separating "instantaneous" operations from those performed over a period of time. The designer must select which model best applies to the operation in question.

Bolognesi and Brinksma also use a formal specification language for formalizing state diagrams to capture the behavior of individual objects [2]. Their model is extended to accommodate aggregate objects through a parallel composition of individual state diagrams.

2.2 Agent Specification

Defining agency and agent systems is a daunting proposition at this time. Researchers use many different characteristics to define agency; some of these are presented in Section 2.2.1. While many alternative views have been asserted [4–6, 10, 13, 18, 19, 22, 23, 26–29, 32, 36] only a representative sample of this diversity is discussed below. Agent modeling is approached from equally diverse positions [6, 10, 19, 24, 26, 29, 32, 36, 41]. Several of these models are selected for their applicability or ease of adaptation to this research and are reviewed in Section 2.2.2. Other models have been proposed but are not reviewed here because either they are similar to those presented or they do not provide views easily applied to automated synthesis activities.

2.2.1 Agent Characteristics. While the use of the term *agent* is overloaded, ambiguous, and widely misunderstood[1], agent architectures abound and many seemingly ad hoc agent systems are appearing in all corners of the computer software world as practitioners create software programs with some level of intelligence or utility and call them agents. Some articles describe agency in the broadest of terms. Lander, for example, takes the view that an agent is "any relatively autonomous software component" that adds expertise to a design and can include communications [27:19].

On the other hand, Foner includes detail in his description of exactly what an agent is. Table 1 lists his "crucial notions" and what they mean in relation to the definition of an agent. Foner states that while each of these characteristics may be present to greater or lesser degrees, they describe aspects that may be useful in designing agents.

Table 1. Foner's Crucial Notions of Agency [13:35-37]

Autonomy	the agent's ability to initiate on its own those actions that will benefit the user
Personalizability	the quality that enables an agent not only to learn what the user's agenda is, but also to remember the available information for use in later settings. The user does not have to program every element of the agent because the agent learns by observing actions and remembering
Discourse	two-way communication takes place in which the agent and user interact; the agent and the user work out a "contract" that determines who will do which part of the task
Graceful Degradation	the agent's ability to complete portions of the task even when some steps cannot be accomplished; if communications between the agent and user are not completely clear (or are disrupted) or if the agent is incapable of accomplishing the entire portion of a task delegated to it, the agent must provide as much of the desired result as possible
Cooperation	required two-way communication in which the user and agent decide together how a goal will be accomplished; another approach to *Discourse*

DeLoach contends in his multiagent systems engineering (MaSE) approach that agents can be modeled as active objects [6]. Agents possess the four primary traits iden-

[1]Misunderstanding the definition of *agent* may not be possible given the diversity of opinions on the subject!

tified in Table 2. This list demonstrates that agents differ from objects in several ways. Agents are active, exhibit goal-directed behavior, and share a common messaging language with other agents whereas objects are passive reactors to the environment and handle message passing differently depending on the given class. Therefore, the basic picture of an agent is that of an object with the added attributes of goals and standardized communications. His approach presents the characteristics of agency as an abstraction of the OO paradigm. Because the designer may model both traditional objects and agents there is no need to define exactly what constitutes an agent.

Table 2. DeLoach's Traits of Agency [6]

Autonomicity	the ability to act without being controlled by an external entity
Cooperativeness	the ability to communicate and act in coordination with other entities
Perceptiveness	the ability to sense the environment and respond to it
Pro-activeness	the ability to act decisively to accomplish goals

Kendall, et al. develop a fairly broad picture of an agent [23]. Their definition of an agent includes up to the eight distinct characteristics in Table 3; the first four describe "weak" agency while the last four add "stronger" qualities. The model further describes agents as specialized objects, adding the traits of "reasoning, pro-activity, migration, concurrency, and collaboration" [23:3] to the OO paradigm. Implicit in another of Kendall's approaches to agents are the characteristics of carrying out actions, maintaining goals, possessing responsibilities, performing tasks, developing (or retaining) expertise, and communicating in some form with other entities [22].

In their work with specifications of agents and agent systems, Luck and d'Inverno [29] describe agents as specializations of objects much like DeLoach [6]. The Z specification, however, identifies a formal framework for two levels of agency: the generic agent and the autonomous agent. An object possesses attributes, actions it can perform, states it can traverse, and interactions with its environment while an agent maintains goals and a perception of both the environment and how its actions affect its goals and the environment. An autonomous agent is further specialized to include motivations which affect the perceptions received and the actions performed.

Table 3. Kendall's Agent Characteristics [23:1-2]

Autonomous	operate without constant directions from external sources, able to move from one (electronic) location to another
Social	"interact with other agents"
Reactive	perceive the environment and act in response to changing perceptions
Pro-active	operate on the environment to affect changes and not just wait for the environment to change them
Mentalistic notions	possess and utilize beliefs, desires, and intentions
Rational	perform those "actions which further its goals"
Veracity	(self-explanatory)
Adaptable	learning ability

2.2.2 Agent Models and Specifications. The variety of descriptions of the agent characteristics above are helpful for understanding what is to be modeled. Two approaches are presented below (with another following in Section 2.3.2.1) providing for both a variety of representation styles and the key background helpful in later chapters.

2.2.2.1 A Z Approach. Luck and d'Inverno [29] present two key reasons for using a *Z* representation:

1. The modularity and abstraction levels *Z* provides communicate the structured nature of agents including the properties of inheritance and specialization.
2. *Z* is useful for bridging the gap between formal specifications and implementation.

The fact that *Z* is widely accessible within the artificial intelligence community is noted as an additional advantage to this model.

Several requirements are presented as prerequisites for a formally specified model to be considered useful:

1. The specification must be clear and readable.
2. Models must provide complete definitions of concepts and terms and allow for alternative design approaches during development.
3. A good formal specification methodology must provide a way to create generic specifications as well as specializations as appropriate or desired.

4. The designer must be allowed to choose the level of abstraction of the specification.

Luck and d'Inverno assert that *Z* fulfills these requirements.

This formal specification represents objects, agents, and autonomous agents, identifying certain characteristics required for agency. The *Z* language is not tied to a particular architecture, providing the designer flexibility in the level of detail and general approach when creating a model. Table 4 presents Luck and d'Inverno's definitions of the "types" used in this specification scheme and Figure 3 shows the *Z* structure of the environment, objects, agents, and autonomous agents.

Table 4. Types for Use in Z Agent Specifications

Attributes	perceivable features in the environment
Actions	discrete events that can alter the environment
Goals	something to be achieved in the environment
Motivations	preferences that lead to goals

The entity hierarchy begins with the environment, which is simply a collection of attributes. An object contains a subset of environmental attributes with the addition of actions. Agents incorporate goals into objects while autonomous agents extend even further to include motivations. Also discussed are *Z* representations of the perceptions, actions, and states objects and object specializations may possess. Because it deals more with agent systems, this approach is addressed more thoroughly in Section 2.3.2.1.

2.2.2.2 Agent Oriented Modeling Technique. This section presents two Agent Oriented (AO) techniques for developing intelligent agents. The first presents a high level view while the second model delves deeper into exactly what an agent is and how it acts.

According to Wooldridge, et al. [41] agent descriptions can be derived from the roles the agents play in a system. Assuming a closed system in which all components work together to accomplish common goals, the software engineer develops the role schema depicted in Table 5; each role schema draws together all information pertinent to the role that is needed during agent design. Once this representation is complete, design is continued by transforming the analysis model into a lower level of abstraction that

$[ATTRIBUTE, ACTION, GOAL, MOTIVATION]$

$$\begin{array}{|l}
\hline
\textit{Env} \\
\hline
\textit{Environment} : \mathbb{P}\, ATTRIBUTE \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Object} \\
\hline
\textit{Env} \\
\\
\textit{capableOf} : \mathbb{P}\, ACTION \\
\textit{Attributes} : \mathbb{P}\, ATTRIBUTE \\
\hline
\textit{Attributes} \subset \textit{Environment} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Agent} \\
\hline
\textit{Object} \\
\\
\textit{Goals} : \mathbb{P}\, GOAL \\
\hline
\textit{Goals} \neq \{\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{AutonomousAgent} \\
\hline
\textit{Agent} \\
\\
\textit{Motivations} : \mathbb{P}\, MOTIVATION \\
\hline
\textit{Motivations} \neq \{\} \\
\hline
\end{array}$$

Figure 3. Z Representation of Objects and Agents

traditional design techniques can handle. To reach this level the identified roles are mapped nearly one-to-one onto types of agents that handle a particular role. The agents are designed to provide the specific services identified with their roles and to communicate via unspecified communication links to external resources or other agents. Details in all areas are left for the engineer to develop using the environment of choice.

The next model, developed by Kinney and Georgeff, begins with a look at the roles agents play in a system. They develop an agent using three sub-models: the belief, goal, and plan models [24]. These models describe the agent's "informational and motivational state and its potential behavior." Each of these models can be represented formally using

Table 5. Template for Role Schema [41]

Description	*short English description of the role*
Protocols	*protocols in which the role plays a part*
Permissions	*"rights" associated with the role*
Responsibilities	
Liveness	*self explanatory*
Safety	*self explanatory*

predefined sets and types. The belief model describes the knowledge base of the agent, including its knowledge about both its internal state and its beliefs about its environment. The state of the agent is determined in part by its belief state. Potential goals of the agent are represented in the goal model. A goal set specifies the domain of the agent's goals and any events to which the agent may respond. Goal states are simply the goals that may help specify the initial state of the agent.

Plans that may be used by the agent in achieving its goals are contained in the plan set, a part of the plan model. These plans are not like an OO description of system behavior but encompass the beginning, intermediate, and ending states to be passed through en route to goal achievement. Each plan has three properties. 1) "Priority" determines the ordering of plan execution in concurrent systems, 2) "Precedence" determines the ordering of plan execution when a new goal is introduced, and 3) "No retry" identifies whether or not the agent should attempt to execute a failed plan again.

An agent represented in this model would be described in detail: exactly what actions it would take in any given situation, what pieces of knowledge the agent could possess, and what the agent's plan would be for any given circumstance.

2.2.3 Agent Representation Summary. The AO model approaches the problem from the aspect of typical agent components and properties such as planning, goals, communications, and beliefs (or perceptions). While this model captures agent properties differently than other languages, more familiar models facilitate representations of the same properties with greater ease.

MaSE uses formal representations, both graphical and predicate logic-based, to develop agents. The Z-based approach also presents a well-defined structure of agents, providing the formal language that is a prerequisite to automated synthesis.

2.3 Agent System Specification

Having considered what constitutes agency and how agents may be represented during design, a closer look at the unique qualities of agent systems that may require consideration during modeling is warranted. For a high-level overview of agent system characteristics, Section 2.3.1 selects two approaches to the identification of agent system characteristics [8, 38]. One identifies the requirements for formal system representations, while the other looks at the question from a more pragmatic standpoint. Other viewpoints exist [4, 5, 12, 18–20, 28] but are not presented here because they either apply to specific problem domains or simply do not add significantly to the two already reviewed.

Section 2.3.2 selects two agent system models for review, one for its ease of application [6] and the other for its formal approach [9]. Many other agent system models have been accomplished within specific domains and are not reviewed here [4, 5, 12, 19–21, 26, 28, 34]. Because all agent systems must interact with humans to be useful, this particular system challenge is addressed in Section 2.3.2.3. One approach is explored for its handling of this interaction [17]. Another may provide additional background in this area but does not lend itself as well to this research and is not reviewed here [34].

2.3.1 Agent Systems Characteristics. d'Inverno, et al. describe some of the qualities a multiagent system should exhibit [8]. Among them are a sense of group knowledge and intention—an indication that the system is working toward a goal or set of goals. Interaction among agents involving both communication and cooperation toward the goals should also be apparent.

Sycara provides an overview of what agent systems are and of considerations the designer must make when developing them [38]. She mentions several reasons these systems are useful:

1. They can solve more complex problems without the concern for resource limitations inherent in a single-agent system.
2. A multiagent environment can avoid the potential risk of a single point of failure.
3. Interconnection of legacy systems is possible by using agents to interface between themselves and the rest of the system. The ability to operate with distributed information or expertise sources is provided.
4. A system can more easily solve problems that look like a "society" such as calendar schedulers or automated news group management.

Some characteristics of multiagent systems include concurrency, reliability, graceful error recovery, extensibility, robustness, uncertainty handling, and the simpler maintainability that comes with modular and possibly duplicate components [38:80]. The system is also more likely to be responsive and flexible within a changing environment.

Issues and challenges inherent to agent system design are similar to problems faced in traditional parallel or distributed systems. These may include the following [38]:

1. What are the system characteristics? How will the system as a whole learn, reason, plan, and move toward goals?
2. How will the agents be organized functionally? Where will the agents reside (e.g. on a single computer or across a network)? What knowledge will an agent possess about other agents or external resources?
3. What is the best way to distribute the work load among agents? Which tasks should be allocated to each agent?
4. How should communication to and from agents be handled? How will an agent recognize the necessity to participate in a given conversation and what protocol will be used for initiating and responding to various communications? How will an agent perceive the existence of other agents?
5. How will the system be maintained? How will system resources be managed? As the environment changes how can system stability be ensured? How will the system respond to conflicting information, perceptions, or actions from different agents?

These characteristics and challenges provide a broad look at what may be addressed by agent system models.

2.3.2 Agent System Models and Specifications.

This section reviews two approaches to agent system development. These approaches are presented as they apply to formal representations that may be used in an automated synthesis tool. Challenges that may be encountered during agent system design in which interaction with humans is required are also addressed here.

2.3.2.1 Multiagent Systems Engineering (MaSE).

MaSE attempts to answer the question of "how to engineer practical multiagent systems" [6:1]. DeLoach's expressed intent is to define a methodology that supports agent system synthesis from a formal model. The languages used within this approach are the agent modeling language (AgML), which is based on graphical representations, and the agent definition language (AgDL) which is based on first order predicate logic. While OO design techniques are foundational for MaSE, this methodology modifies the semantics to capture unique agency characteristics and system cooperation behaviors. AgML and AgDL have formal definitions while OO representations do not.

Perhaps the most outstanding distinction between this and other methodologies is its handling of individual agents and components before completion of the system level design. AgML provides a graphical representation of agents similar to many OO object class diagrams and defines "high-level features of multiagent systems" [6:3] with five diagram types assisting in MaSE agent system development as examples depict in Figures 4, 5,[2] 6, 7, and 8. Four key steps to agent system design in the MaSE methodology are outlined below: domain level design, agent level design, component design, and system design.

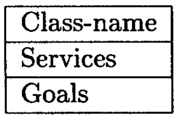


Figure 4. AgML Agent Class Model [6]

[2]Note that the two main blocks in Figure 5 match the agent classes described in Figure 4

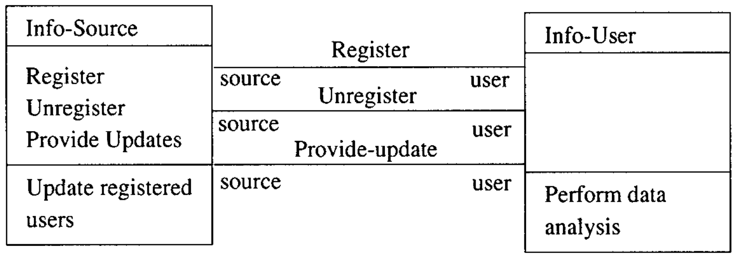


Figure 5. Conversations in AgML [6]

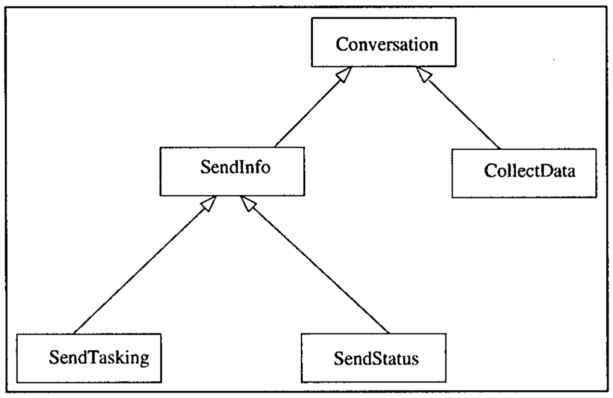


Figure 6. AgML Communication Hierarchy [6]

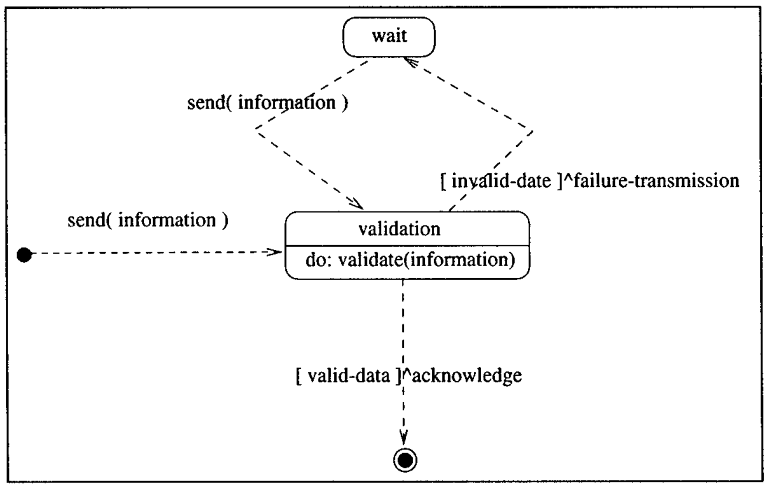

wait
send(information)
[invalid-date]^failure-transmission
send(information)
validation
do: validate(information)
[valid-data]^acknowledge

Figure 7. Communication Class in AgML [6]

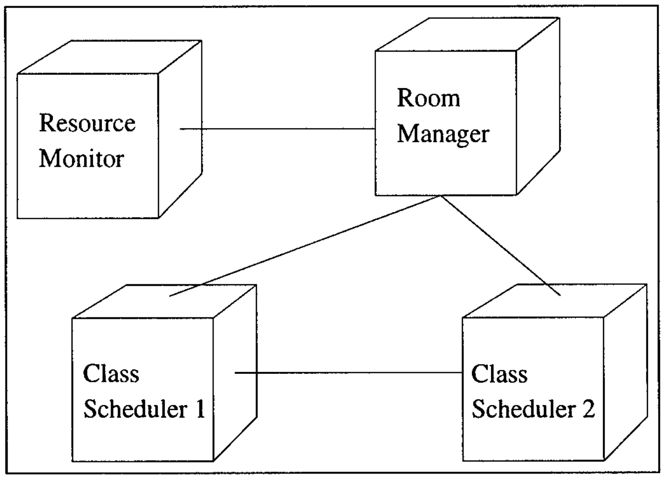


Figure 8. Deployment Diagram in AgML [6]

1. During domain level design the software engineer identifies the types of agents that will be used and develops the interactions (conversations) that will occur between the agents. Conversations are mapped out in terms of the possible sequences of messages, defined as coordination protocols. Four diagram classes are developed in this step: agent class diagrams (Figure 4), communication requirement diagrams (Figure 5), conversation class diagrams (Figure 7), and the communication hierarchy diagram (Figure 6). While these diagrams appear much like OO object diagrams, they additionally identify interfaces to the agent and the semantics of agent relationships via conversations. Such classes are the basis for reuse; the structure can be extended for specific agents while using the predefined structure.

   Two general types of conversations are presented by DeLoach [6]: CollectData and SendInfo conversation classes. These classes are shown with some subclasses in Figure 6. This diagram identifies the types of conversations that may be employed in the agent system and depicts how these types relate to each other.

2. In agent level design the engineer uses three steps that further develop the model:

   (a) Determine agent components by identifying actions in agent conversations.

   (b) Define any data structures required by the communications.

   (c) Define any data structures required for data flow between components within the agent.

   An additional agent level design objective is to reuse agent architectures whenever possible.

3. Component design requires the engineer to develop the components identified in the previous step. These designs are reused whenever possible and may include modules such as planners, search algorithms, calculation routines, or learning algorithms.

4. Within the system design step the designer selects the number and types of agents needed in the system. After selecting the agent types the designer determines exactly how many agents of each type are required, defines the physical location of each agent, specifies which conversations will be needed, and develops any other parameters that are required by the domain model. The designer graphically captures these decisions in a deployment diagram such as the example in Figure 8 to complete system design.

"[B]oth AgDL and AgML semantics are based on multi-sorted algebras" [6:7]. This statement allows one to formally verify that each conversation will end and that it will end in a particular state. The approach is claimed to be relatively easy to use and understand because it closely resembles the OO designs that many software engineers are accustomed to using.

2.3.2.2 The Z *Approach for Multiagent Systems.* Section 2.2.2.1 presented some of the basics of agent specifications using the *Z* approach of d'Inverno, et al. The methodology is extended in a later work by d'Inverno and Luck [9] in which the framework is extended to inter-agent relationships.

A multiagent system is built by merging the previously represented *Z* schemas with some additional attributes and objects. Entities group attributes together, NeutralObjects are objects with no goals, and ServerAgents are agents with at least one motivation. Any multiagent system's components can be represented by the schema in Figure 9 and interactions between entities (objects, agents, and autonomous agents) can be modeled based on this representation.

$$NeutralObject == [Object \mid goals = \{\}]$$
$$ServerAgent == [Agent \mid motivations \neq \{\}]$$

$$\begin{array}{|l}
\hline
\textit{Entity} \\
attributes : \mathbb{P}\, Attribute \\
capableof : \mathbb{P}\, Action \\
goals : \mathbb{P}\, Goal \\
motivations : \mathbb{P}\, Motivation \\
\hline
attributes \neq \{\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{MultiAgentSysComponents} \\
entities : \mathbb{P}\, Entity \\
objects : \mathbb{P}\, Object \\
agents : \mathbb{P}\, Agent \\
autoagents : \mathbb{P}\, AutoAgent \\
neutralobjects : \mathbb{P}\, NeutralObject \\
serveragents : \mathbb{P}\, ServerAgent \\
\hline
autoagents \subset agents \subset objects \\
agents = autoagents \cup serveragents \\
objects = neutralobjects \cup agents \\
\hline
\end{array}$$

Figure 9. Z Representation of an Agent System [9]

One key to a successful agent system is the ability of a specific agent to request help from other agents to accomplish goals. This is achieved by transferring a goal from one agent to another.

Another requirement for a productive autonomous agent system is cooperation among the agents. Cooperation occurs in the system when two or more autonomous agents adopt the same goal via direct engagement. Direct engagement describes the process of one entity (the client) contacting another (the server) to activate a particular goal the client needs help to accomplish. A series of direct engagements may be necessary before the goal can be reached. Figure 10 concisely and formally describes this cooperation.

The multiagent system structure shown in Figure 11 is simply the combination of several entity types. Within this framework the activities of individual agents and the

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Cooperation}} \\
goal : Goal \\
genagent : AutoAgent \\
coopagents : \mathbb{P}\ AutoAgent \\
\hline
goal \in genagent.goals \\
\forall aa : coopagents \bullet goal \in aa.goals \\
\neg\ (genagent \in coopagents) \\
coopagents \neq \{\} \\
\hline
\end{array}$$

Figure 10. Z Representation of Cooperation [9]

interactions between them are captured. Exactly how agents engage each other is the next step; d'Inverno and Luck describe the way this is accomplished with agents filling the client and server roles [9:6].

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MultiAgentSysStructure}} \\
MultiAgentSysComponents \\
SysEngChains \\
SysCoops \\
\hline
\end{array}$$

Figure 11. Z Representation of a Multiagent System [9]

2.3.2.3 Mixed-initiative agent systems. Hartrum and DeLoach [17] focus on the mixed-initiative question: how to deal with the interaction between humans and agents in a system. Their approach identifies specific ways interactions between humans and agents differ from other interactions in an agent system as well as an approach that formally captures those unique properties.

Preliminary assertions are made that agents are an abstraction beyond an OO approach and that an agent may or may not possess intelligence. In this way the design can capture the more complex intelligent agents as well as the more simply designed input/response software systems.

The mixed-initiative system uses MaSE as the system design methodology, providing the formal foundation for representations of agents and their interactions. The specifications are represented by a formal Z model and a state transition table in order to capture the structural and behavioral aspects of the system. This model can be parsed into a synthesis tool that verifies consistency and correctness, transforms the system into a formal design model, and generates OO source code.

At a basic level the mixed-initiative agent must have the ability to handle tables, graphs, and various other representations in order to interact appropriately with a human. The object is to provide the user with meaningful information that will assist his or her interaction with the system.

Other issues that are unique to mixed-initiative agents are:

1. The agent architectures must have the capability of handling the roles of both client and server. Sometimes the human has the information needed by the agent while other times the agent has or can acquire information the human desires. Sometimes the human and agent(s) must work together to discover the desired information and reach a common goal.
2. Agents must deal with ad-hoc interaction with the human. Humans frequently ask for or do things the computer (or agent in this case) has not seen before. Asynchronous inputs from other agents must also be dealt with gracefully. The agent must appropriately handle all circumstances.
3. The agent must handle both the human responses and queries made on behalf of the human.

Ultimately the agent architecture must be tailored to the particular human and the problem being solved.

Two types of conversations occur in these systems. Transaction based conversations occur when a human is queried for something and answers with a "submit" response. Incremental-based conversations require collaboration between agents and the human; all participants respond to incremental changes in the data posted by others in blackboard-style interactions.

Human/agent design issues center around several questions:

1. What queries will be made and what information will be transferred as a result of those queries?
2. What will be the responses to queries and what information will be included in the response?
3. What syntax will be used for communications?
4. What will be the form of information exchanged?
5. How will the information be presented?

The answers to these questions provide information required for automatic code generation and support the hypothesis that "design decisions can be supported by a formally based design tool that would aid the software engineer (agent designer) in specifying a specific human/agent" [17].

2.3.3 Agent System Representation Summary. MaSE uses graphical and predicate logic-based formal representations to develop agents and the systems within which they operate. The method is well-suited to automated synthesis and has a mild learning curve for those familiar with the OO paradigm and UML representations. The Z approach uses a formal language to present a well-defined structure of agents and agent systems well-suited for automated synthesis. While there is still much work to be accomplished toward the automation of agent system synthesis, the groundwork is in place for further development and integration into existing synthesis systems.

III. System Models

Before transformations can manipulate an analysis specification into a correct design, the initial analysis representation must be identified and the semantics of the model clarified. Transformations, including those using designer input, manipulate the model into an alternate representation that must also be unambiguous with predictable behavior. This chapter presents views of both the initial analysis specification and the final design specification, following the paradigm of the DOM and GOM from *AFIT*tool. A discussion of agency follows with the definition of the use of the term within the context of AWSOME and this research.

Specific attention is paid to those parts of specifications pertaining to the dynamic model with graphical representations of object classes following an object "name: field type" format in boxes with squared corners as demonstrated in Figure 12. Square brackets ("[" and "]") are used to denote sequences and curly brackets ("{" and "}") are similarly used for sets. An object instance is presented in the same way, except with rounded corners and the field type replaced by a representation of an instance of the type.

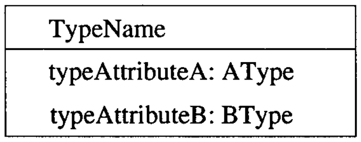


Figure 12. Example Graphical Representation of a Type

3.1 Analysis Specification

The use of *Z* schemas for the representation of OO models are built with a LaTeX *Z* format used for *AFIT*tool specifications introduced by Hartrum [16][1]; this chapter extends and adapts his models to fully capture the dynamic model within the AWSOME structure. AWSOME and *Z* representations capture identical semantics, requiring the examination of only one; the AWSOME representation is selected for this discussion to facilitate a comparison of the analysis and design models within the same framework.

[1]Specification rules outlined in this chapter were developed in cooperation with Dr. Thomas C. Hartrum, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH.

The root node of any AWSOME specification is a package (Figure 13) containing a set of declarations, a set of packages, and an identifier (Figure 14). Data type declarations and class specifications are both captured in the declarations field of a package. An identifier provides a means for naming the object, identifying the type, and providing a description.

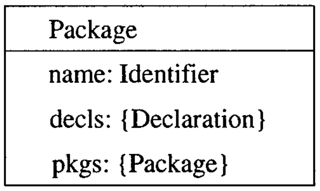


Figure 13. AWSOME Package

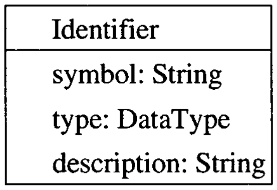


Figure 14. AWSOME Identifier

Data types are identified by name and, with the exception of the boolean type, must be created from the type classes defined by the AWSOME inheritance hierarchy in Figure 16. This figure provides a view of the types as well as the defined fields for each type with the exception of *Class* which is detailed in Figure 15. The type *Name* is used throughout this model and can be one of six subtypes defined in Figure 17.

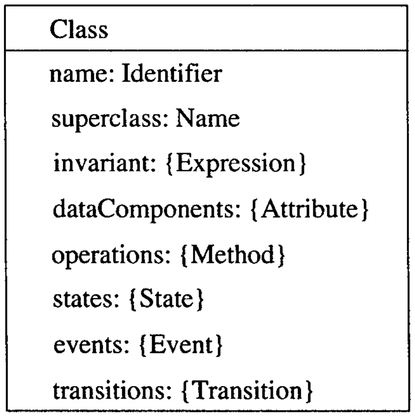


Figure 15. AWSOME Class

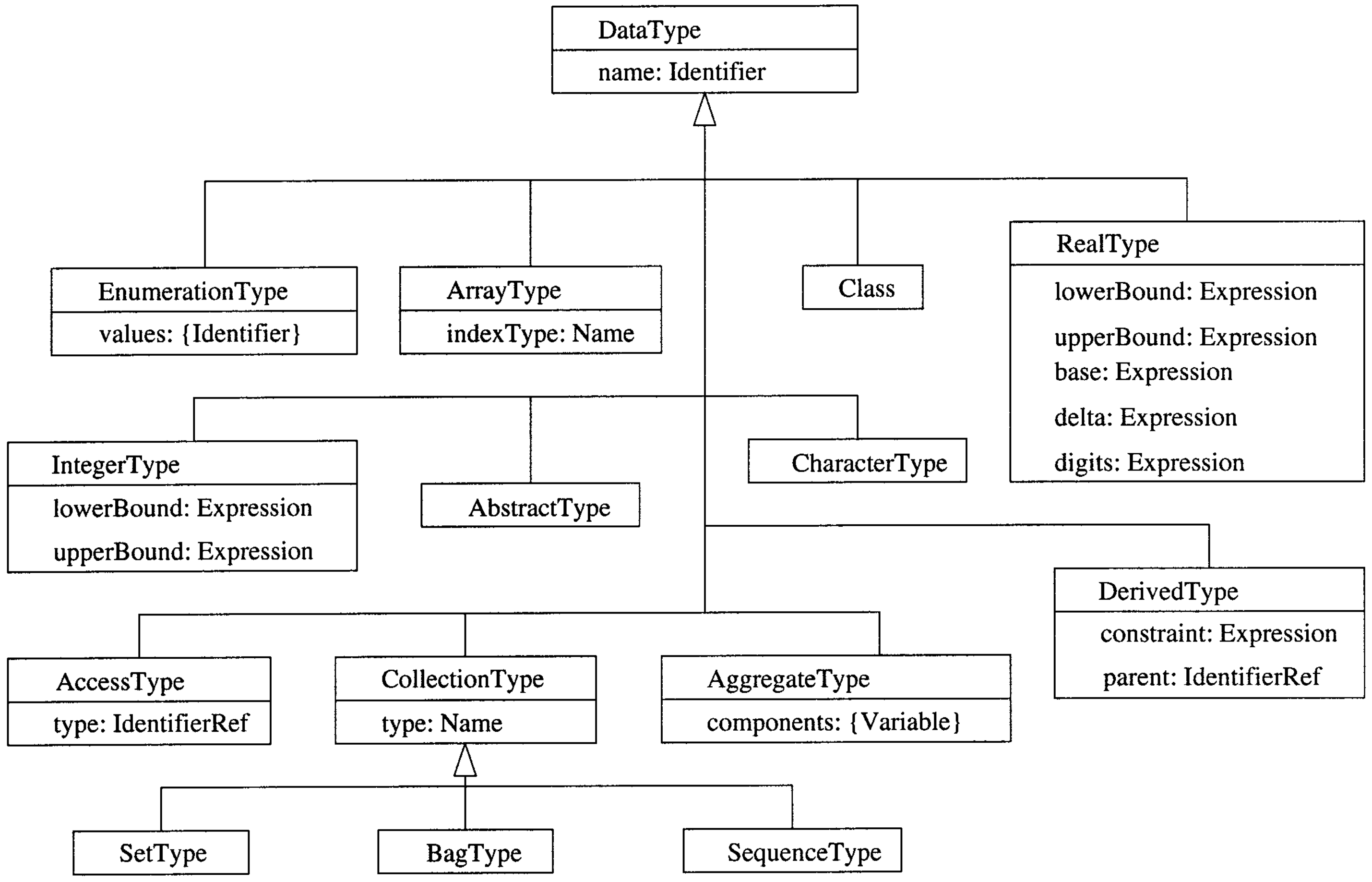


Figure 16. AWSOME Data Type and Its Inheritance

3.1.1 OO Structural Model in AWSOME. Classes are defined using the aggregation from Figure 15; the AWSOME class elements map directly to the *Z* structural, functional, and dynamic models. These classes follow the OO paradigm of inheritance, but cannot inherit from more than one class as evidenced by the singular superclass attribute. The invariant is represented in the tree as a set of boolean expressions, but may be entered or handled as a single expression by using the logical conjunction of the set. Class attributes are captured with the identification of the attributes' names, their types, and their values (Figure 18) and may be defined as either a constant or a variable (Figure 19); the "IdentifierRef" (Figure 20) is used within the attribute description and throughout AWSOME as a pointer to object identifiers. The remaining four elements comprising the class model are explored in the following two sections.

3.1.2 OO Functional Model in AWSOME. The *Z* functional model is represented in the class' operations. Each operation appears in the AWSOME tree as a method (Figure 21) that contains a subprogram (function or procedure) which, in turn, contains input and/or output parameters, pre-conditions, and post-conditions with the qualities listed below. The graphical representation of the subprogram appears in Figure 22 with formal parameters fitting the structure presented in Figure 23. Because the subprogram captures the functional qualities of a class' operations, subprograms rather than methods are discussed throughout this chapter. The rules governing analysis specification of operations follow.

Operation Specification Rule 1: The name identifies the operation and may be referenced as an action in a transition.

Operation Specification Rule 2: A set of formal parameters identify inputs and outputs.

Operation Specification Rule 3: Pre-conditions are represented as a set of expressions.

Operation Specification Rule 4: Post-conditions are represented as a set of expressions.

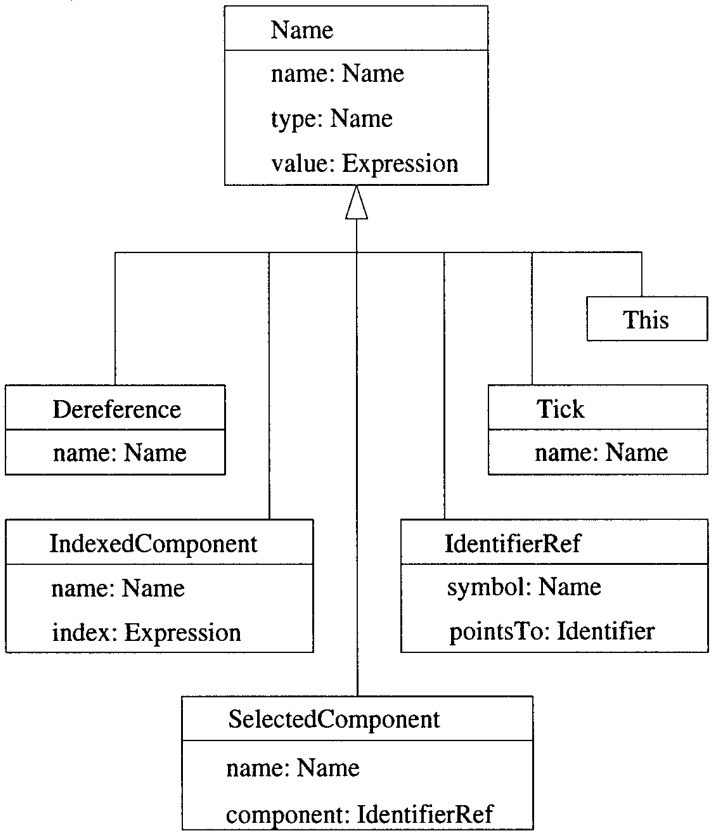


Figure 17. AWSOME Name Type and Its Inheritance

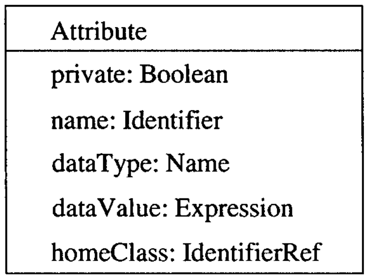


Figure 18. AWSOME Attribute

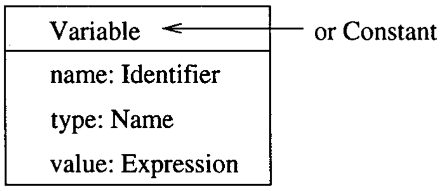


Figure 19. AWSOME Variable and Constant

IdentifierRef
symbol: String
pointsTo: Identifier

Figure 20. AWSOME IdentifierRef

Method
private: Boolean
classMethod: Boolean
methodSubprogram: Subprogram

Figure 21. AWSOME Method

3.1.3 OO Dynamic Model in AWSOME. Dynamic modeling in Z matches the AWSOME class' events, states, and transitions. The specification of the semantics for each element is critical if correctness-preserving transformations are to be developed and implemented.

3.1.3.1 AWSOME States. Each state is represented in the AWSOME tree as in Figure 24, and must follow the rules listed below. Each state is defined by a name and by a set of boolean expressions, the state invariant. The domain of variables in the invariant is the class' attributes and may or may not include a "state variable," whose sole purpose is to identify the state of the object. Substates are presented as a part of a state in Figure 24 but are not handled in this thesis.

State Specification Rule 1: A name is required for identification.

Subprogram
name: Identifier
preConditions: {Expression}
postConditions: {Expression}
formals: [Parameter]

Figure 22. AWSOME Subprogram in Analysis

Parameter
name: Identifier
type: IdentifierRef
in: Boolean
out: Boolean

Figure 23. AWSOME Parameter

State
name: Identifier
invariant: {Expression}
substates: {State}

Figure 24. AWSOME State

State Specification Rule 2: A set of boolean expressions denote an invariant for the given state; an invariant is required.

State Specification Rule 3: The invariant "$state = S_i$," making use of a "state variable" is optional.

3.1.3.2 AWSOME Events. Events define information that can be sent between objects. Each is captured in the AWSOME AST by a name, the associated parameters, and the constraints imposed on those parameters (Figure 25). If the event is received, data of the identified type(s) is understood to be received from an external source. While events specified in Z are not associated with a particular class, the AWSOME model defines all associated events within each class that may send or receive those events. Specification rules below provide guidelines that must be followed when specifying events.

Event
name: Identifier
parameters: {Parameter}
constraint: {Expression}

Figure 25. AWSOME Event

Event Specification Rule 1: A unique name for each event is required for identification.

Event Specification Rule 2: Parameters identify all data items transferred with the event.

Event Specification Rule 3: An expression specifies the constraints imposed on the parameters.

Event Specification Rule 4: Any parameter identified in the constraints must be defined within the event.

Event Specification Rule 5: Parameters and the constraint are optional.

3.1.3.3 AWSOME Transitions. Figure 26 presents the structure of an AWSOME transition with its six possible entries; two fields are required in every transition: *CurrentState* and *NextState.* These six entries specify the interactions of the class' states, events, and operations.

Transition
currentState: IdentifierRef
receiveEvent: IdentifierRef
guard: Expression
nextState: IdentifierRef
action: SubprogramCall
sendEvents: {SubprogramCall}

Figure 26. AWSOME Transition

CurrentState provides the name of the object's current state for entry into the transition while the *ReceiveEvent*, also referred to as the causal event, is the name of the event triggering the transition. The *Guard*, or guard condition, is a boolean expression pertaining to received event parameters and/or class attributes. The absence of a received event defines an automatic transition with the guard condition alone determining whether the transition is to occur. The absence of both the received event and a guard condition is a special kind of automatic transition in which the transition will occur immediately upon entry into the identified current state.

The *Action* includes the name of the operation from the functional model that must be performed prior to any *SendEvents* (identified by name) during the transition. Section 2.1 presented the use of both actions and activities. The concept of actions in that section relates directly to the *Action* in the AWSOME transition; there is no provision in AWSOME for activities.

NextState provides a postcondition for the transition, identifying the name of the object's state after the transition. It is understood that the results of receiving the event, meeting the guard condition, and performing the action within a transition guarantees the satisfaction of the next state's invariant. The one exception is that a state variable as defined in Section 3.1.3.1, if not updated within the action, must be set appropriately at the end of the transition.

Transition specifications must conform to the rules below.

Transition Specification Rule 1: The fields are defined as follows and have the inclusion requirement identified below unless specified otherwise by another rule.

1. *CurrentState* is the name of the current state (mandatory).
2. *ReceiveEvent* is the name of the causing event (optional).
3. *Guard* is a boolean expression (optional). The absence of a *Guard* is interpreted as "true."
4. *NextState* is the name of the next state (mandatory).
5. *Action* is the name of any action (optional).
6. *SendEvent* is any event that is sent to another object (optional).

Transition Specification Rule 2: There is exactly one startup transition.

1. *CurrentState* is "START."
2. *ReceiveEvent* is empty.
3. *Guard* is empty.
4. *NextState* is the name of the next state.

5. *Action* is the name of any startup action (optional).
6. *SendEvent* is any event sent to another object (optional).

Transition Specification Rule 3: There is exactly one shutdown transition.

1. *CurrentState* is the name of the current state.
2. *ReceiveEvent* is the name of the causing event (optional).
3. *Guard* is the guard condition (optional).
4. *NextState* is "END."
5. *Action* is the name of final action (optional).
6. *SendEvent* is any event sent to another object (optional).

Transition Specification Rule 4: Parameters are "connected" using matching names.

1. *Action* input parameter names match *ReceiveEvent* parameter names.
2. *Action* output parameter names match *SendEvent* parameter names.
3. There are no other *Action* input or output parameters.
4. *ReceiveEvent* parameter names may have the same names as *SendEvent* parameter names, but these represent different variables.
5. *ReceiveEvent* invariants act as *Action* pre-conditions.
6. *SendEvent* invariants act as *Action* post-conditions.
7. The *Action* defines (via post-conditions) any manipulations of:
   (a) Attributes as they relate to *ReceiveEvent* parameters.
   (b) *SendEvent* parameters as they relate to *ReceiveEvent* parameters.
   (c) *SendEvent* parameters as they relate to attributes.

Transition Specification Rule 5: The domain of variables in a *Guard* includes the class attributes and *ReceiveEvent* parameters.

3.2 Design Specification

While the AWSOME AST provides for specifications of entire software systems, this research assumes certain aspects will be implemented independently from automatically generated code. A "listen"ing process will run concurrently with an AWSOME-generated system for the purposes of receiving requests from external entities (perhaps using differing protocols), managing the requests according to specified protocols (discussed in Chapter IV), and calling the appropriate "receive" subprogram outlined below (Section 3.2.2.2). Information is sent to other entities through a "send" subprogram (Section 3.2.2.2) that performs a call to a communication protocol-specific subprogram supplied apart from the AWSOME models.

The AWSOME design model does, however, provide the means for stepping through the transitions and performing the correct *Actions* and *SendEvents* in the correct order at the correct time. A "while" loop containing selection statements provides checks for each allowable *CurrentState-ReceiveEvent-Guard* combination to ensure the proper *Actions*, *SendEvents*, and changes to the *NextState* state will commence as dictated by the specification. An assumption is that the "listen"er mentioned above can run simultaneously to this "while" loop, receiving the *ReceiveEvents* and setting the appropriate class' attributes for interpretation within this loop.

While the same AWSOME tree is used to model both the analysis and design specifications, the portions of the tree in use shifts from the dynamic model elements to the more extensive use of operations. The dynamic model is, in fact, unnecessary after the appropriate transformations are accomplished (Chapter IV). To those classes with transitions, events, and states specified, operations and attributes are added for an alternate and full representation of dynamic model semantics.

A notable distinction between analysis and design models is the altered presentation of class operations (Figure 27 as compared with Figure 22). The pre-conditions and post-conditions are no longer represented but are replaced by a sequence of statements and a set of variables and constants. Statements fall into one of the six subclasses of the AWSOME statement inheritance hierarchy shown in Figure 28.

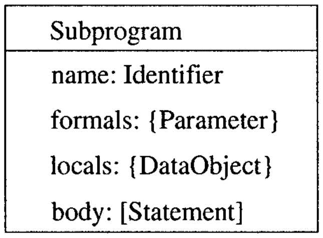


Figure 27. AWSOME Subprogram in Design

3.2.1 Additions to Class Attributes. The attributes of a class must be augmented to provide storage for the information required by additional subprograms (Section 3.2.2). Included in the design model are attributes to store the parameters from each event received or sent, as shown in Figure 29. Another attribute, identified in Figure 30, is created and added to the class attributes to track the received event name while a similar attribute named "transitionNum" is added for identification of the transition currently being handled.

3.2.2 Additions to Class Operations. Several operations are added to the class to capture its dynamic nature. A single subprogram (named "transitions") captures the behavior of all transitions while one subprogram is required to implement each event in each direction; for example an event "Event$< X >$" could be sent and/or received leading to the creation of subprograms named "sendEvent$< X >$" and/or "receiveEvent$< X >$" as required.

3.2.2.1 Subprogram "transitions" Specification. The subprogram "transitions" directly reflects the semantics of transitions specified in Section 3.1.3.3. It has no formal or local parameters since event parameters are stored in class attributes and the only information passing between elements of a transition are event parameters. Every transition maps to a selection statement modeled in Figure 31. "CurrentInvar" and "NextInvar" in this figure refer to the boolean expression from the relative state's invariant. "ThisEvent" is the place holder for a boolean expression comparing the value in "receivedEvent" to the *ReceiveEvent*'s name while "ActionCall" and each "SendCall" identify subprogramCalls to the related operation (event operations are discussed below). The last

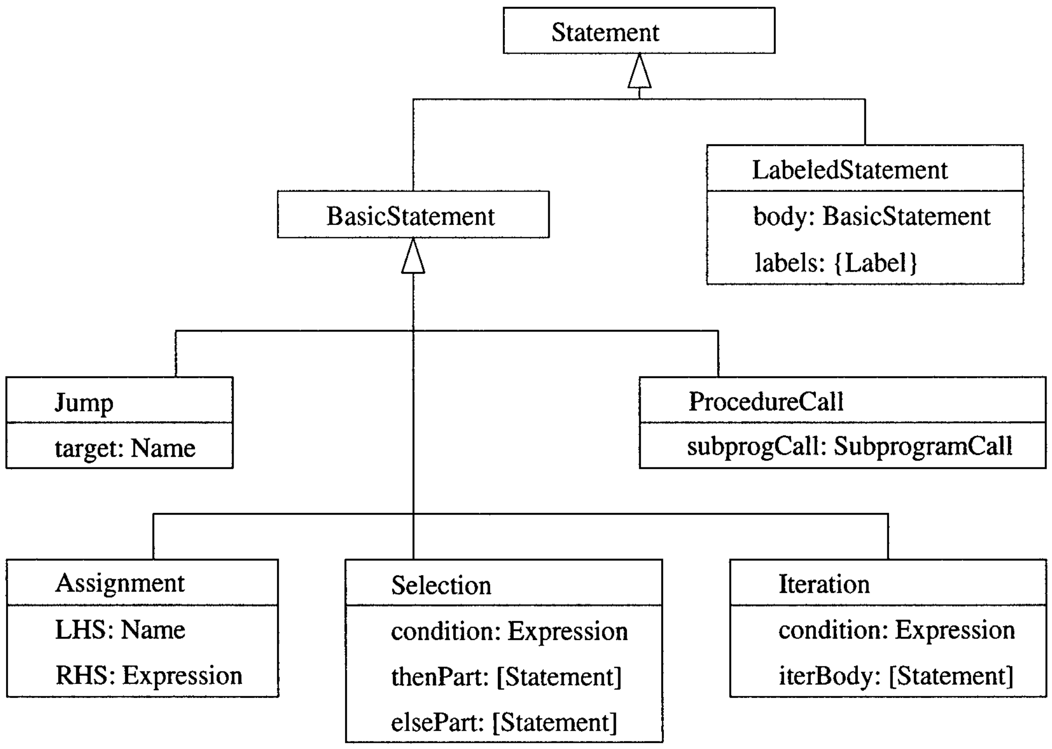


Figure 28. AWSOME Statement inheritance Hierarchy

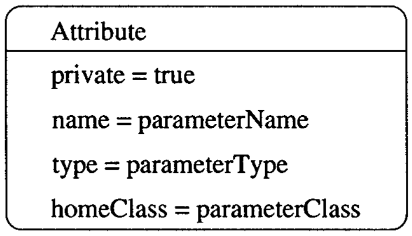


Figure 29. Example Attribute Instance Created from an Event Parameter

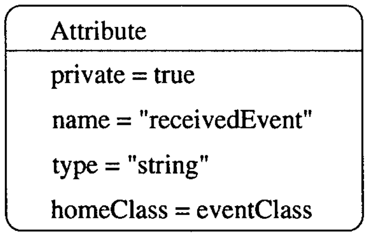


Figure 30. Example Attribute Instance for Storing an Event Name

two statements in the "thenPart" include the clearing of "receivedEvent" and the setting of the state variable if applicable.

Selection
condition = CurrentInvar AND ThisEvent AND Guard
thenPart = [transID = tX, ActionCall, Send1Call, Send2Call, ..., receivedEvent = "", transID = 0, state = NextState]
elsePart = []

Figure 31. Example Selection Statement Instance Based on a Transition

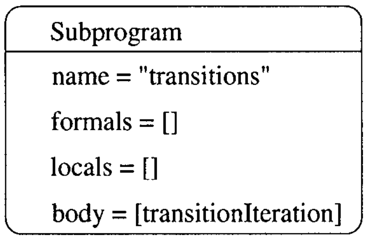
A *SendEvent* may be used in several transitions using different information passing protocols; the same event could be sent to both a human interface, providing mixed-initiative activitiy, and to another object for inter-object communications. The "transID" attribute is used for transition identification within a "sendEvent$< X >$" operation. The value of this attribute is automatically set by the related transition, each transition using a unique "transID" value. The setting and resetting of this value is reflected in Figure 31.

The set of selection statements generated from transitions (as in Figure 31) is embedded in an iteration statement's body, Figure 32, with the iteration condition matching the negation of the "END" state's invariant. This representation ensures the transitions continuously cycle as specified until the object is intentionally terminated. The complete "transitions" subprogram is represented in Figure 33; the placement of "transitionIteration" represents the embedding of Figure 32 within the body of the subprogram.

Iteration
condition = NOT (State = END)
iterBody = [transStmt1, trasnStmt2, ...]

Figure 32. Example Iteration Statement Instance for "transitionIteration"

3.2.2.2 Subprogram "sendEvent$< X >$" and "receiveEvent$< X >$" Specification. One subprogram is created for each event appearing in the *ReceiveEvent* field of a transition and another for each event referenced in the *SendEvent* field. These subpro-



Subprogram

name = "transitions"

formals = []

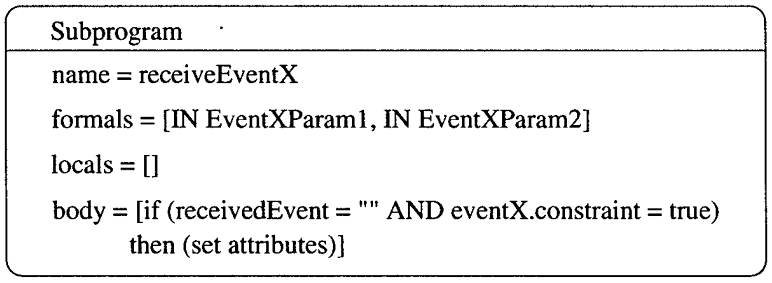
locals = []

body = [transitionIteration]

Figure 33. “transitions” Subprogram Instance Structure

grams serve as the interface between the “transitions” subprogram and external sources of, or targets for, information transfer.

For received events the subprogram is named “receiveEvent$< X >$” and has formal “IN” parameters matching the event parameters. The statement body performs several tasks, the first of which is a verification that the input does not violate the event constraints. The second task is verification that the “receivedEvent” attribute does not already contain an entry (the assumption is that the class can deal with only one *ReceiveEvent* at a time, ignoring all other external inputs). The third task is checking for validity of the event according to the transition table; if the event is not dealt with in the object’s current state, it is ignored. If the first three tasks are fulfilled, the final task is the setting of appropriate class attributes. Appropriate class attributes include those variables matching the event parameter names and the assignment of the event’s name to “receivedEvent.” An example of this subprogram is presented in Figure 34. The “receiveEvent$< X >$” subprogram must be called by “listen” discussed at the beginning of this section.



Subprogram

name = receiveEventX

formals = [IN EventXParam1, IN EventXParam2]

locals = []

body = [if (receivedEvent = "" AND eventX.constraint = true)
then (set attributes)]

Figure 34. Example “receiveEvent$< X >$” Subprogram Instance

Send events’ subprograms are similarly named “sendEvent$< X >$” but have no formal parameters (Figure 36). The body of these subprograms performs two functions, determining which transition is in execution and performing the appropriate send. To ensure the correct send is called, a sequence of selection statements is generated, one for each transition calling the send. These selection statements' "condition"s check the "transID" attribute to ensure the *Send* for the correct transition is performed. The "thenPart" consists of a single subprogramCall (Figure 35) to be identified by the designer; this subprogramCall must have as arguments the value(s) of the event's parameters.

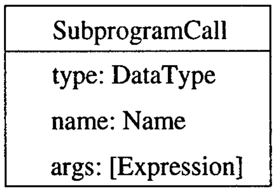


Figure 35. AWSOME SubprogramCall

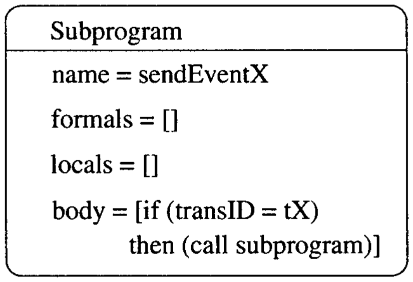


Figure 36. Example "sendEvent< X >" Subprogram Instance

3.3 Handling Agency

As identified in Chapter II, no single, widely-accepted definition of agency exists. The approach in this research assumes agents are special kinds of objects, a viewpoint shared by Luck [29] and DeLoach [6]. Along with this assumption, two concepts are adopted here as keys for defining agency. First, agents are developed differently from general objects by imposing a common messaging language for inter-agent communications [6, 11, 23, 24]. Second, agents are not merely passive entities reacting only when required by external stimuli [6, 13, 24, 29]. The former element is addressed within the context of AWSOME while the second is not; nevertheless, the proactive aspect of agency for the purposes of this work is also discussed here.

The approach to agent system design in this thesis requires generic OO objects to be defined apart from specifications of the various aspects of agency; this treatment of objects implies they are components in a larger "agent" entity. Once the specification of objects is represented in AWSOME, the designer specifies those which are to be treated as agents, the communication protocols to be implemented, and any other agent-specific details. Components to be integrated must be available for incorporation into the class models. Implied here is prior definition (within analysis models or existing code) of the items to be incorporated. Any elements not specified must match procedure or function calls that will be available upon implementation in the language desired.

The classes and types above are all contained within a single package, and are initially independent of agent-specific domain models and the associated communication protocols. The domain specifications of various systems are maintained in different packages for integration with other specifications during design.

3.3.1 Inter-Agent Communications. Agent systems and the OMT or UML models used for AWSOME research thus far manage inter-object communications very differently. While object models do not provide a means for specific instance-to-instance interactions, agent systems require that communications be directed between specific agents. Message passing in prior AWSOME models has been handled by events through which an object has no control of the received event's origin or the sent event's destination; a key assumption is that sent information is received appropriately. One recognized difference between object theory and instance theory is that when one instance sends an event, other object instances may or may not receive the event, depending on the potential receivers' current states and the acceptable receive events for those states.

When an agent sends a message it typically must know exactly which agent(s) will receive the message. Because the object models supported by AWSOME do not support this specification, either design transformations must incorporate this concept or the analysis specification model must be extended to object identification for message passing as identified below. Transmission of messages among agents can be summarized by two cases: in one-to-one message passing, information is sent from an object to another specified object; in multicast messaging, information is sent from an object to a set of specified objects. All communications can be modeled by using combinations of these two classes of message passing. For example, suppose a room manager agent maintains information about a set of non-reserved rooms and wants to know how many reserved rooms are maintained by room user agents; the manager sends a multicast message to all users requesting the respective numbers, and the users respond with a one-to-one message.

Identification of target objects for communications must be considered. Static identification occurs when an object is designed with the information required for connection to another specified object, whereas dynamic identification requires runtime detection of the required message recipients. This research expects the designer to set attributes and include procedures for the desired communications apart from the steps outlined here.

Agent messaging languages and protocols appear in many varieties as well, and can be incorporated during the transformation process. To facilitate various agent communication systems, agent and agent system models are specified without clarification of the type of message passing to be used. Conversations such as those used in MaSE can be represented by mandating sequences of class states; for simplicity this research disallows simultaneous handling of conversations. After the specification is represented in the tree and the transformation process has been initiated, the designer is required to select the desired messaging protocol (such as JAFMAS [3] or agentMom [7]) and the applicable code segments are automatically incorporated into the final product.

During system analysis and specification it may be possible to identify the number of instances and the specific communications that may occur between them. For example, a system may be desired in which two room user agents interact with a single room manager; decisions can be made during analysis regarding the handling of message passing between the agents. One such solution makes use of a message class with a naming convention implemented by an agent class (Figure 37) that can simplify message passing among several agents. Before sending a message the agent sets its *msg* values according to its own name, the names of the intended recipients, and the information the agent wishes to convey. The performative field in the *msg* is set to describe the purpose and contents of the message. A key to proper message passing is the assurance that any message sent will be received

$Message$

$sender : Name\_Type$
$receiver : P\,Name\_Type$
$performative : \mathrm{seq}\,CHAR$
$content : Object$

$Agent$

$name : Name\_Type$
$msg : Message$

Figure 37. Z Representation of the Message and Agent Classes

by the appropriate agent. While this factor could be modeled in the analysis step, it is assumed to be true for all agent communication.

This research approaches communication by accepting object designs devoid of explicit messaging protocols or the added class attributes of the Agent class in Figure 37. The designer selects existing communication elements which are then integrated into the design during transformation. Transformations and code generation, therefore, must account for proper and complete message passing while integrating design decisions.

3.3.2 Proactive Behavior. Defining the reactive or proactive quality of agents is not a simple task. Because a great variety of approaches exist for modeling independent agent activity, this research requires the system specifier either to model this aspect of agency within the framework specified (Section 3.1) or to provide appropriate existing code or design specifications. While it is assumed here that non-reactivity and the components that cause this behavior can be integrated in the initial analysis specification, this thesis focuses primarily on objects rather than on a particular form of behavior. If a preexisting component must be used, appropriate events calling those components must be included in the specification and the associated design decisions for integration must be made during the transformation to code.

3.4 Model Summary

This chapter identifies the AST structure for specifying analysis models, presenting the syntax and semantics of the OO dynamic model as used within AWSOME. Design models are also specified according to the transformations that will be applied and the structure of AWSOME representations. Analysis models are defined as consisting of data types and class representations, each class including attributes, operations, states, events, and transitions. Because the OO dynamic model is to exist only in analysis specifications, the states, events, and transitions are transformed into operations and attributes in the design specification. Design model operations are also represented with sequences of statements rather than with the pre- and post-conditions of analysis models.

Elements not specified in the analysis model can be incorporated if they match subprogram calls from "sendEvent" or to "receiveEvent" subprograms or if subprogram calls are specified in a class' operations; such elements may include the incorporation of intelligent agent characteristics. Agency is defined for this thesis' context but the handling of agent characteristics are limited to the structured messaging aspects of a class; autonomicity is assumed either to be specified within the agent/object or to be accessed via supbrogram calls.

IV. Transformations

Having defined the structures that contain the analysis and design specifications, this chapter now develops the specific transformations for moving from one to the other. *AFIT*tool DOM to DOM transformations for the dynamic model first proposed by Hartrum [15] focus on the conversion of OO dynamic model components into OO functional model components, transforming static *Z* schema representations of event definitions into dynamic *Z* schemas. His transformations provide a new "Do<EVENT_NAME>" procedure for each event in the system by merging automatic transitions with their related non-automatic transitions, creating input parameters from event parameters, and adding an implication statement for each corresponding row of the state table. He also provides an approach for collapsing automatic transitions (transitions without causal events) into non-automatic transitions.

In this research a different scheme is used for dynamic model-to-functional model transformation within AWSOME; the event, current state, and guard condition have equal weight in determining what action to perform next. The set of class operations are extended to include operations for sending and receiving events, and the activities resulting from automatic transitions are merged into other appropriate sequences of operations. Designer decisions assist the automated transformation process by identifying information such as which objects are considered agents and what communication protocols or procedure calls are required in the final code. The first transformations identified in Section 4.1 outline the process for integrating domains into the specification and creating sub-packages as desired by the designer. Section 4.2 steps through the process of developing the operations and adding attributes as required before code generation can begin. Examples in this chapter are drawn from the Room User found in Appendix A.4.

4.1 Incorporating Additional Domains

The designer may desire to integrate pre-defined domain specifications into the system such as agent communication characteristics, components of a planning system, or a reasoning engine. While this research does not implement domain integration, possible methods for accomplishing such component assimilation into the specifications addressed thus far are addressed here.

Classes and other data types in the analysis model are initially maintained in a single package, but the designer may choose to create several packages, each with specifications pertaining to a particular type of object in the system. Additional domains such as an agent class and support for a particular message passing protocol can be specified and maintained in separate packages.

Once transformations begin, the designer could be presented with the option to integrate additional domains into the existing system by selecting the package to be integrated (by name), identifying the integration method for each type (including classes) in the system, and allowing the AWSOME transformations to perform the information assimilation. Three methods for integration may be used:

1. Add the type to the package.
2. Add the type's information into another type (limited to the integration of two class types).
3. Add the type as a superclass to an object (limited to the integration of two class types).

Sometimes only certain aspects of a domain are desired, adding a fourth option:

4. Do not add the type.

In each of the integration options identified here the specifications remain in the framework presented in Section 3.1. When options 2 or 3 are selected, it may be desirable to separate classes into packages for easy identification; this option is presented to the designer. A key issue that requires attention is name conflict resolution, since the rules outlined in Chapter III must also apply to the integrated system. Another area that would require further analysis pertains to domain limitations: should a given domain be eligible for integration with any other domain? These issues are not addressed here, but are left for future work.

4.2 Dynamic Model Transformations

Specification of dynamic model semantics in Chapter III provides the formalization necessary for a complete and accurate transformation into an alternate representation. *Z* class specifications as used in *AFIT*tool are parsed into or otherwise accurately represented by the AWSOME tree prior to model manipulation. Section 4.2.1 describes the notations used in predicate calculus expressions developed below for capturing the transformations of the four constructs in Section 3.2: attributes, send event operations, receive event operations, and the "transitions" operation. The notations used in this chapter's figures follow:

1. Arrows show the source and destination of information used in a transformation.
2. Arrows originating at the top or bottom of a diagram indicate the entire object is used for transformation purposes.
3. Arrows originating at either side of a diagram indicate that the attribute corresponding to the arrow's position is used for the transformation.
4. Merging lines indicate that the multiple sources are used for transformation.
5. Diverging lines indicate that the same information is used in more than one step of the transformation process.
6. An arrow such as the following indicates that the destination consists entirely of information from the source: $\longrightarrow$
7. An arrow such as the following indicates that the destination is generated by using the source and some additional information: $\dashrightarrow$
8. An arrow such as follows indicates information from the source is used to find data for the transformation that originates in the destination: $\cdots\cdots\cdots>$
9. A comma (,) is used to separate elements in sets and sequences.

4.2.1 Formal Notations. To formalize the transformations discussed in this chapter, this section defines notations used and expresses the results of the transformations within predicate logic equations. Because "string" names are the primary ingredients for many transformations, many of the variable types below are represented as this type rather than the types presented elsewhere in this chapter (such as Identifier or IdentifierRef). Expressions are not detailed here, but are preceded below by the associated type as applicable; strings can fill the role of an expression only when they reference another object.

1: The "universe" used for equations is limited to the package in which the specification is maintained

2: An input from a source external to the package is annotated by the word INPUT

3: Equality is indicated by the symbol $=$

4: Assignment is indicated by the symbol $:=$

5: The variable type is declared by the symbol **:** followed by the name of the type

6: Sets are indicated by the pair of symbols $\{$ and $\}$

7: Sequences are indicated by the pair of symbols *[* and *]*

8: Items from sets are separated by **,** when explicitly delineated

9: Items from sequences are separated by **,** when explicitly delineated and are assumed to appear in the order required by the sequence

10: A sub-field of any composite type is indicated by use of the "dot" such as **E**.name or **O**.formals

11: String concatenation is indicated by the symbol $+$

12: The concatenation of sequences is indicated by the symbol $\frown$

13: A sequence is generated from a set by applying the "toSeq" operator; the order produced is arbitrary

14: A single expression capturing the disjunction of a set of expressions is generated by applying the "orExpOp" operator

15: An element of type Event is represented by the symbol **E** with subelements

  1. name : string
  2. parameters : *[***P***]* — (**P** defined below), note that the analysis model has a set of parameters that must be put into some sequence before transformations
  3. constraint : boolean expression

16: An element of type State is represented by the symbol **S** with subelements

  1. name : string

2. invariant : boolean expression

17: An element of type Transition is represented by the symbol **T**

1. event : string
2. current : string
3. guard : boolean expression
4. next : string
5. action : string
6. send : {string}
7. number : $\mathcal{N}$ This is a unique number assigned by the transformation system.

18: An element of type Attribute is represented by the symbol **A**

1. name : string
2. dataType : string

19: An element of type Parameter is represented by the symbol **P**

1. name : string
2. dataType : string

20: An element of type Subprogram is represented by the symbol **O**

1. name : string
2. formals : *[***P***]*
3. body : *[***ST***]*

21: An element of type Statement is represented by the symbol **ST**. This type must be implemented as a Selection, SubprogramCall, Iteration, or another Statement type. The three listed here are the only Statements specified and used below.

22: An element of type Selection is represented by the symbol **SS**

1. condition : boolean expression
2. thenPart : *[***ST***]*

23: An element of type Subprogram Call is represented by the symbol **SC**

1. name : string
2. args : *[*expression*]*

24: An element of type Iteration is represented by the symbol **I**

1. condition : expression
2. iterBody : *[***ST***]*

25: An element of type Equality Expression is represented by the symbol **EE**

1. LHS : expression
2. RHS : expression

26: An element of type Assignment Statement is represented by the symbol **AS**

1. LHS : string
2. RHS : expression

27: An element of type Class is represented by the symbol **C**

1. attrs : **{A}**
2. ops : **{O}**
3. events : **{E}**
4. states : **{S}**
5. trans : **{T}**
6. invariant : **{expression}**

4.2.2 Adding Class Attributes. Each class is augmented with a number of attributes to facilitate data transfer from the dynamic model's *Events* to *Actions* and from *Actions* to *Sends*. This transformation step is quite simple, adding attributes to the class' set of dataComponents; the names and dataTypes of these attributes match the names (prepended by "temp") and types of the events' parameters in the set of *Sends* and *Events* from class transitions as illustrated in Figure 38a.

Additional attributes are added to identify the name of a received *Event* and the current transition in progress. These attributes have the names "receivedEvent" and "transID" respectively, as shown in Figure 38b. An assumption here is that the "String" type (sequence of characters) and "Natural" type (the set of positive integers) exist, or will exist as an available type after code generation. The attributes added to the specification during transformations comply with the following two equations:

Transformation Requirement 1: $\forall e : \mathbf{E}, p : \mathbf{P}, t : \mathbf{T}, c : \mathbf{C} \mid$

$(e \in c.events \wedge t \in c.trans \wedge e \in (\{t.event\} \cup t.send) \wedge p \in e.parameters)$

$\Rightarrow (\exists a : \mathbf{A} \mid a.name = \text{``}temp\text{''} + p.name \wedge a.dataType = p.dataType \wedge a \in c.attrs)$

Transformation Requirement 2: $\forall c : \mathbf{C} \mid$

$(\exists t : \mathbf{T}, e : \mathbf{E} \mid t \in c.trans \wedge e.name = t.event) \Rightarrow$

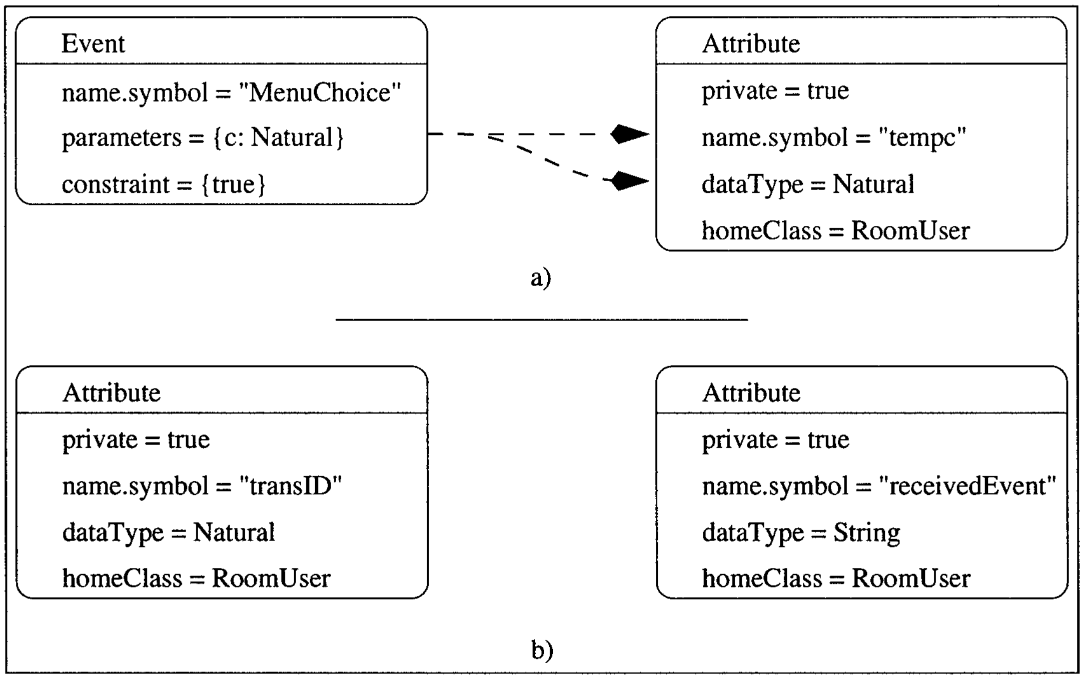


Figure 38. Creating Attributes from Events

$$(\exists a : \mathbf{A} \mid a.name = \text{“}receivedEvent\text{”} \wedge a.type = String \wedge a \in c.attrs)$$
$$\wedge\ (\exists a : \mathbf{A} \mid a.name = \text{“}transID\text{”} \wedge a.type = \mathcal{N} \wedge a \in c.attrs)$$

4.2.3 Adding Operations for Received Events. For each event *RE* that can be received by an object, an operation named "receive< *RE* >" is added to the class' set of operations. This operation is called by the class' "listen"ing component to verify the validity of received parameters and to set class attributes as required whenever the listener receives the corresponding event.

The operation "receive< *RE* >" is created with formal "IN" parameters matching the event parameters and with a body consisting of a single selection statement. The condition of the selection statement corresponds to *RE*'s constraint conjuncted with the disjunction of the invariants defining all states that can receive this event, as defined in the set of transitions; this ensures not only that valid data is received within the event but also that each event is received only if the object is in a state that can deal with it. The statement's thenPart performs the two functions identified in Section 3.2.2.2: set "receivedEvent" and other applicable class attribute values. Both of these functions are captured in an assignment statement with the RHS matching the value of a parameter and the LHS matching the name of the corresponding class attribute. The attribute "receivedEvent" is set by creating an assignment statement with the RHS holding the literal string value of the event's name while the LHS is the name of the attribute "receivedEvent." The process for creating an operation for a received event is depicted in Figure 39.

The operations added as a result of received events are represented in the following formula:

Transformation Requirement 3: $\forall e : \mathbf{E}, c : \mathbf{C} \mid$

$(e \in c.events \wedge (\exists t : \mathbf{T} \mid t \in c.trans \wedge e.name = t.event))$

$\Rightarrow (\exists o : \mathbf{O} \mid o.name = (\text{"}receive\text{"}{+}e.name) \wedge o.formals = e.parameters$

$\wedge (\exists ss : \mathbf{SS}, invars : \{expression\} \mid o.body = [ss]$

$\wedge\ invars = orExpOp\{s : \mathbf{S}, t : \mathbf{T} \mid t \in c.trans \wedge t.event = e.name$

$\wedge\ s.name = t.current \bullet s.invar\} \wedge ss.condition = (e.constraint \wedge invars)$

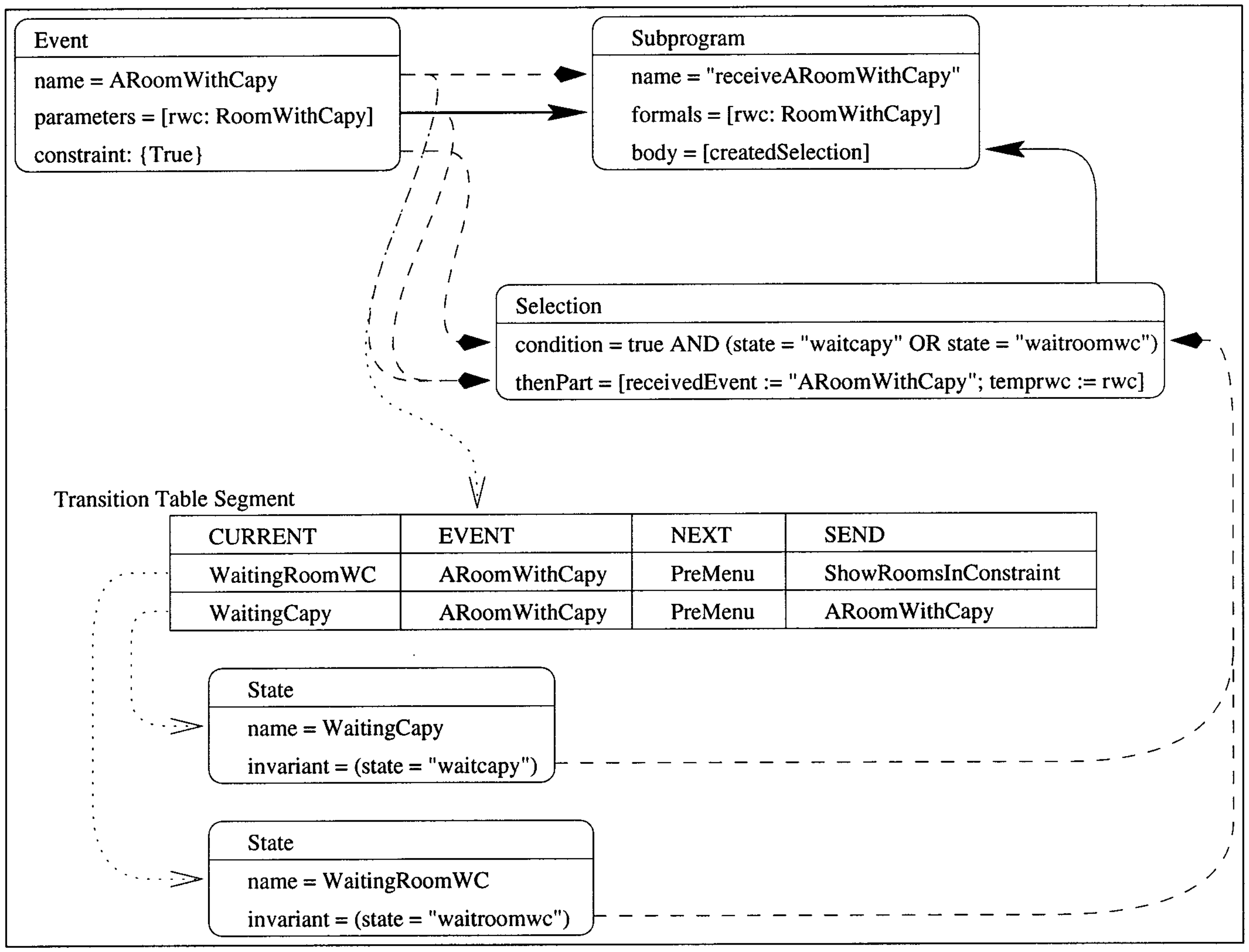
$\wedge\ ss.thenPart = [receivedEvent := e.name] \frown toSeq\{p : \mathbf{P} \mid$

$p \in e.parameters \bullet \text{"}temp\text{"}{+}p.name := p.name\})$

$\wedge\ o \in c.ops)$

4.2.4 Adding Operations for Send Events. For each event *SE* that can be sent by an object, an operation named "send< *SE* >" is added to the class' set of operations. This operation is called at the appropriate time(s) by the "transitions" operation (Section 4.2.5) and performs the function of determining which send to execute (the same event could be sent using different protocols during different transitions).

The operation "send< *SE* >" is created with no formal parameters and a body consisting of one selection statement for each transition that can send *SE*. The condition of the selection statements correspond to an equality check between the "transID" value and the possible transition. The "thenPart" is a subprogram call with arguments corresponding to *SE*'s parameters. The subprogramCall's name cannot be determined automatically because it is designated by the communication protocols selected by the designer. The name is entered by the designer when prompted by the system and then set in the design specification. The process for creating an operation for a send event is depicted in Figure 40.


Event
name = ARoomWithCapy
parameters = [rwc: RoomWithCapy]
constraint: {True}
Subprogram
name = "receiveARoomWithCapy"
formals = [rwc: RoomWithCapy]
body = [createdSelection]
Selection
condition = true AND (state = "waitcapy" OR state = "waitroomwc")
thenPart = [receivedEvent := "ARoomWithCapy"; temprwc := rwc]
Transition Table Segment
State
name = WaitingCapy
invariant = (state = "waitcapy")
State
name = WaitingRoomWC
invariant = (state = "waitroomwc")

CURRENT	EVENT	NEXT	SEND
WaitingRoomWC	ARoomWithCapy	PreMenu	ShowRoomsInConstraint
WaitingCapy	ARoomWithCapy	PreMenu	ARoomWithCapy

Figure 39. Creating Operations for Received Event Handling

The operations added as a result of send events are represented in the following equation:

Transformation Requirement 4: $\forall e : \mathbf{E}, c : \mathbf{C} \mid$

$(e \in c.events \wedge (\exists t : \mathbf{T} \mid t \in c.trans \wedge e.name \in t.send))$

$\Rightarrow (\exists o : \mathbf{O} \mid o.name = \text{“}send\text{”}{+}e.name \wedge (\forall t : \mathbf{T} \mid$

$(t \in c.trans \wedge e.name \in t.send) \Rightarrow (\exists ss : \mathbf{SS}, sc : \mathbf{SC} \mid$

$sc.name = INPUT \wedge sc.args = [p1 : \mathbf{P}, p2 : \mathbf{P} \mid$

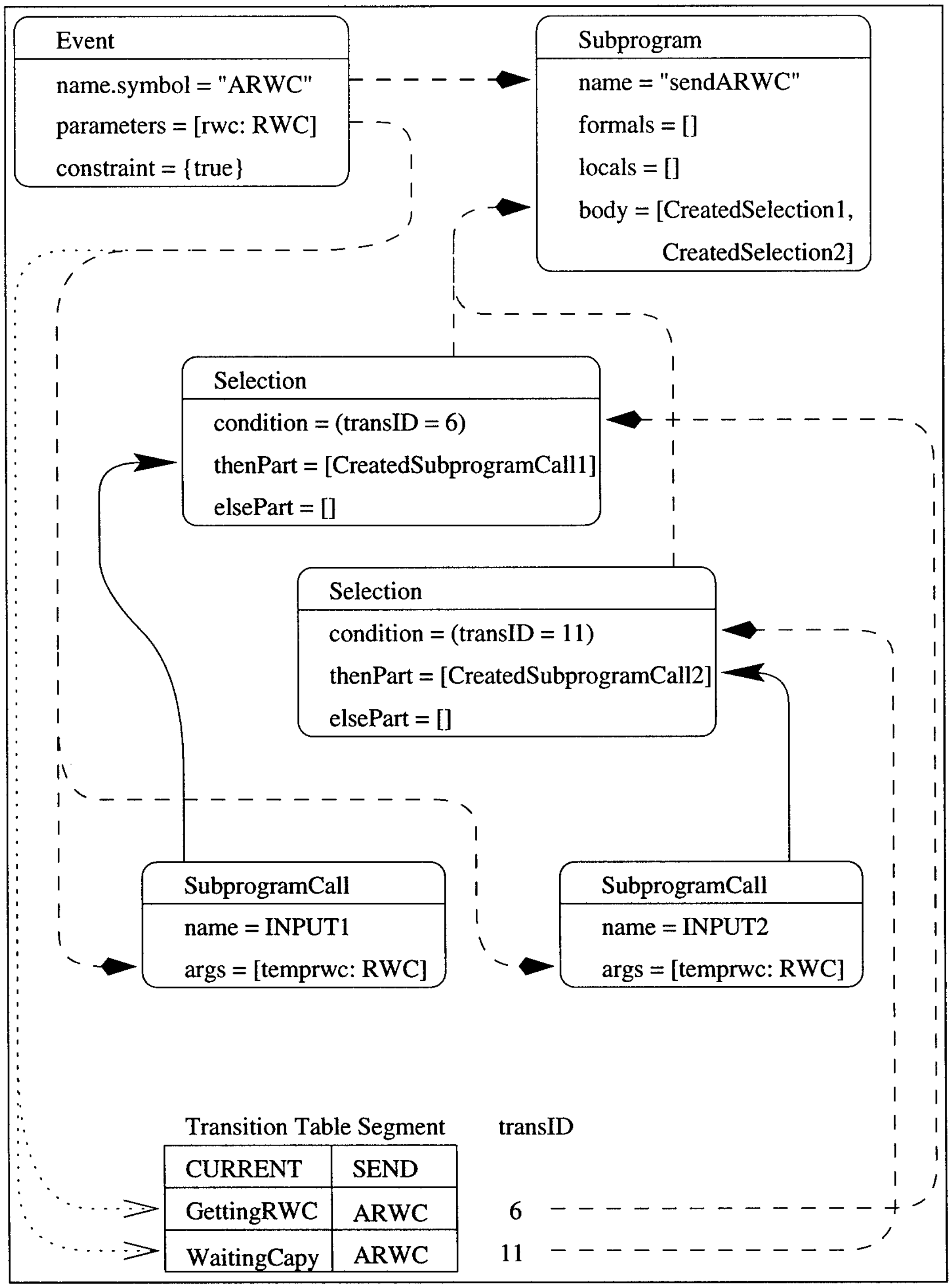
$p1 \in e.params \wedge p2.name = \text{“}temp\text{”}{+}p1.name \wedge p2.type = p1.type \bullet p2]$

$\wedge (\exists ee : \mathbf{EE} \mid ss.condition = ee \wedge ee.lhs = \text{“}transID\text{”} \wedge ee.rhs = t.number)$

$\wedge\ ss.thenPart = [sc] \wedge ss \in o.body)) \wedge o \in c.ops)$

4.2.5 Adding the "transitions" Operation. The "transitions" subprogram is the central piece of the dynamic model's functional representation. It is significantly more complex than the operations added thus far, but is created without any interaction with the designer. The single statement comprising the body of this subprogram is the iteration statement from Figure 32.

The transformation steps for creating the transition selection statements are straightforward, with three conditions and some number of subprogramCalls and assignment statements in the thenPart. The process for developing this statement is demonstrated in Figure 41. The first subprogramCall must correspond to the *Action* if one is required and any other subprogramCalls must correspond to *Sends*. In both cases there are no formal parameters because all required information is accessible within class attributes. Assignment statements include two for setting "transID" appropriately at the beginning of the thenPart and setting it to 0 at the end; an assignment for the state variable is included as necessary. There may be methods for optimizing this subprogram by merging automatic transitions' selection statements into those of non-automatic transitions; these methods are left for handling by further research.



CURRENT	SEND	transID
GettingRWC	ARWC	6
WaitingCapy	ARWC	11

Figure 40. Creating Operations for Send Event Handling

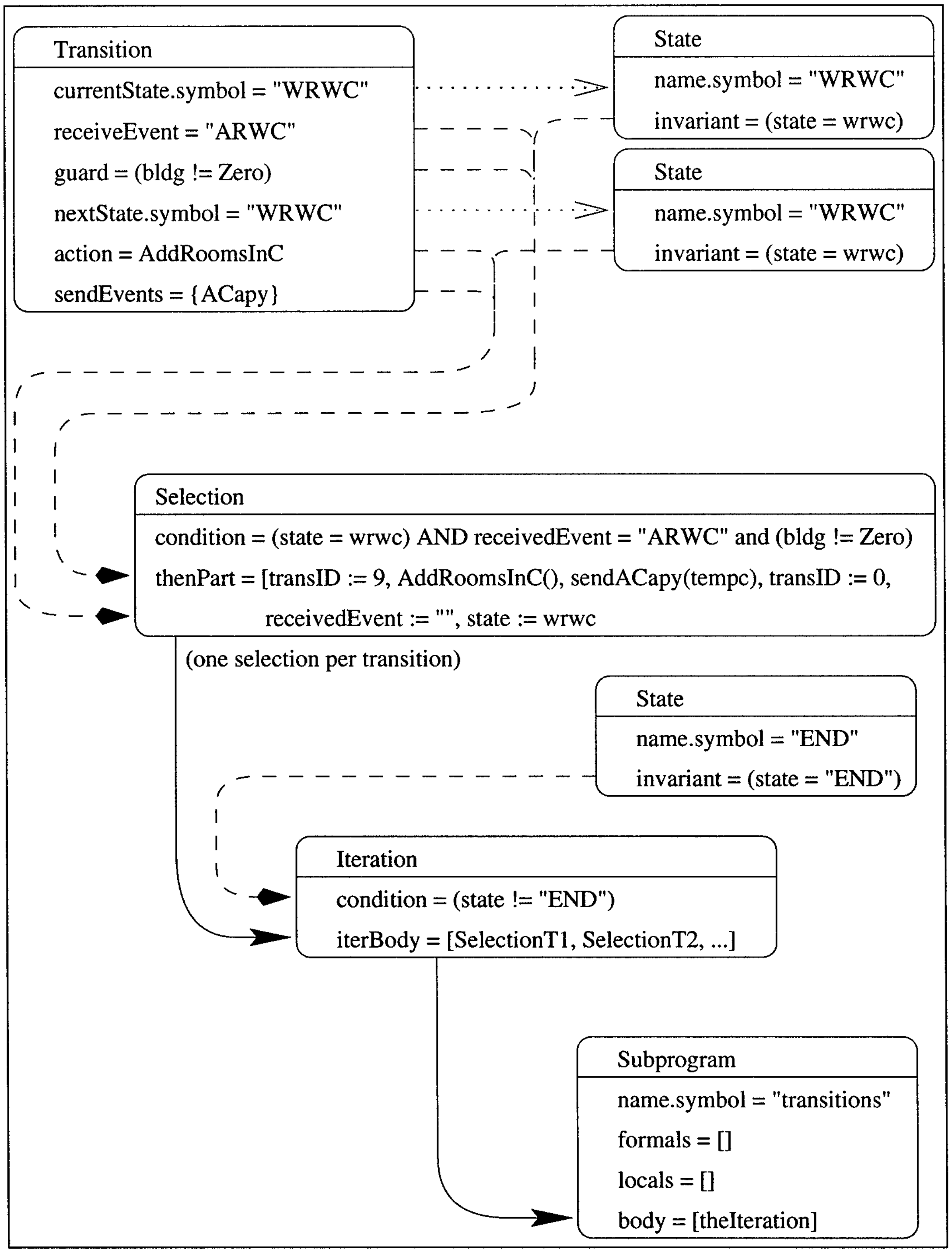

Transition
currentState.symbol = "WRWC"
receiveEvent = "ARWC"
guard = (bldg != Zero)
nextState.symbol = "WRWC"
action = AddRoomsInC
sendEvents = {ACapy}
State
name.symbol = "WRWC"
invariant = (state = wrwc)
State
name.symbol = "WRWC"
invariant = (state = wrwc)
Selection
condition = (state = wrwc) AND receivedEvent = "ARWC" and (bldg != Zero)
thenPart = [transID := 9, AddRoomsInC(), sendACapy(tempc), transID := 0,
receivedEvent := "", state := wrwc
(one selection per transition)
State
name.symbol = "END"
invariant = (state = "END")
Iteration
condition = (state != "END")
iterBody = [SelectionT1, SelectionT2, ...]
Subprogram
name.symbol = "transitions"
formals = []
locals = []
body = [theIteration]

Figure 41. Creating "transitions"

The operation added as a result of the transitions is represented in the following equation:

Transformation Requirement 5: $\forall c : \mathbf{C} \mid (\exists t : \mathbf{T} \mid t \in c.trans) \Rightarrow (\exists o : \mathbf{O}, i : \mathbf{I} \mid$

$(\exists s : state \mid (s.name = \text{"}END\text{"} \wedge s \in c.states) \Rightarrow i.condition = \neg\ (s.invariant))$

$\wedge\ (\neg\ (\exists s : state \mid (s.name = \text{"}END\text{"} \wedge s \in c.states)) \Rightarrow i.condition = true)$

$\wedge\ (\forall t : \mathbf{T} \mid t \in c.trans \Rightarrow (\exists ss : \mathbf{SS}, s1 : \mathbf{S}, s2 : \mathbf{S} \mid$

$s1.name = t.current \wedge s2.name = t.next \wedge (\exists exp : expression \mid$

$ss.condition = exp \wedge exp = (s1.condition \wedge \text{"}receivedEvent\text{"} = t.event \wedge t.guard))$

$\wedge\ \exists\, opSeq : [statement], st1 : [statement]\, st2 : [statement] \mid ss.thenPart = opSeq$

$\wedge\ \neg\ (t.action = null) \Rightarrow st1 = [(sc : \mathbf{SC}, o1 : \mathbf{O} \mid$

$o1 \in c.ops \wedge o1.name = t.action \wedge sc.name = o1.name \bullet sc)]$

$\wedge\ (t.action = null) \Rightarrow st1 = []$

$\wedge\ (\exists ee : \mathbf{EE} \mid ee.LHS = \text{"}state\text{"} \wedge ee = s2.invariant)$

$\Rightarrow (\exists as : \mathbf{AS} \mid as.LHS = \text{"}state\text{"} \wedge as.RHS = ee.RHS \wedge sc2 = [as])$

$\wedge\ (\neg\ (\exists ee : \mathbf{EE} \mid ee.LHS = \text{"}state\text{"} \wedge ee = s2.invariant)) \Rightarrow sc2 = []$

$\wedge\ ss.thenPart = [\text{"}transID\text{"} := t.number] \frown sc1 \frown toSeq\{sc1 : \mathbf{SC} \mid$

$(\forall str : String \mid str \in t.send \Rightarrow (\exists e : \mathbf{E} \mid e.name = str \wedge e \in c.events$

$\wedge\ sc1.name = str)) \bullet sc1\} \frown [\text{"}transID\text{"} := 0, \text{"}receivedEvent\text{"} := \text{""}] \frown sc2$

$\wedge\ ss \in i.thenPart))$

$\wedge\ o.body = [i] \wedge o.name = \text{"}transitions\text{"} \wedge o \in c.ops)$

4.3 Transformation Summary

This chapter provides graphical and mathematical descriptions of the transformations required for full representation of the three dynamic model elements (state, event, and transition) within class attributes and operations. The figures visually demonstrate the interactions of various objects in the system required for the creation of elements which are added to the appropriate class. Predicate calculus and a precisely defined symbology are used to present the mathematical requirements for complete and information-preserving transformations.

V. Demonstration

A sample system is specified and manipulated in this chapter through the transformations discussed in Chapter IV. Each step taken is discussed and implemented to show that the methodology outlined in previous chapters is feasible. A discussion of communication protocols precedes the description of the specifications and code generated for the proof of concept below.

5.1 Agent Communication Protocols

The systems used in this chapter include the *Z*-specified Room System in Appendix A and three event-passing protocols: an interface to Java's standard input and output, the Multi-Agent Relationships via Socket cHannels (MARSH) system, and agentMom [7]. Other protocols may be used for agent systems provided they can be conformed to the specification requirements identified in this thesis. This example requires that the selection of a communication protocol be made before incorporating agent attributes and operations into the foundation classes. The procedures in this document dictate a specific technique for the integration of information passing protocols via the "sendEvent" and "receiveEvent" subprograms. Any communication protocol that can be integrated using the methodology presented here can also be used in the generation of a software system.

5.2 Analysis Model

A room management system is used as the example system here. A room keeper tracks rooms added by various room users: it adds rooms specified by users, finds rooms for users meeting a user-specified capacity constraint, and provides the capacity of a user-specified room. The room user is the other primary object in the system, providing a person access to the information.

Because the means for parsing *Z* specifications into the AWSOME tree (implemented in Java) do not yet exist, the specifications are instantiated via a hard-coded Java program that explicitly builds the representative AWSOME AST. The creation of the AST could be handled by a graphical input program with a parser providing suitable translations between representations, by queries to a repository, or perhaps by other means; without these tools at the author's disposal, hard-coding Java code is suitable. The template used to create the Java-coded types and classes is provided in Appendix B and a sample of the room keeper's dynamic model is provided below. The steps for creating the analysis model include the initial specification of the system in *Z*, the execution of the system model code (created to reflect the *Z* specification), and the setting of appropriate elements to the associated children/parents in the AST.

A sample from the RoomKeeper's dynamic model specification in *Z* provides a starting point for the transformation process.

Waiting

RoomKeeper

$state = Waiting$

ARoomWithCapy

$rwc : RoomWithCapy$

True

Current	Event	Guard	Next	Action	Send
Waiting	ARoomWithCapy		Waiting	MakeRoom	

The *Z* specification must be captured within the AWSOME structure. This is accomplished in this chapter by hard-coding the model as presented in the Java code segment below. Presented is the same portion of the RoomKeeper as is specified above.

```
public static void addRoomKeeper(WsPackage pgm) {
    WsClass RoomKeeper = new WsClass();
    {//create the dynamic model
        {//create the states
            RoomKeeper.addState(new WsState("Waiting", "state = waiting",
                "Waiting for an input from an external source."));
        }
        {//create the events
            tempevent = new WsEvent("ARoomWithCapy");
            tempevent.setDescription("RoomWithCapy sent to or received "+
                "from a user.");
            tempevent.setWsEventParameters(new Vector());
            tempevent.addWsEventParameter(
                new WsParameter("rwc", RoomWithCapy));
            RoomKeeper.addEvent(tempevent);
        }
        //create the transitions
        {//T1
            temptrans = new WsTransition();
            //set current state
            temptrans.setWsCurrentState("Waiting");
            //set receive event
            temptrans.setWsReceiveEvent("ARoomWithCapy");
            //set next state
            temptrans.setWsNextState("Waiting");
            //set the action
            temptrans.setWsAction("MakeRoom");
            RoomKeeper.addTransition(temptrans);
        }
    }
    pgm.addWsDecl(RoomKeeper);
}
```

To enter the entire model a "main" program first calls methods for class and data type creation, performs pointer-setting responsibilities, and then calls the methods for completing transformations. The models are not stored beyond the execution of the method, but are output as simple text to the screen via the "outlineVisitor"s in the last two lines of code. The code used for these activities follows.

```
public static void main(java.lang.String[] args)
{   roomAnalysis = new WsPackage();
    roomDesign = new WsPackage();
    roomAnalysis.setWsArtName(new WsIdentifier("RoomAnalysis"));
    addTypes(roomAnalysis);
    addRoom(roomAnalysis);
    addRoomWithCapy(roomAnalysis);
    addRoomKeeper(roomAnalysis);
    addRoomUser(roomAnalysis);
    roomDesign =
       (WsPackage)roomAnalysis.acceptVisitor(new WsCopyVisitor(), null);
    roomDesign.setWsArtName(new WsIdentifier("RoomDesign"));
    //set transition pointers over all child packages
    for (Enumeration e = roomDesign.getWsPackages().elements();
        e.hasMoreElements(); )
    {
        WsPackage p = (WsPackage) e.nextElement();
        for (Enumeration ds = p.getWsDecls().elements();
            e.hasMoreElements(); )
        {
            WsDeclaration thisd = (WsDeclaration) ds.nextElement();
            if (thisd instanceof WsClass)
                ((WsClass) thisd).setTransitionPointers();
        }
    }
    //set transition pointers for all classes
    for (Enumeration e = roomDesign.getWsDecls().elements();
        e.hasMoreElements(); )
    {
        WsDeclaration d = (WsDeclaration) e.nextElement();
        if (d instanceof WsClass)
            ((WsClass) d).setTransitionPointers();
    }
    Transformations.addAttributesFromEvents(roomDesign);
    Transformations.addReceiveEventProcedures(roomDesign);
    Transformations.addSendEventProcedures(roomDesign);
    Transformations.addTransitions(roomDesign);
    roomAnalysis.acceptVisitor(new WsOutlineVisitor(), null);
    roomDesign.acceptVisitor(new WsOutlineVisitor(), null);
}
```

The “OutlineVisitor”s present a simple text representation of the specification. The following shows how the analysis model is represented within the AWSOME AST, reflecting the same specification portions as identified above.

```
.  wsDecls:Class
.  .  wsDeclName :Identifier (RoomKeeper)
.  .  wsDynamicModel :Dynamic Model
.  .  .  wsClassStates:State
.  .  .  .  wsDeclName:Identifier (Waiting)
.  .  .  .  wsStateInvariant:(state = waiting)
.  .  .  wsTransitions:Transition
.  .  .  .  wsCurrentState:Identifier Reference (Waiting)
.  .  .  .  wsReceiveEvent:Identifier Reference (ARoomWithCapy)
.  .  .  .  wsAction:Subprogram Call
.  .  .  .  .  wsSubprogCallName :Identifier Reference (MakeRoom)
.  .  .  .  wsNextState:Identifier Reference (Waiting)
.  .  .  wsClassEvents:Event
.  .  .  .  wsDeclName:Identifier (ARoomWithCapy)
.  .  .  .  wsEventParameter:Parameter
.  .  .  .  .  wsParameterName :Identifier (rwc)
.  .  .  .  .  wsParameterType :Identifier Reference (RoomWithCapy)
```

5.3 Transformation Process

Transformations can be performed in any order, although they are ordered here according to presentation in Chapter IV. These transformations are implemented according to the algorithms outlined in the following sections and are in compliance with the transformation requirements specified in Chapter IV.

1. Add the necessary attributes to classes with dynamic models, meeting Transformation Requirements 1 and 2.
2. Add the operations corresponding to received events, meeting Transformation Requirement 3.
3. Add the operations corresponding to send events, prompting for the subprogram call names as required, meeting Transformation Requirement 4.
4. Add the operation "transitions," meeting Transformation Requirement 5.
5. Output the resulting model with a text string representation.

5.3.1 Creating Necessary Attributes. The algorithm for creating attributes intended for the temporary storage of event parameters is quite simple, requiring only a few steps.

```
- Enumerate over all class types in the package (including sub-packages)
  and the class' events
  - Enumerate over the event parameters
    - Create an attribute corresponding to the event parameter
    - Name the attribute "tempEventParamName"
    - Add the attribute to the class
  - If transitions exist in the class
    - Create an attribute
      - Name the attribute "receivedEvent"
      - Set the data type to "String"
      - Add the attribute to the class
    - Create an attribute
      - Name the attribute "transID"
      - Set the data type to "Natural"
      - Add the attribute to the class
```

5.3.2 Creating Received Event Operations. Received event operation creation is much more complex than attribute generation, taking into account not only event information but also the state definitions associated through transitions.

```
- Enumerate over all class types in the package (including sub-packages)
  and the events in class transitions' received events
  - Create a procedure
    - Set the name of the procedure: "receiveEventName"
    - Create a selection statement
      - Enumerate over all current states in transitions that permit
        the event to be received
        - Create a disjunction of these states' invariants
      - Create an expression
        - Conjunct the disjunction from above with the event's constraint
      - Set the selection's condition to the resultant expression
      - Create an assignment statement
        - Left hand side = "receivedEvent"
        - Right hand side = <TheEventName>
      - Add the statement to the Selection's thenPart
      - Enumerate over the event parameters
        - Create an assignment statement
        - Left hand side = "temp<TheParameterName>"
        - Right hand side = <TheParameterName>
      - Append the statement to the Selection's thenPart
    - Set the body of the procedure as the selection
  - Add the procedure to the class' operations as a method
```

5.3.3 Creating Send Event Operations. Send event operations also require information from the other dynamic model components as well as inputs from the designer. Each event that can be sent in any given transition requires a subprogram call name that must be provided from a source external to the automated process such as, in the case of this example, through standard IO.

```
- Enumerate over all class types in the package (including sub-packages)
  and the events in class transitions' send events
  - Create a procedure
    - Set the name of the procedure: "sendEventName"
    - Initialize local integer variable TheTransitionID and set to 1
    - Enumerate over all transitions with the event in the send field
      - Create a selection statement
        - Set the condition of the selection to the equality:
          transID = TheTransitionID
        - Create a subprogram call
          - Request the name of the appropriate subprogramCall from
            the designer
          - Set the subprogram name as specified
          - Add an argument for each parameter in the event
        - Set the thenPart of the selection to the subprogram call
      - Add the selection to the procedure's body
      - Increment TheTransitionID by 1
  - Add the procedure to the class
```

5.3.4 Creating "transitions" Operation. The "transitions" operation is created directly from the list of transitions, using the properties of the events and states in the associated class. The transformations defined here yield a correct design specification; nevertheless the elimination of independent automatic transition handling would likely yield a more efficient system, an intended step left for future study and implementation.

```
- Enumerate over all class types in the package (including sub-packages)
  - Create a subprogram (procedure)
  - Initialize local integer variable TheTransitionID and set to 1
  - Name the subprogram "transitions"
  - Create an iteration
  - Set the condition to the negation of the END state invariant or "True"
    if END does not exist
  - Enumerate over the transitions (increment TransitionID for each
    transition handled)
    - Create a selection statement
    - Set the condition to the conjunction of the <Current> invariant,
      receivedEvent = <EventName>, and <Guard>
    - Add the assignment transID := TheTransitionID to the selection's
      thenPart
    - Add a procedure call for the transition's <Action> to the selection's
      thenPart (if applicable)
    - Add procedure calls for the transition's <Send> events to the
      selection's thenPart (if applicable)
    - Add the assignment receivedEvent := "" to the selection's thenPart
    - Add the assignment transID := 0 to the selection's thenPart
    - Add the selection statement to the body of the iteration
    - Increment TheTransitionID by 1
  - Add the subprogram (as a method) to the class
```

5.4 Resultant Design Model

After the transformations are executed on the analysis specification, including the entry of appropriate subprogram call names by the designer, the system outputs the design model. The class elements added to the RoomKeeper that directly result from the above specification are presented below first in the outline view as represented in the AWSOME tree and then by the Java code generated automatically by a code generation tool which has been accomplished in cooperation with Ashby [1]. Because the AWSOME design model maps directly to code the designer can hard-code any portions that are not handled by the code generator.

```
.  wsDecls:Class
.  .  wsDeclName :Identifier (RoomKeeper)
.  .  .  wsPrivate :(true)
.  .  .  wsAttributeDataObject:Variable
.  .  .  .  wsDeclName:Identifier (temprwc)
.  .  .  .  wsDataObjectType:Identifier Reference (RoomWithCapy)
.  .  .  wsAttributeHomeClass:Identifier Reference (RoomKeeper)
.  .  wsClassDataComponent :Attribute
.  .  .  wsAttributeDataObject:Variable
.  .  .  .  wsDeclName:Identifier (transID)
.  .  .  .  wsDataObjectType:Identifier Reference (Natural)
.  .  .  wsAttributeHomeClass:Identifier Reference (RoomKeeper)
.  .  wsClassDataComponent :Attribute
.  .  .  wsAttributeDataObject:Variable
.  .  .  .  wsDeclName:Identifier (receivedEvent)
.  .  .  .  wsDataObjectType:Identifier Reference (STRING)
.  .  .  wsAttributeHomeClass:Identifier Reference (RoomKeeper)
.  .  wsClassOperation :Method
.  .  .  wsMethodSubprogram :Procedure
.  .  .  .  wsDeclName:Identifier (transitions)
.  .  .  .  wsSubprogBody:Iteration
.  .  .  .  .  wsIterCondition :(True)
.  .  .  .  .  wsIterBody :Selection
.  .  .  .  .  .  wsIterBody :if
.  .  .  .  .  .  .  (wsBinExpOp1 :(True)
.  .  .  .  .  .  .   AND :
.  .  .  .  .  .  .  .  (wsBinExpOp1 :Identifier Reference (receivedEvent)
.  .  .  .  .  .  .  .   = :
```

```
.  .  .  .  .  .  .  .  .  wsBinExpOp2 :(ARoomWithCapy) ) )
.  .  .  .  .  .  .  then :Assignment
.  .  .  .  .  .  .  .  wsAssignLHS:Identifier Reference (transID)
.  .  .  .  .  .  .  .   := :
.  .  .  .  .  .  .  .  .  wsAssignRHS:Literal Integer (2)
.  .  .  .  .  .  .  then :Procedure Call
.  .  .  .  .  .  .  .  wsProcCallSupprogCall :Subprogram Call
.  .  .  .  .  .  .  .  .  wsSubprogCallName :Identifier Reference (MakeRoom)
.  .  .  .  .  .  .  then :Assignment
.  .  .  .  .  .  .  .  wsAssignLHS:Identifier Reference (transID)
.  .  .  .  .  .  .  .   := :
.  .  .  .  .  .  .  .  .  wsAssignRHS:Literal Integer (0)
.  .  wsClassOperation :Method
.  .  .  wsMethodSubprogram :Procedure
.  .  .  .  wsDeclName:Identifier (receiveARoomWithCapy)
.  .  .  .  wsSubprogFormal:Parameter
.  .  .  .  .  wsParameterName :Identifier (rwc)
.  .  .  .  .  wsParameterType :Identifier Reference (RoomWithCapy)
.  .  .  .  wsSubprogBody:Selection
.  .  .  .  .  wsSubprogBody:if
.  .  .  .  .  if:(state = waiting)
.  .  .  .  .  then :Assignment
.  .  .  .  .  .  wsAssignLHS:Identifier Reference (receivedEvent)
.  .  .  .  .  .   := :
.  .  .  .  .  .  wsAssignRHS:Identifier Reference (ARoomWithCapy)
.  .  .  .  .  then :Assignment
.  .  .  .  .  .  wsAssignLHS:Identifier Reference (temprwc)
.  .  .  .  .  .   := :
.  .  .  .  .  .  wsAssignRHS:Identifier Reference (rwc)
.  .  wsClassOperation :Method
.  .  .  wsMethodSubprogram :Procedure
.  .  .  .  wsDeclName:Identifier (sendARoomWithCapy)
.  .  .  .  wsSubprogBody:Selection
.  .  .  .  .  wsSubprogBody:if
.  .  .  .  .  .  (wsBinExpOp1 :Identifier Reference (transID)
.  .  .  .  .  .   = :
.  .  .  .  .  .  .  wsBinExpOp2 :Literal Integer (4) )
.  .  .  .  .  then :Procedure Call
.  .  .  .  .  .  wsProcCallSupprogCall :Subprogram Call
.  .  .  .  .  .  .  wsSubprogCallName :Identifier Reference (MARSHSystemSend)
.  .  .  .  .  .  .  wsSubprogCallArg :Identifier Reference (temprwc)


public RoomKeeper() {
    temprwc = null;
    transID = 0;
    receivedEvent = "";
}

public void transitions() {
    while (true) {
        if (state == start) {
            transID = 1;
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }

        if (state == waiting && receivedEvent.equals("ARoomWithCapy")) {
            transID = 2;
            MakeRoom();
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
        if (state == waiting & receivedEvent.equals("ARoom") & true) {
            transID = 3;
            GetCapy();
            sendARoomWithCapy();
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
        if (state == waiting & receivedEvent.equals("ACapy") & true) {
            transID = 4;
            FindRoom();
            sendARoomWithCapy();
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
    }
}

public void receiveARoomWithCapy(RoomWithCapy rwc) {
    if (state == waiting) {
        receivedEvent = "ARoomWithCapy";
        temprwc = rwc;
    }
}

public void sendARoomWithCapy() {
    if (transID == 3) {
        socketSystemSend.send(temprwc);
    }
    if (transID == 4) {
        socketSystemSend.send(temprwc);
    }
}
```

5.5 System Implementation

Because domain integration is not addressed within the design generation above, it is necessary to hard-code additions to or copy existing code segments into the design specification from other domains as desired. For the Java standard input and output segments, the "standardIO" class and methods are added from Appendix E.

Two different Java socket systems are used to separately demonstrate the use of the system generated above. The MARSH system is a simple Java socket-based protocol requiring the addition of several class attributes (such as "myName" and "destPort") but no further system specification; the available code is used directly by the RoomKeeper and RoomUser for communications via subprograms generated by events. The code for the RoomKeeper, RoomUser, and MARSH system are included in Appendix C. The MARSH system adds a "listen"er to objects for receiving events from other objects, and hard-codes IO to/from the user for appropriate interfacing with the "send" and "receive" events. The classes specific to the MARSH system protocol are in Appendix D and the classes used for IO to/from the user are provided in Appendix E. The IO interaction demonstrates the potential for implementing mixed-initiative programs as presented by Hartrum and DeLoach [17] within AWSOME.

The agentMom protocol requires the RoomKeeper and RoomUser extend the Agent class by the superclass relationship; conversations must be defined and implemented as outlined by DeLoach [7]. Because conversations can be viewed as moving from state to state parallel to the object/agent's state changes, a boolean variable is added to the appropriate class and used by the conversation as a signal that the next step in the conversation should or should not be taken. The RoomKeeper, RoomUser, and additional classes needed specifically for the agentMom implementation are provided in Appendix F.

"Send" procedures (such as the "socketSystemSend" referenced in the "sendARoomWithCapy" method above) are created to handle the send event requirements. The "listener" and "Send"s provide the interface between code generated from the design specification and the communication packages.

Within both systems outlined here constructors are added to provide the means of instantiating the objects and setting initial values as required. Other Java-specific methods such as the "run" method are created to begin the execution of the object's "transitions" method, and to initiate the "listener"s required by the related protocol.

5.6 Summary

This chapter provides a walk through an example beginning with the analysis specification and ending with an executable system. Transformations perform automated additions to the analysis model and integrate designer inputs to develop a design specification that can be ported to code. While segments of code must be implemented by the designer directly, the transformations of the dynamic model provide attributes and operations that can be directly correlated with executable code constructs.

VI. Conclusion, Contributions, and Recommendations

This thesis began with the focus on developing a complete specification-to-code methodology for an entire agent system. As the research progressed it became clear that code generation would likely be too grand an objective to be reasonably achieved in the short months allowed, and the focus shifted toward the insertion of agency into a generic object model. The goal shifted to consider how to generalize models for the first steps of transformation and then to integrate more complex, existing components into the model while maintaining the original intention of the analysis specification.

6.1 Contributions

Ongoing research at AFIT is focused on the development of the methodology and implementation of an automated synthesis tool. Previous work laid out the structure for capturing Z and OO models within meaningful ASTs and provided for the manipulations of certain portions of those ASTs. This research presents an aspect not previously addressed, specifically targeting the OO dynamic model, the semantics that are implied by its various elements, and its integration with communication protocols that agent systems commonly employ. The dynamic model is transformed into procedures similar to the OO functional model using designer preferences to create a system consistent with the initial specifications. Flexibility is provided not only for the implementation of mixed-initiative programs but also for the design of agent systems. An agent system can be generated from generic object specifications provided that an agent communication protocol is well-defined, the components for the agent system are specified appropriately, and existing agent system elements are in place and correctly coded.

Implementation of the methodology presented here is demonstrated by applying the automated transformations developed to basic object specifications. Automated transformations integrate designer specifications of "send" method names and create a complete design specification. After code generation the designer must provide only the interface for the particular communication protocol selected.

The transformations progress automatically with the identification of the "sending" procedures required from the designer and an automatic code generator creates Java code from the resultant design specification. While the designer must integrate additional domain information (such as agent components and communication protocols) by creating a "listener" method to handle incoming events and the "send" methods identified during the transformations, these components are designed to interface the automatically generated methods with the communication protocols desired for the system. Additional attributes may be added to the class to accommodate requirements of the desired protocols. The designer creates constructors to incorporate the "init" method (if included in the analysis specification) and to set other initial class attribute values as necessary. Finally, a "main" method (or "run" or other appropriate method) is added to enable the execution of the code.

Five transformations are identified for the complete representation of the dynamic model within class attributes and operations. Each is defined mathematically to provide the formal basis required for implementation on any capable platform. These five equations capture the most significant portions of this work, giving the formal foundation necessary for true correctness-preserving transformations.

6.2 Recommendations for Further Research

The methodology for system analysis currently requires an in-depth understanding of the formal *Z* specification language. Complicated formulae tend to be the rule for any practical system, causing many to pursue other less rigorous means for system development. Previous doctoral and master's work, along with this research, provide a well-defined semantic framework for the entire object model that now can be harnessed within a graphical based system most system designers and analysts will be more apt to use.

Integration of existing domains requires significant additional research. Dealing with naming restrictions and allowable domain integrations is not a trivial problem. The development of this area is key to a more fully automatable synthesis system.

The opportunity for optimization abounds within the "transitions" subprogram. This research treats automatic transitions exactly the same as non-automatic transitions generating the potential for much slower system execution due to unnecessary code execution. Merging the statements resulting from automatic transition transformations into the statements from non-automatic transition transformations could yield significant improvements in the execution time of the final product.

6.3 Conclusions

A great variety of agent systems exist within many different modeling schemes; communication protocols abound in equal variety. This research addresses agency from the standpoint that an agent system is composed of objects exhibiting some level of proactive behavior and communicating with a structured communication language. The proactiveness aspect is assumed either to be implemented within the analysis specification or integrated via the message passing model as send and receive events.

The task of separating unique characteristics of one agent communication system from another is daunting at best. An agent system can be designed for use with agentMom, using conversations defined in a specific manner and integrating much of the design at the analysis level. It is possible to convert generic objects into objects using agentMom communications for interaction by asserting the appropriate design decisions during AWSOME transformations. The MARSH system can be similarly harnessed for inter-object information passing, and interfacing with the human user is a straightforward process. However, there are major pieces of code (object constructors, initialization method(s), and methods for interfacing the generated code with the chosen communication protocol) that must be written aside from that potentially generated by AWSOME.

The process of generating useful executable code from requirements specifications is one step closer to a reality as the result of this research. Combining the work here with that of other researchers past, present, and future, provides an ever-increasing knowledge base from which this thesis' opening example will become a reality.

Appendix A. Generic Room System Specification in Z

This appendix provides a room system comprised of a single room keeper and a number of room users. Two types of rooms are represented, one with only a building and room number specified and another with the added capacity attribute. The room keeper keeps track of rooms added by all users, and responds to requests for the capacity of a given room and for the rooms with a capacity greater than or equal to the input value.

In response to the request for rooms of a given capacity (or greater), the room keeper returns a single room, but keeps track of all the rooms sent so that with repeated requests the keeper eventually returns all rooms meeting the criteria, returning "Zero" values when no more rooms qualify. When performing the corresponding request, the room user repeats the request until a "Zero" response is received. This sequence of events assumes the same user maintains exclusive access to the keeper during this sequence of activity.

A.1 Room

Room Structure Definition

Z Static Schema:

$STRING : \mathrm{seq}\, CHAR$

Room

$bldg : STRING$

$num : STRING$

$True$

Room

$\Delta Room$

$True$

A.2 RoomWithCapy

RoomWithCapy Structure Definition

Z Static Schema:

RoomWithCapy

$capacity : \mathcal{N}$

$Room$

$True$

RoomWithCapy

$\Delta Room$

$True$

A.3 RoomKeeper

RoomKeeper Structure Definition

Z Static Schema:

$[Room, RoomWithCapy, STRING]$

$Keeper\_States ::= start \mid waiting$

$RoomWithCapySetType : \mathrm{seq}\, RoomWithCapy$

RoomKeeper

$roomSet : RoomWithCapySetType$
$sentRoomSet : RoomWithCapySetType$
$size : \mathcal{N}$
$state : Keeper\_States$

$size = \#roomSet$

InitRoomKeeper

$\Delta RoomKeeper$

$roomSet' = \{\bot\}$
$sentRoomSet' = \{\bot\}$
$size' = 0$
$state' = start$

RoomKeeper Functional Model

Process Name: MakeRoom

MakeRoom

$\Delta RoomKeeper$
$rwc? : RoomWithCapy$

$rwc? \in roomSet'$

Process Name: GetCapy

$$
\begin{array}{|l}
\textit{GetCapy} \\
\hline
\Delta RoomKeeper \\
rwc! : RoomWithCapy \\
r? : Room \\
\hline
(\exists\, rm : RoomWithCapy \bullet \\
(rm \in roomSet \land ((rm.bldg = r?.bldg \land rm.num = r?.num) \land rwc! = rm))) \\
\lor\ (!\,\exists\, rm : RoomWithCapy \bullet \\
(rm \in roomSet \land ((rm.bldg = r?.bldg \land rm.num = r?.num) \\
\land\ (rwc!.bldg = Zero \land rwc!.num = Zero)))) \\
\hline
\end{array}
$$

Process Name: FindRoom

$$
\begin{array}{|l}
\textit{FindRoom} \\
\hline
\Delta RoomKeeper \\
rwc! : RoomWithCapy \\
c? : \mathcal{N} \\
\hline
(\exists\, r : RoomWithCapy \bullet (r \in roomSet \land ((r.bldg = rwc!.bldg \land r.num = rwc!.num) \\
\land\ r.capacity >= c?)) \land ((rwc! \notin sentRoomSet) \land (rwc! \in sentRoomSet')) \\
\lor\ (!(\exists\, r : RoomWithCapy \bullet (r.capacity = c? \land r \notin sentRoomSet)) \\
\land\ ((rwc!.bldg = Zero \land rwc!.num = Zero)) \land sentRoomSet' = \{\bot\})) \\
\hline
\end{array}
$$

RoomKeeper Dynamic Model

State Name: START

$$
\begin{array}{|l}
\textit{START} \\
\hline
RoomKeeper \\
\hline
state = start \\
\hline
\end{array}
$$

State Name: Waiting

$$
\begin{array}{|l}
\textit{Waiting} \\
\hline
RoomKeeper \\
\hline
state = Waiting \\
\hline
\end{array}
$$

Event Name: ARoomWithCapy

$$
\begin{array}{|l}
\textit{ARoomWithCapy} \\
\hline
rwc : RoomWithCapy \\
\hline
True \\
\hline
\end{array}
$$

Event Name: ARoom

ARoom

$r : Room$

$True$

Event Name: ACapy

ACapy

$c : \mathcal{N}$

$True$

State Transition Table:

Current	Event	Guard	Next	Action	Send
START			Waiting	InitRoomKeeper	
Waiting	ARoomWithCapy		Waiting	MakeRoom	
Waiting	ARoom		Waiting	GetCapy	ARoomWithCapy
Waiting	ACapy		Waiting	FindRoom	ARoomWithCapy

A.4 *RoomUser*

RoomUser Structure Definition

$[Room, RoomWithCapy, RoomWithCapySetType, STRING]$

$User\_States ::= start \mid menu \mid getcapy \mid getroom \mid getroomwc \mid waitcapy$
$\mid waitroomwc \mid premenu \mid end$

$Menu\_Choice ::= add \mid capy \mid room \mid quit$

RoomUser

$roomsInConstraint : RoomWithCapySetType$
$theCapy : \mathcal{N}$
$state : User\_States$

InitRoomUser

$\Delta RoomUser$

$state' = start$
$theCapy' = 0$
$roomsInConstraint' = \{\perp\}$

RoomUser Functional Model

Process Name: AddRoomsInConstraint

AddRoomsInConstraint

$\Delta RoomUser$
$rwc? : RoomWithCapy$
$c! : \mathcal{N}$

$rwc \in roomsInConstraint'$
$c! := theCapy$

Process Name: ClearRoomsInConstraint

ClearRoomsInConstraint

$\Delta RoomUser$

$roomsInConstraint' = \{\perp\}$

Process Name: XferRoom

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{XferRoom}} \\
\Xi RoomUser \\
r? : Room \\
r! : Room \\
\hline
r! = r?
\end{array}$$

Process Name: XferRWC

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{XferRWC}} \\
\Xi RoomUser \\
rwc? : RoomWithCapy \\
rwc! : RoomWithCapy \\
\hline
rwc! = rwc?
\end{array}$$

Process Name: SaveXferCapy

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SaveXferCapy}} \\
\Delta RoomUser \\
c? : \mathcal{N} \\
c! : \mathcal{N} \\
\hline
theCapy' = c? \\
c! = c?
\end{array}$$

Process Name: OutputRIC

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OutputRIC}} \\
\Xi RoomUser \\
ricset! : RoomWithCapySetType \\
\hline
ricset! = roomsInConstraint \\
roomsInConstraint' =
\end{array}$$

RoomUser Dynamic Model

State Name: START

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{START}} \\
RoomUser \\
\hline
state = start
\end{array}$$

State Name: END

```
┌─ START ─────────────────────────────
│ RoomUser
├──────────
│ state = end
└─────────────────────────────────────
```

State Name: TopMenu

```
┌─ TopMenu ───────────────────────────
│ RoomUser
├──────────
│ state = menu
└─────────────────────────────────────
```

State Name: GettingRoomWC

```
┌─ GettingRoomWC ─────────────────────
│ RoomUser
├──────────
│ state = getroomwc
└─────────────────────────────────────
```

State Name: GettingCapy

```
┌─ GettingCapy ───────────────────────
│ RoomUser
├──────────
│ state = getcapy
└─────────────────────────────────────
```

State Name: GettingRoom

```
┌─ GettingRoom ───────────────────────
│ RoomUser
├──────────
│ state = getroom
└─────────────────────────────────────
```

State Name: WaitingRoomWC

```
┌─ WaitingRoomWC ─────────────────────
│ RoomUser
├──────────
│ state = waitroomwc
└─────────────────────────────────────
```

State Name: WaitingCapy

```
┌─ WaitingCapy ───────────────────────
│ RoomUser
├──────────
│ state = waitcapy
└─────────────────────────────────────
```

State Name: Premenu

$Premenu$

$RoomUser$

$state = premenu$

Event Name: ShowMenu

$ShowMenu$

$true$

Event Name: MenuChoice

$MenuChoice$

$choice : MENU\_CHOICE$

$true$

Event Name: RoomWithCapyPrompt

$RoomWithCapyPrompt$

$true$

Event Name: CapyPrompt

$CapyPrompt$

$true$

Event Name: RoomPrompt

$RoomPrompt$

$true$

Event Name: ARoomWithCapy

$ARoomWithCapy$

$rwc : RoomWithCapy$

$true$

Event Name: ACapy

ACapy

$c : \mathcal{N}$

---

true

Event Name: ARoom

ARoom

$r : Room$

---

true

Event Name: ShowRoomsInConstraint

ShowRoomsInConstraint

$ricset : RoomWithCapySetType$

---

true

State Transition Table:

Current	Event	Guard	Next	Action	Send
START			TopMenu		MenuPrompt
TopMenu	MenuChoice	$choice = add$	GettingRoomWC		RoomWithCapyPrompt
TopMenu	MenuChoice	$choice = capy$	GettingCapy		CapyPrompt
TopMenu	MenuChoice	$choice = room$	GettingRoom		RoomPrompt
TopMenu	MenuChoice	$choice = quit$	END		
GettingRoomWC	ARoomWithCapy		TopMenu	XferRWC	ARoomWithCapy, MenuPrompt
GettingCapy	ACapy		WaitingRoomWC	SaveXferCapy	ACapy
GettingRoom	ARoom		WaitingCapy	XferRoom	ARoom
WaitingRoomWC	ARoomWithCapy	$rwc.bldg \neq Zero$	WaitingRoomWC	AddRoomsInConstraint	ACapy
WaitingRoomWC	ARoomWithCapy	$rwc.bldg = Zero$	PreMenu	OutputRIC	ShowRoomsInConstraint
WaitingCapy	ARoomWithCapy		PreMenu	XferRWC	ARoomWithCapy
PreMenu			TopMenu	ClearRoomsInConstraint	MenuPrompt

This template can be used to create AWSOME representations of types and objects. A sample from the code used for creating the RoomUser is provided here, along with STRING type creation. The input Package *pgm* is named "Room System" elsewhere and contains definitions for the other types and classes required in the system.

```
public static void addRoomClient(WsPackage pgm)
{
    WsSubprogram tempsubp;
    WsState tempstate;
    WsEvent tempevent;
    WsTransition temptrans;
    WsParameter tempparam;

    //Type String
    STRING = new WsSequenceType();
    STRING.setName("STRING");
    STRING.setElementTypeName("Character");
    pgm.addWsDecl(STRING);

    //Class RoomMgr
    WsClass RoomClient = new WsClass();
    RoomClient.setName("RoomClient");
    // RoomClient.setSuperclass(); //not used
    RoomClient.setWsInvariant("True");
    RoomClient.addWsClassDataComponent(
        new WsAttribute("Variable","roomsInConstraint",
        "RoomWithCapySetType"));

    //create the functional model
    {//create AddRWC
        tempsubp = new WsProcedure();
        tempsubp.setName("AddRWC");
        {//create formal parameter
            tempparam = new WsParameter();
            tempparam.setWsParameterName("rwc?");
            tempparam.setWsParameterType("RoomWithCapy");
            tempparam.setWsParameterIn(true);
            tempsubp.addWsSubprogFormal(tempparam);
        }
        tempsubp.setWsPostConditions("rwc? IN rwcset'");
        RoomClient.addWsClassOperation(new WsMethod(tempsubp));
    }

    //create the dynamic model
    {//create a state, arguments include the state name and the state
```

```
     //invariant
        RoomClient.addState(new WsState("Init", "state = initial"));

    {//create an event
        tempevent = new WsEvent("AMenuChoice");
        tempevent.addWsEventParameter(
            new WsParameter("menuchoice", "MENU_CHOICE"));
        RoomClient.addEvent(tempevent);
    }

    //create a transition
    {
        temptrans = new WsTransition();
        //set current state
        temptrans.setWsCurrentState(new WsIdentifierRef("WaitingRoomWC"));
        //set receive event
        temptrans.setWsReceiveEvent(new WsIdentifierRef("ARoomWithCapy"));
        //set guard condition
        temptrans.setWsGuard("rwc.bldg != ZERO");
        //set next state
        temptrans.setWsNextState(new WsIdentifierRef("WaitingRoomWC"));
        //set the action
        temptrans.setWsAction(new WsSubprogramCall("AddRWC"));
        //create the send event(s)
        temptrans.addWsSendEvent(new WsSubprogramCall("ACapy"));
        RoomClient.addTransition(temptrans);
    }
    pgm.addWsDecl(RoomClient);
}
```

Appendix C. Room System Implementation with MARSH System Communications

The Java code below provides the Room System discussed in this thesis, as implemented for use with the MARSH system inter-object communication protocol.

```
public interface Keeper_States {
    public final int start = 21;
    public final int waiting = 22;
}

public interface User_States {
    public final int start = 1;
    public final int menus = 2;
    public final int getcapy = 3;
    public final int getroom = 4;
    public final int getroomwc = 5;
    public final int waitcapy = 6;
    public final int waitroomwc = 7;
    public final int premenu = 8;
    public final int end = 9;
}

public interface Menu_Choice {
    public final int add = 41;
    public final int capy = 42;
    public final int room = 43;
    public final int quit = 44;
}

/**
 * The data type for containing a set of RoomWithCapy objects
 */
import java.util.*;
public class RoomWithCapySetType {
    protected Vector items;

public RoomWithCapySetType() {
    super();
    items = new Vector();
}

public void addElement(RoomWithCapy rwc) {
    if (!this.contains(rwc))
        items.addElement(rwc);
}

public boolean contains(Object o) {
    if (o instanceof RoomWithCapy)
        for (Enumeration e = items.elements(); e.hasMoreElements(); )
            if (((RoomWithCapy) e.nextElement()).equals((RoomWithCapy) o))
                return true;
    return false;
}

public RoomWithCapy elementAt(int i) {
    return (RoomWithCapy) items.elementAt(i);
}

public Enumeration elements() {
    return items.elements();
}

public Vector getElements() {
    return items;
}

public void setElements(Vector e) {
    items = e;
}

public int size() {
    return items.size();
}

public String toString() {
    String s = "";
    for (Enumeration enum = this.items.elements(); enum.hasMoreElements(); )
        s = s+"\n "+enum.nextElement();
    return s;
}
}

\**
 * The RoomUser
 */
import java.io.*;
import java.util.*;
import java.net.*;
import Support.*;
public class RoomUser implements User_States, Menu_Choice, Runnable {
    protected RoomWithCapySetType roomsInConstraint;
    protected int theCapy;
    protected int state;

    protected int tempchoice;
    protected RoomWithCapy temprwc;
    protected int tempc;
    protected Room tempr;
    protected RoomWithCapySetType tempricset;
    protected int transID;
    protected String receivedEvent;
    protected StdIO stdioSystemSend;
    protected MARSHSystemSend socketSystemSend;
    protected MARSHSystemReceive socketSystemRcv;

public RoomUser() {
    super();
    this.initRoomUser();
    this.receivedEvent = "";
    this.tempc = 0;
    this.tempr = null;
    this.temprwc = null;
    this.tempchoice = 0;
}

public void AddRoomsInConstraint() {
    RoomWithCapy rwc = temprwc;
    if (!roomsInConstraint.contains(rwc))
        roomsInConstraint.addElement(rwc);
    tempc = theCapy;
}

public void ClearRoomsInConstraint() {
    roomsInConstraint = new RoomWithCapySetType();
}

public int getState() {
    return state;
}

public int getTransID() {
    return transID;
}

public void initRoomUser() {
    state = start;
    theCapy = 0;
    roomsInConstraint = new RoomWithCapySetType();
}

public static void main(String[] args) {
    RoomUser RU = new RoomUser();
    String myname = "ru1";
    String myhost = "localhost";
    int myport = (3400);
    String targetname = "RoomKeeper";
    String targethost = "localhost";
    int targetport = 3000;
    RU.socketSystemRcv = new MARSHSystemReceive(myname, myhost, myport,
        targetname, targethost, targetport, RU);
    RU.socketSystemRcv.start();
    RU.socketSystemSend = new MARSHSystemSend(myname, myhost, myport,
        targetname, targethost, targetport, RU);
    }
    RU.stdioSystemSend = new StdIO(RU);
    RU.transitions();
}

public void OutputRIC() {
    tempricset = roomsInConstraint;
    roomsInConstraint = new RoomWithCapySetType();
}

public void receiveACapy(int c) {
    if (state == getcapy) {
        receivedEvent = "ACapy";
        tempc = c;
    }
}

public void receiveAMenuChoice(int choice) {
    if (state == menus) {
        receivedEvent = "AMenuChoice";
        tempchoice = choice;
    }
}

public void receiveARoom(Room r) {
    if (state == getroom) {
        receivedEvent = "ARoom";
        tempr = r;
    }
}

public void receiveARoomWithCapy(RoomWithCapy rwc) {
    if (state == getroomwc || state == waitroomwc || state == waitroomwc
        || state == waitcapy) {
        receivedEvent = "ARoomWithCapy";
        temprwc = rwc;
    }
}

public void run() {}

public void SaveXferCapy() {
    theCapy = tempc;
    tempc = theCapy;
}

public void sendACapy() {
    if (transID == 7) {
        socketSystemSend.send(tempc);
    }
    if (transID == 9) {
        socketSystemSend.send(tempc);
    }
}

public void sendARoom() {
    if (transID == 8) {
        socketSystemSend.send(tempr);
    }
}

public void sendARoomWithCapy() {
    if (transID == 6) {
        socketSystemSend.send(temprwc);
    }
    if (transID == 11) {
        stdioSystemSend.showRoomWithCapy(temprwc);
    }
}

public void sendCapyPrompt() {
    if (transID == 3) {
        stdioSystemSend.capyPrompt();
    }
}

public void sendMenuPrompt() {
    if (transID == 1) {
        stdioSystemSend.menuPrompt();
    }
    if (transID == 6) {
        stdioSystemSend.menuPrompt();
    }
    if (transID == 12) {
        stdioSystemSend.menuPrompt();
    }
    if (transID == 13) {
        stdioSystemSend.menuPrompt();
    }
}

public void sendRoomPrompt() {
    if (transID == 4) {
        stdioSystemSend.roomPrompt();
    }

}

public void sendRoomWithCapyPrompt() {
    if (transID == 2) {
        stdioSystemSend.roomWithCapyPrompt();
    }
}

public void sendShowRoomsInConstraint() {
    if (transID == 10) {
        stdioSystemSend.showRICSet(tempricset);
    }
}

public synchronized void setReceivedEvent(String s) {
    if (receivedEvent.equals(""))
        receivedEvent = s;
}

public void transitions() {
    while (state != end) {
        if (state == start) {
            transID = 1;
            sendMenuPrompt();
            transID = 0;
            receivedEvent = "";
            state = menus;
        }
        if (state == menus && receivedEvent.equals("AMenuChoice") &&
            tempchoice == add) {
            transID = 2;
            sendRoomWithCapyPrompt();
            transID = 0;
            receivedEvent = "";
            state = getroomwc;
        }
        if (state == menus && receivedEvent.equals("AMenuChoice") &&
            tempchoice == capy) {
            transID = 3;
            sendCapyPrompt();
            transID = 0;
            receivedEvent = "";
            state = getcapy;
        }
        if (state == menus && receivedEvent.equals("AMenuChoice") &&
            tempchoice == room) {
            transID = 4;
            sendRoomPrompt();
            transID = 0;
            receivedEvent = "";
            state = getroom;
        }
        if (state == menus && receivedEvent.equals("AMenuChoice") &&
            tempchoice == quit) {
            transID = 5;
    transID = 0;
    receivedEvent = "";
    state = end;
}
if (state == getroomwc && receivedEvent.equals("ARoomWithCapy")) {
    transID = 6;
    XferRWC();
    sendARoomWithCapy();
    sendMenuPrompt();
    transID = 0;
    receivedEvent = "";
    state = menus;
}
if (state == getcapy && receivedEvent.equals("ACapy")) {
    transID = 7;
    SaveXferCapy();
    sendACapy();
    transID = 0;
    receivedEvent = "";
    state = waitroomwc;
}
if (state == getroom && receivedEvent.equals("ARoom")) {
    transID = 8;
    XferRoom();
    sendARoom();
    transID = 0;
    receivedEvent = "";
    state = waitcapy;
}
if (state == waitroomwc && receivedEvent.equals("ARoomWithCapy") &&
    !temprwc.getBldg().equals("Zero")) {
    transID = 9;
    AddRoomsInConstraint();
    sendACapy();
    transID = 0;
    receivedEvent = "";
    state = waitroomwc;
}
if (state == waitroomwc && receivedEvent.equals("ARoomWithCapy") &&
    temprwc.getBldg().equals("Zero")) {
    transID = 10;
    OutputRIC();
    sendShowRoomsInConstraint();
    transID = 0;
    receivedEvent = "";
    state = premenu;
}
if (state == waitcapy && receivedEvent.equals("ARoomWithCapy")) {
    transID = 11;
    XferRWC();
    sendARoomWithCapy();
```

```
            transID = 0;
            receivedEvent = "";
            state = premenu;
        }
        if (state == premenu) {
            transID = 12;
            sendMenuPrompt();
            transID = 0;
            receivedEvent = "";
            state = menus;
        }
    }
}

public void XferRoom() {
    tempr = tempr;
}

public void XferRWC() {
    temprwc = temprwc;
}

public RoomWithCapy xferRWC(RoomWithCapy rwc) {
    return rwc;
}
}

\**
 * The RoomKeeper
 */
import java.util.*;
import java.net.*;
import java.io.*;
import Support.*;
public class RoomKeeper implements Keeper_States {
    protected RoomWithCapySetType roomSet;
    protected int size;
    protected RoomWithCapySetType sentRoomSet;
    protected int state;
    protected RoomWithCapy temprwc;
    protected Room tempr;
    protected int tempc;
    protected int transID;
    protected String receivedEvent;
    protected MARSHSystemSend socketSystemSend; \\added for MARSHsystem support
    protected MARSHSystemReceive socketSystemRcv;\\added for MARSHsystem support

public RoomKeeper() {
    super();
    this.initRoomKeeper();
    receivedEvent = "";
```

```
        transID = 0;
        temprwc = null;
        tempr = null;
        tempc = 0;
    }

    public void FindRoom() {
        int c = tempc;
        RoomWithCapy rwc = new RoomWithCapy();
        rwc.setBldg("Zero");
        rwc.setNum("Zero");
        for (Enumeration e = roomSet.elements(); e.hasMoreElements(); ) {
            RoomWithCapy thisrwc = (RoomWithCapy) e.nextElement();
            if (thisrwc.getCapacity() >= c && !sentRoomSet.contains(thisrwc)) {
                rwc = thisrwc;
                sentRoomSet.addElement(thisrwc);
                break;
            }
        }
        if (rwc.getBldg().equals("Zero"))
            sentRoomSet = new RoomWithCapySetType();
        temprwc = rwc;
    }

    public void GetCapy() {
        Room r = tempr;
        RoomWithCapy rwc = new RoomWithCapy();
        for (Enumeration rooms = roomSet.elements(); rooms.hasMoreElements();) {
            RoomWithCapy thisroom = (RoomWithCapy) rooms.nextElement();
            if (thisroom.getBldg().equals(r.getBldg()) &&
                thisroom.getNum().equals(r.getNum())) {
                rwc = thisroom;
                break;
            }
        }
        temprwc = rwc;
    }

    public Agent getSocketRcvTarget() {
        return socketSystemRcv.getTarget();
    }

    public MARSHSystemSend getSocketSystemSend() {
        return socketSystemSend;
    }

    public void initRoomKeeper() {
        roomSet = new RoomWithCapySetType();
        sentRoomSet = new RoomWithCapySetType();
        state = start;
        size = 0;
```

```
}

public static void main(String[] args) {
    RoomKeeper RK = new RoomKeeper();
    RK.socketSystemRcv = new MARSHSystemReceive(
        "RoomKeeper", "localhost", 3000, RK);
    RK.socketSystemRcv.start();
    RK.socketSystemSend = new MARSHSystemSend(
        "RoomKeeper", "localhost", 3000, RK);
    RK.transitions();
}

public void MakeRoom() {
    RoomWithCapy rwc = temprwc;
    boolean exists = false;
    if (rwc != null && !roomSet.contains(rwc))
        roomSet.addElement(rwc);
    size = roomSet.size();
}

public void receiveACapy(int c) {
    if (state == waiting) {
        receivedEvent = "ACapy";
        tempc = c;
    }
}

public void receiveARoom(Room r) {
    if (state == waiting) {
        receivedEvent = "ARoom";
        tempr = r;
    }
}

public void receiveARoomWithCapy(RoomWithCapy rwc) {
    if (state == waiting) {
        receivedEvent = "ARoomWithCapy";
        temprwc = rwc;
    }
}

public void sendARoomWithCapy() {
    if (transID == 3) {
        socketSystemSend.send(temprwc);
    }
    if (transID == 4) {
        socketSystemSend.send(temprwc);
    }
}

public void transitions() {
    while (true) {
        if (state == start) {
            transID = 1;
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
        if (state == waiting && receivedEvent.equals("ARoomWithCapy")) {
            transID = 2;
            MakeRoom();
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
        if (state == waiting & receivedEvent.equals("ARoom")
            & true) {
            transID = 3;
            GetCapy();
            sendARoomWithCapy();
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
        if (state == waiting & receivedEvent.equals("ACapy")
            & true) {
            transID = 4;
            FindRoom();
            sendARoomWithCapy();
            transID = 0;
            receivedEvent = "";
            state = waiting;
        }
    }
}
}
```

Appendix D. MARSH System Protocol Objects

The Java code below provides the classes used for implementing the MARSH system.

```
public interface Performatives
{
    public final int makeroom = 30;
    public final int getacapy = 31;
    public final int findroom = 32;
    public final int givecapy = 33;
    public final int giveroom = 34;
}

/**
 * The Message class used for the MARSH system.
 */
public class Message implements java.io.Serializable, Performatives
{
    protected Agent sender;
    protected Agent receiver;
    protected int performative = 0;
    protected Object content = new Object();

public Message()
{
    super();
    sender = null;
    receiver = null;
}

public Object getContent() {
    return content;
}

public int getPerformative() {
    return performative;
}

public Agent getReceiver() {
    return receiver;
}

public Object getSender() {
    return sender;
}

public void setContent(int newvalue) {
    this.content = new Integer(newvalue);
}

public void setContent(Object newValue) {
    this.content = newValue;
}

public void setPerformative(int newValue) {
    this.performative = newValue;
}

public void setReceiver(Agent newagent) {
    receiver = newagent;
}

public void setSender(Agent newagent) {
    sender = newagent;
}

public String toString() {
    return "" + sender + receiver + performative + content;
}
}

/**
 * The Agent class used for the MARSH system.
 */
import java.net.*;
public class Agent implements java.io.Serializable
{
    protected java.lang.String name;
    protected java.lang.String host;
    protected int port;
public Agent()
{
    super();
    name = "";
    host = "";
    port = 0;
}
public Agent(String s)
{
    super();
    name = s;
    port = 3000;
    host = "localhost";
}
public Agent(String s, int port)
{
    super();
    name = s;
    port = port;
    host = "localhost";
}
```

```
public Agent(String name, String host, int port)
{
    super();
    this.name = name;
    this.port = port;
    this.host = host;
}

public java.lang.String getHost() {
    return host;
}

public java.lang.String getName() {
    return name;
}

public int getPort() {
    return port;
}

public void run()
{
}

public void setHost(java.lang.String newHost) {
    host = newHost;
}

public void setName(java.lang.String newName) {
    name = newName;
}

public void setPort(int newPort) {
    port = newPort;
}

public String toString() {
    return "Name: " + name + "\nHost: " + host + "\nPort: " + port;
}
}


/**
 * The receiving "listener" for the MARSH system, tailored for the Room System.
 */
import java.net.*;
import java.io.*;
import RoomSystem.*;
import Support.*;
public class MARSHSystemReceive extends Thread {
    protected Message msg;
    protected Agent me;
    protected Agent target;
    protected RoomKeeper kparent;
    protected RoomUser uparent;
    protected ServerSocket clientConnect;
    protected Socket commsock;
    protected ObjectInputStream din;

public MARSHSystemReceive(String s, String host, int port,
    String tname, String thost, int tport, RoomUser ru) {
    super();
    msg = null;
    me = new Agent(s, host, port);
    target = new Agent(tname, thost, tport);
    uparent = ru;
    try {
        clientConnect = new ServerSocket(port);
    }
    catch(IOException e) {
        System.out.println("Problem setting up serverSocket: " + e);
    }
}

public MARSHSystemReceive(String s, String host, int port, RoomKeeper rk) {
    super();
    me = new Agent(s, host, port);
    target = new Agent();
    kparent = rk;
    try {
        clientConnect = new ServerSocket(port);
    }
    catch(IOException e) {
        System.out.println("Problem setting up serverSocket: " + e);
    }
}

public boolean closeConnection() {
    if (this.commsock != null) {
        try {
            this.commsock.close();
            System.out.println("commsock socket closed.");
            this.commsock = null;
        }
        catch (IOException e) {
            System.out.println("Trouble closing commsock socket.");
            return false;
        }
        return true;
    }
    else
        return false;
```

```
}

public void closeListener() {
    try {
        this.clientConnect.close();
        System.out.println("Listener socket closed.");
    }
    catch (IOException e) {
        System.out.println("Trouble closing Listener socket.");
    }
    this.clientConnect = null;
}

public Agent getMe() {
    return me;
}

public void getMessage() {
    Message aMsg = new Message();
    if (this.makeListener(this.getMe().getPort())) {
        // open connection with client
        if (this.openConnection()) {
            // receive message from client
            ObjectInputStream agentIn;
            try {
                din = new ObjectInputStream(this.commsock.getInputStream());
                this.receiveMsg((Message) din.readObject());
                din = null;
            }
            catch (Exception e) {
                System.out.println("Error : " + e);
            }
        }
        this.closeConnection();
    }
}

public Agent getTarget() {
    return this.target;
}

public boolean makeListener(int port) {
    boolean out = true;
    if (clientConnect == null) {
        try {
            this.clientConnect = new ServerSocket(port);
        }
        catch (IOException e) {
            out = false;
        }
    }
    return out;
}

public boolean openConnection() {
    try {
        this.commsock = this.clientConnect.accept();
        System.out.println("Got a connection on port " +
            this.commsock.getPort());
    }
    catch (IOException e) {
        return false;
    }
    return true;
}

public void receiveMsg(Message newValue) {
    msg = newValue;
    if (kparent != null) {
        if (msg.getContent() instanceof Integer)
            kparent.receiveACapy(((Integer) msg.getContent()).intValue());
        if (msg.getContent() instanceof RoomWithCapy)
            kparent.receiveARoomWithCapy((RoomWithCapy) msg.getContent());
        else if (msg.getContent() instanceof Room)
            kparent.receiveARoom((Room) msg.getContent());
        target = (Agent) msg.getSender();
        kparent.getSocketSystemSend().setTarget(target);
    }
    if (uparent != null) {
        if (msg.getContent() instanceof Integer)
            uparent.receiveACapy(((Integer) msg.getContent()).intValue());
        if (msg.getContent() instanceof RoomWithCapy)
            uparent.receiveARoomWithCapy((RoomWithCapy) msg.getContent());
        else if (msg.getContent() instanceof Room)
            uparent.receiveARoom((Room) msg.getContent());
    }
}

public void run() {
    boolean run = true;
    while (run)     {
        this.getMessage();
        if (uparent != null)
            if (uparent.getState() == uparent.end)
                run = false;
    }
}
}

/**
 * The "sender" for the MARSH system, tailored for the Room System.
 */
```

```
import java.net.*;
import java.io.*;
import RoomSystem.*;
import Support.*;
public class MARSHSystemSend {
    protected Message msg;
    protected Agent me;
    protected Agent target;
    protected RoomKeeper kparent;
    protected RoomUser uparent;
    protected Socket commsock;
    protected ObjectOutputStream dout;

public MARSHSystemSend(String s, String host, int port,
    String tname, String thost, int tport, RoomUser ru) {
    super();
    msg = null;
    me = new Agent(s, host, port);
    target = new Agent(tname, thost, tport);
    uparent = ru;
}
public MARSHSystemSend(String s, String host, int port, RoomKeeper rk) {
    super();
    me = new Agent(s, host, port);
    target = new Agent();
    kparent = rk;
}

public MARSHSystemSend(String s, String host, int port, RoomUser ru) {
    super();
    me = new Agent(s, host, port);
    target = new Agent();
    uparent = ru;
}

public boolean closeConnection() {
    if (this.commsock != null) {
        try {
            this.commsock.close();
            System.out.println("commsock socket closed.");
            this.commsock = null;
        }
        catch (IOException e) {
            System.out.println("Trouble closing commsock socket.");
            return false;
        }
        return true;
    }
    else
        return false;
}
```

```
public Agent getMe() {
    return me;
}

public String getTargetHost() {
    return msg.getReceiver().getHost();
}

public int getTargetPort() {
    return msg.getReceiver().getPort();
}

public boolean makeConnection() {
    try {
        this.commsock = new Socket(this.getTargetHost(), this.getTargetPort());
        System.out.println("Got a socket connection...");
    }
    catch (Exception e) {
        System.out.println("Error: " + e);
        return false;
    }
    return true;
}

public void send(int i) {
    msg = new Message();
    msg.setContent(i);
    msg.setSender(me);
    msg.setReceiver(target);
    this.sendMessage(msg);
}

public void send(Object o) {
    msg = new Message();
    msg.setContent(o);
    msg.setSender(me);
    msg.setReceiver(target);
    if (kparent != null)
        msg.setReceiver(kparent.getSocketRcvTarget());
    this.sendMessage(msg);
}

public boolean sendMessage(Message msg) {
    if ((commsock == null && this.makeConnection()) || commsock != null) {
        System.out.println("Connection made.");
        try {
            dout = new ObjectOutputStream(this.commsock.getOutputStream());
            dout.writeObject(msg);
            dout = null;
        }
        catch (IOException e) {
            System.out.println("Problem in sendMgrMessage: " + e);
            return false;
        }
    }
    else {
        System.out.println("Connection not made");
        this.closeConnection();
        return false;
    }
    this.closeConnection();
    return true;
}

public void setTarget(Agent a) {
    target = a;
}
}
```

```
import java.io.*;
\**
 * Thread for handling user inputs of a Capy (interfaces with RoomUser)
 */
import java.io.*;
public class InCapy extends Thread {
    roomsystem.RoomUser parent;

public InCapy(roomsystem.RoomUser RU) {
    super();
    parent = RU;
}

public void capyEntry() {
    capyPrompt();
    String s = "";
    boolean b = false;
    int i = 0;
    while (!b) {
        try {
            s = StdIO.stdioStaticEntry();
            if (s.equals(""))
                s = StdIO.stdioStaticEntry();
            i = (Integer.parseInt(s));
            if (i < 1)
                throw new IOException();
            b = true;
        }
        catch (Exception e) {
            parent.stdioSystemSend.stdioStaticPrint("Enter a valid capacity: ");
            b = false;
            s = "";
            capyPrompt();
        }
    }
    parent.receiveACapy(i);
}

public void capyPrompt() {
    parent.stdioSystemSend.stdioStaticPrint("Enter desired capacity: ");
}

public void run() {
    capyEntry();
}
}

\**
 * Thread for handling user inputs of a Room (interfaces with RoomUser)
 */
```

```
public class InRoom extends Thread {
    roomsystem.RoomUser parent;

public InRoom(roomsystem.RoomUser RU) {
    super();
    parent = RU;
}

public void roomEntry() {
    roomPrompt();
    String s = "";
    boolean b = false;
    roomsystem.Room r = new roomsystem.Room();
    while (!b) {
        try {
            s = StdIO.stdioStaticEntry();
            int m = 0;
            for (int i = 1; i <= s.length(); i++) {
                if (m == 0 && s.charAt(i) == ',')
                    m = i;
            }
            String t, u;
            t = s.substring(0, m);
            u = s.substring(m + 1, s.length());
            r.setBldg(t.trim());
            r.setNum(u.trim());
            if (m == s.length())
                b = false;
            else b = true;
        }
        catch (IOException e) {
            parent.stdioSystemSend.stdioStaticPrint("Enter a valid Room");
            b = false;
            s = "";
            roomPrompt();
        }
    }
    parent.receiveARoom(r);
}

public void roomPrompt() {
    parent.stdioSystemSend.stdioStaticPrint(
        "Enter desired room in the format <bldg, num>: ");
}

public void run() {
    roomEntry();
}
}

/**
```

```
 * Thread for handling user inputs of a RoomWithCapy (interfaces with RoomUser)
 */
import java.io.*;
public class InRWC extends Thread {
    roomsystem.RoomUser parent;

public InRWC(roomsystem.RoomUser RU) {
    super();
    parent = RU;
}

public void run() {
    rwcEntry();
}

public void rwcEntry() {
    rwcPrompt();
    String s = "";
    boolean b = false;
    roomsystem.RoomWithCapy rwc = new roomsystem.RoomWithCapy();
    while (!b) {
        try {
            s = StdIO.stdioStaticEntry();
            int m = 0;
            int n = 0;
            for (int i = 0; i < s.length(); i++) {
                if (m == 0 && s.charAt(i) == ',')
                    m = i;
                if (m > 0 && s.charAt(i) == ',')
                    n = i;
            }
            String t = "";
            String u = "";
            String v = "";
            t = s.substring(0, m);
            u = s.substring(m + 1, n);
            v = s.substring(n + 1, s.length());
            rwc.setBldg(t.trim());
            rwc.setNum(u.trim());
            rwc.setCapacity(Integer.parseInt(v.trim()));
            b = true;
        }
        catch (Exception e) {
            parent.stdioSystemSend.stdioStaticPrint(
                "Enter a valid RoomWithCapy");
            b = false;
            s = "";
            rwcPrompt();
        }
    }
    parent.receiveARoomWithCapy(rwc);
```

```
}

public void rwcPrompt() {
    parent.stdioSystemSend.stdioStaticPrint(
        "Enter desired roomWithCapy in the format <bldg, num, capy>: ");
}
}

\**
 * Thread for handling user inputs of a MenuChoice (interfaces with RoomUser)
 */
import java.io.*;
public class InMenu extends Thread {
    roomsystem.RoomUser parent;

public InMenu(roomsystem.RoomUser RU) {
    super();
    parent = RU;
}

public void menuEntry() {
    menuPrompt();
    String s;
    int i = 0;
    boolean validEntry = false;
    while (!validEntry) {
        try {
            s = StdIO.stdioStaticEntry();
            i = Integer.parseInt(s);
            validEntry = true;
            if (i < 1 || i > 4)
                throw new IOException();
        }
        catch (Exception e) {
            parent.stdioSystemSend.stdioStaticPrint(
                "Please enter a valid choice <1..4>! \n");
            menuPrompt();
            validEntry = false;
            s = "";
        }
    }
    parent.receiveAMenuChoice(40 + i);
}
public void menuPrompt() {
    parent.stdioSystemSend.stdioStaticPrint("Input your selection:\n\t"+
        "1 to add a room, \n\t2 to find a room with a specific capacity, "+
        "\n\t3 to get the capacity of a room, or\n\t4 to quit");
}

public void run() {
    menuEntry();
```

```
}
}

\**
 * Thread for handling sends to the user (interfaces with RoomUser)
 */
import java.io.*;
import java.util.*;
public class StdIO {
    roomsystem.RoomUser parent;
    int state;

public StdIO(roomsystem.RoomUser RU) {
    super();
    parent = RU;
}

public void capyPrompt() {
    InCapy IC = new InCapy(parent);
    IC.start();
}

public void menuPrompt() {
    InMenu IM = new InMenu(parent);
    IM.start();
}

public void roomPrompt() {
    InRoom R = new InRoom(parent);
    R.start();
}

public void roomWithCapyPrompt() {
    InRWC RWC = new InRWC(parent);
    RWC.start();
}

public void showRICSet(roomsystem.RoomWithCapySetType ricset) {
    if (ricset != null) {
        stdioPrint("\n\nSet of rooms meeting criteria:\n");
        for (int i = 0; i < ricset.size(); i++)
            stdioPrint("\t"+(roomsystem.RoomWithCapy) ricset.elementAt(i));
        stdioPrint("\n");
    }
}

public void showRoomWithCapy(roomsystem.RoomWithCapy rwc) {
    stdioPrint("\nThe room (including its capacity) is: "+rwc);
}

public String stdioEntry() throws IOException {
        String s = "";
        BufferedReader entry = new BufferedReader(new InputStreamReader(System.in));
        s = entry.readLine();
        return s;
    }

    public void stdioPrint(String s) {
        System.out.println(s);
    }

    public static String stdioStaticEntry() throws IOException {
        String s = "";
        BufferedReader entry = new BufferedReader(new InputStreamReader(System.in));
        s = entry.readLine();
        return s;
    }

    public static void stdioStaticPrint(String s) {
        System.out.println(s);
    }
    }
```

The Java code below contains the classes, methods, and attributes that differ from the MARSH system code presented in Appendix D. The agent.mom package contains the agentMom classes noted as required by DeLoach [7].

```
/**
 * The RoomUser's side of a conversation begun by the "ACapy" event
 */
import java.net.*;
import java.io.*;
import agent.mom.*;
public class CAPY_INTERFACE extends Conversation {
    Agent parent;
    int thiscapy;

public CAPY_INTERFACE(RoomUser a, int capy) {
    super(a, a.name, a.port);
    parent = a;
    thiscapy = capy;
}

public void run() {
    Message m = new Message();
    boolean notDone = true;
    System.out.println("Starting Capy conversation.");
    try {
        connection = new Socket(connectionHost, connectionPort);
        output = new ObjectOutputStream(connection.getOutputStream());
        output.flush();
        input = new ObjectInputStream(connection.getInputStream());
        while (notDone) {
            m.performative = "getroom";
            m.content = new Integer(thiscapy);
            sendMessage(m, output);
            m = readMessage(input);
            ((RoomUser) parent).receiveARoomWithCapy(
                (RoomWithCapy) m.getContent());
            if (((RoomWithCapy) m.getContent()).getBldg().equals("Zero"))
                notDone = false;
            else while (!((RoomUser) parent).getReady()){}
        }
        input.close();
        output.close();
        connection.close();
    }
    catch (Exception e) {
        System.out.println("Error: " + e);
    }
}
}

\**
 * The RoomUser's side of a conversation begun by the "ARoom" event
 */
import java.net.*;
import java.io.*;
import agent.mom.*;
public class ROOM_INTERFACE extends Conversation {
    Agent parent; // override parent
    Room thisr;

public ROOM_INTERFACE(RoomUser a, Room r) {
    super(a, "localhost", 3300);
    parent = a;
    thisr = r;
}

public void run() {
    Message m = new Message();
    System.out.println("Starting Room conversation.");
    try {
        connection = new Socket(connectionHost, connectionPort);
        output = new ObjectOutputStream(connection.getOutputStream());
        output.flush();
        input = new ObjectInputStream(connection.getInputStream());

        m.performative = "getcapy";
        m.content = thisr;

        sendMessage(m, output);
        m = readMessage(input);
        ((RoomUser) parent).receiveARoomWithCapy(
            (RoomWithCapy) m.getContent());
        input.close();
        output.close();
        connection.close();
    }
    catch (Exception e) {
        System.out.println("Error: " + e);
    }
}
}

/**
 * The RoomUser's side of a conversation begun by the "ARoomWithCapy" event
 */
import java.net.*;
import java.io.*;
import agent.mom.*;
```

```
public class RWC_INTERFACE extends Conversation
{
    Agent parent; // override parent
    RoomWithCapy thisrwc;

public RWC_INTERFACE(RoomUser a, RoomWithCapy rwc) {
    super(a, "localhost", 3300);
    parent = a;
    thisrwc = rwc;
}

public void run() {
    Message m = new Message();
    System.out.println("Starting Rwc conversation.");
    try {
        connection = new Socket(connectionHost, connectionPort);
        output = new ObjectOutputStream(connection.getOutputStream());
        output.flush();
        input = new ObjectInputStream(connection.getInputStream());
        m.performative = "add";
        m.content = thisrwc;
        sendMessage(m, output);
        input.close();
        output.close();
        connection.close();
    }
    catch (Exception e) {
        System.out.println("Error: " + e);
    }
}
}

\**
 * The RoomKeeper's side of all conversations
 */
import java.net.*;
import java.io.*;
import afit.mom.*;
public class KEEPER_RECEIVE_INTERFACE extends Conversation {
    RoomKeeper parent; // override parent

public KEEPER_RECEIVE_INTERFACE(Socket s, ObjectInputStream i,
    ObjectOutputStream o, RoomKeeper a, Message m1) {
    super(s, i, o, a, m1);
    parent = a;
}

public void run() {
    int state = 0;
    boolean notDone = true;
    System.out.println("Got >>" + m.getPerformative() + " - " + m.getContent() +
        " from " + m.getSender());
    try {
        if (m.getPerformative().equals("add"))
            parent.receiveARoomWithCapy((RoomWithCapy) m.getContent());
        else if (m.getPerformative().equals("getcapy")) {
            parent.receiveARoom((Room) m.getContent());
            while (!parent.getReady()){}
            m.setContent(parent.getTemprwc());
            parent.setNotReady();
            sendMessage(m, output);
        }
        else if (m.getPerformative().equals("getroom")) {
            while (notDone) {
                parent.receiveACapy(((Integer) m.getContent()).intValue());
                while (!parent.getReady()){}
                m.setContent(parent.getTemprwc());
                parent.setNotReady();
                sendMessage(m, output);
                if (parent.getTemprwc().getBldg().equals("Zero"))
                    notDone = false;
                else m = readMessage(input);
            }
        }
        input.close();
        output.close();
        connection.close();
    }
    catch (Exception e) {
        System.out.println("Error: " + e);
    }
}
}

\**
 * The RoomUser
 */
import java.io.*;
import java.util.*;
import java.net.*;
import afit.mom.*;
import amomsupport.*;
public class RoomUser extends Agent implements User_States, Menu_Choice,
    Runnable {
    .
    .
    .
    protected boolean ready; //added for agentMom support
    public String keeperHost; //added for agentMom support
    public int keeperPort; //added for agentMom support

public RoomUser(String agentName, int agentPort, String sHost, int sPort) {
        super(agentName, agentPort);
        this.initRoomUser();
        this.receivedEvent = "";
        this.tempc = 0;
        this.tempr = null;
        this.temprwc = null;
        this.tempchoice = 0;
        this.transID = 0;
        this.keeperHost = sHost; // the Host to connect to
        this.keeperPort = sPort; // the Port to connect to
    }

    public void agentMomSend(int i) {
        (new Thread(new CAPY_INTERFACE(this, i))).start();
    }

    public void agentMomSend(Room r) {
        (new Thread(new ROOM_INTERFACE(this, r))).start();
    }

    public void agentMomSend(RoomWithCapy rwc) {
        (new Thread(new RWC_INTERFACE(this, rwc))).start();
    }

    public synchronized void finalize() {
    }

    public boolean getReady() {
        return ready;
    }

    public boolean isReady() {
        return ready;
    }

    public static void main(String[] args) {
        RoomUser RU = new RoomUser("RoomUser", 4400, "localhost", 3300);
        RU.run();
    }

    public void OutputRIC() {
        tempricset = roomsInConstraint;
        roomsInConstraint = new RoomWithCapySetType();
    }

    public void receiveMessage(Socket server, ObjectInputStream input,
        ObjectOutputStream output) {
    }

    public void run() {
        this.stdioSystemSend = new amomsupport.StdIO(this);
    this.transitions();
}

public void sendACapy() {
    if (transID == 7) {
        agentMomSend(tempc);
    }
    if (transID == 9) {
        agentMomSend(tempc);
    }
}

public void sendARoom() {
    if (transID == 8) {
        agentMomSend(tempr);
    }
}

public void sendARoomWithCapy() {
    if (transID == 6) {
        agentMomSend(temprwc);
    }
    if (transID == 11) {
        stdioSystemSend.showRoomWithCapy(temprwc);
    }
}

public void sendCapyPrompt() {
    if (transID == 3) {
        stdioSystemSend.capyPrompt();
    }
}

public void sendMenuPrompt() {
    if (transID == 1) {
        stdioSystemSend.menuPrompt();
    }
    if (transID == 6) {
        stdioSystemSend.menuPrompt();
    }
    if (transID == 12) {
        stdioSystemSend.menuPrompt();
    }
    if (transID == 13) {
        stdioSystemSend.menuPrompt();
    }
}

public void sendRoomPrompt() {
    if (transID == 4) {
        stdioSystemSend.roomPrompt();
    }
}

public void sendRoomWithCapyPrompt() {
    if (transID == 2) {
        stdioSystemSend.roomWithCapyPrompt();
    }
}

public void sendShowRoomsInConstraint(){
    if (transID == 10) {
        stdioSystemSend.showRICSet(tempricset);
    }
}

public void setNotReady() {
    this.ready = false;
}

public void setReady() {
    this.ready = true;
}

public void setReady(boolean newReady) {
    ready = newReady;
}
}

\**
 * The RoomKeeper
 */
import java.util.*;
import java.net.*;
import java.io.*;
import afit.mom.*;
import amomsupport.*;
public class RoomKeeper extends Agent implements Keeper_States {
    .
    .
    .
    protected boolean ready; //added for agentMom support

public RoomKeeper(String s, int p) {
    super(s, p);
    this.initRoomKeeper();
    receivedEvent = "";
    transID = 0;
    temprwc = null;
    tempr = null;
    tempc = 0;
    MessageHandler handler;
    handler = new MessageHandler(port, this);
    handler.start();
}

public boolean getReady() {
    return ready;
}

public RoomWithCapy getTemprwc() {
    return temprwc;
}

public boolean isReady() {
    return ready;
}

public static void main(String[] args) {
    RoomKeeper RK = new RoomKeeper("RoomKeeper", 3300);
    RK.run();
}

public void receiveMessage(Socket server, ObjectInputStream input,
    ObjectOutputStream output) {
    int i;
    Message m;
    Thread t;
    try {
        m = (Message) input.readObject();
        System.out.println("Received message "+m.performative+" from "+m.sender);
        t = new Thread(new KEEPER_RECEIVE_INTERFACE(server, input, output, this,
            m));
        t.start(); // start new thread
    }
    catch (Exception e) {
        System.out.println("Error: " + e);
    }
}

public void run() {
    this.transitions();
}

public void sendARoomWithCapy(){
    if (transID == 3)
        this.setReady();
    if (transID == 4)
        this.setReady();
}

public void setNotReady() {
    this.ready = false;
```

```
}

public void setReady() {
    this.ready = true;
}

public void setReady(boolean newReady) {
    ready = newReady;
}

public void setTemprwc(RoomWithCapy newTemprwc) {
    temprwc = newTemprwc;
}

}
```

Bibliography

1. Ashby, Michael R. *Tool-Based Integration and Code Generation of Object Models.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 2000. AFIT/GE/ENG/00M-02.

2. Bolognesi, T. and E. Brinksma. "Introduction to the ISO specification language LOTOS," *The Formal Description Technique LOTOS*, 23–37 (1989).

3. Chauhan, D. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation.* MS thesis, University of Cincinnati, ECECS Department, 1997.

4. Cohen, Philip R., et al. "An Open Agent Architecture," *Readings in Agents*, 197–204 (1998). Morgan Kaufmann Publishers, Inc.

5. Cutkosky, Mark R., et al. "PACT: An Experiment in Integrating Concurrent Engineering Systems," *Readings in Agents*, 46–55 (1998). Morgan Kaufmann Publishers, Inc.

6. DeLoach, Scott A. "Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems." *Proceedings of Agent Oriented Information Systems '99 (AOIS'99).* May 1999.

7. DeLoach, Scott A. "Using agentMom." Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, 1999.

8. d'Inverno, M., et al. "Formalisms for Multi-Agent Systems," *The Knowledge Engineering Review*, *12*(3) (1997).

9. d'Inverno, M. and M. Luck. "Development and Application of an Agent Based Framework." *Proceedings of the First IEEE International Conference on Formal Engineering Methods*, edited by Hinchey and Shaoying. 222–231. 1997.

10. Durfee, E. H., et al. "The Agent Architecture of the University of Michigan Digital Library," *Readings in Agents*, 98–108 (1998). Morgan Kaufmann Publishers, Inc.

11. Finin, Tim, et al. "KQML as an agent communication language," *Communications of the ACM*, 456–463 (November 1994).

12. Fischer, Klaus, et al. "A Pragmatic BDI Architecture," *Readings in Agents*, 217–231 (1998). Morgan Kaufmann Publishers, Inc.

13. Foner, Leonard N. *What's An Agent, Anyway? A Sociological Case Study.* Technical Report Agents Memo 93-01, MIT Media Laboratory, E15-305, 20 Ames St, Cambridge, MA 02139: MIT Media Lab, May 1993.

14. Graham, Robert P. "Common Object-Oriented Imperitive Language." Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, September 1999.

15. Hartrum, Thomas C. "Object Oriented Design." Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, 1998.

16. Hartrum, Thomas C. "Introduction to Formal Systems." Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, 1999.

17. Hartrum, Thomas C. and Scott A. DeLoach. "Design Issues for Mixed-Initiative Agent Systems." *Proceedings of Agent Oriented Information Systems '99 (AOIS'99)*. July 1999. AAAI-99 Workshop in Mixed-Initiative Intelligence.

18. Huhns, Michael N. and Munindar P. Singh. "Agents and Multiagent Systems: Themes, Approaches, and Changes," *Readings in Agents*, 1–23 (1998). Morgan Kaufmann Publishers, Inc.

19. Huhns, Michael N., et al. "Global Information Management via Local Autonomous Agents," *Readings in Agents*, 36–45 (1998). Morgan Kaufmann Publishers, Inc.

20. Ishizaki, Suguru. "Multiagent Model of Dynamic Design," *Readings in Agents*, 172–179 (1998). Morgan Kaufmann Publishers, Inc.

21. Kautz, Henry, et al. "An Experiment in the Design of Software Agents," *Readings in Agents*, 125–138 (1998). Morgan Kaufmann Publishers, Inc.

22. Kendall, Elizabeth A. "Agent Roles and Role Models: New Abstractions for Intelligent Agent System Analysis and Design." *AIP'98 (Intelligent Agents for Information and Process Management)*. 1998.

23. Kendall, Elizabeth A., et al. "The Application of Object-Oriented Analysis to Agent Based Systems," *The Report on Object Oriented Analysis and Design in conjunction with The Journal of Object Oriented Programming* (June 1997). Royal Melbourne Institute of Technology.

24. Kinney, D., et al. "A Methodology and Modelling Technique for Systems of BDI Agents." *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW-96*. 1996.

25. Kissack, John A. *Transforming Aggregate Object-Oriented Formal Specifications to Code*. MS thesis, Air Force Institute of Technology, March 1999. DTIC Number ADA361759.

26. Kuokka, Danuel and Larry Harada. "Matchmaking for Information Agents," *Readings in Agents*, 91–97 (1998). Morgan Kaufmann Publishers, Inc.

27. Lander, Susan E. "Issues in Multiagent Design Systems," *IEEE Expert*, *12*(2):18–26 (March-April 1997).

28. Lashkari, Yezdi, et al. "Collaborative Interface Agents," *Readings in Agents*, 111–116 (1998). Morgan Kaufmann Publishers, Inc.

29. Luck, Michael and Mark d'Inverno. "Structuring a *Z* Specification to Provide a Formal Framework for Autonomous Agent Systems," *Lecture Notes in Computer Science*, *967*:47–72 (1995). In ZUM'95: The *Z* Formal Specification Notation.

30. Noe, Penelope Ann. *A Structured Approach to Software Tool Integration*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1999. DTIC Number ADA361674.

31. Object Management Group, Inc. *OMG Unified Modeling Language Specification.* http://www.rational.com/uml/index.jtmpl, 1999. Version 1.3.

32. Rao, Anand S. "Modeling Rational Agents withn a BDI-Architecture," *Readings in Agents*, 317–328 (1998). Morgan Kaufmann Publishers, Inc.

33. Reasoning Systems, Inc. *REFINE User's Guide*, 1995.

34. Rich, Charles and Candace L. Sidner. "COLLAGEN: When Agents Collaborate with People," *Readings in Agents*, 117–124 (1998). Morgan Kaufmann Publishers, Inc.

35. Rumbaugh, James, et al. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey 07632: Prentice Hall, 1991.

36. Shoham, Yoav. "Agent-Oriented Programming," *Readings in Agents*, 329–349 (1998). Morgan Kaufmann Publishers, Inc.

37. Sward, Ricky E. *Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code.* PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, September 1997. DTIC Number ADA361674.

38. Sycara, Katia P. "Multiagent Systems," *AI Magazine*, *19*(2):79–92 (Summer 1998). American Association for Artificial Intelligence.

39. Tankersley, Travis W. *Generating Executable Code from Formal Specifications.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1999. DTIC Number ADA361722.

40. Wang, Enoch Y., et al. "Formalizing and Integrating the Dynamic Model within OMT." *Proceedings of the International Conference on Software Engineering.* May 1997. Department of Computer Science, Michigan State University.

41. Wooldridge, Michael, et al. "A Methodology for Agent-Oriented Analysis and Design." *Autonomous Agents '99, Proceedings of the Third Annual Conference on Autonomous Agents*, edited by Oren Etzioni, et al. 69–76. ACM Press, May 1999. Queen Mary & Westfield College and University of Melbourne.

31. Object Management Group, Inc. *OMG Unified Modeling Language Specification.* http://www.rational.com/uml/index.jtmpl, 1999. Version 1.3.

32. Rao, Anand S. "Modeling Rational Agents withn a BDI-Architecture," *Readings in Agents*, 317–328 (1998). Morgan Kaufmann Publishers, Inc.

33. Reasoning Systems, Inc. *REFINE User's Guide*, 1995.

34. Rich, Charles and Candace L. Sidner. "COLLAGEN: When Agents Collaborate with People," *Readings in Agents*, 117–124 (1998). Morgan Kaufmann Publishers, Inc.

35. Rumbaugh, James, et al. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey 07632: Prentice Hall, 1991.

36. Shoham, Yoav. "Agent-Oriented Programming," *Readings in Agents*, 329–349 (1998). Morgan Kaufmann Publishers, Inc.

37. Sward, Ricky E. *Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code.* PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, September 1997. DTIC Number ADA361674.

38. Sycara, Katia P. "Multiagent Systems," *AI Magazine*, *19*(2):79–92 (Summer 1998). American Association for Artificial Intelligence.

39. Tankersley, Travis W. *Generating Executable Code from Formal Specifications.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1999. DTIC Number ADA361722.

40. Wang, Enoch Y., et al. "Formalizing and Integrating the Dynamic Model within OMT." *Proceedings of the International Conference on Software Engineering.* May 1997. Department of Computer Science, Michigan State University.

41. Wooldridge, Michael, et al. "A Methodology for Agent-Oriented Analysis and Design." *Autonomous Agents '99, Proceedings of the Third Annual Conference on Autonomous Agents*, edited by Oren Etzioni, et al. 69–76. ACM Press, May 1999. Queen Mary & Westfield College and University of Melbourne.

Vita

David Wesley Marsh was born on 4 September 1970 in Salem, Oregon. He graduated from Nampa Christian High School, Nampa, Idaho in June 1988. He entered undergraduate studies at Seattle Pacific University in Seattle, Washington where he graduated with a Bachelor of Science in Electrical Engineering on 30 May 1992. He accepted a four-year scholarship and was commissioned through the Detachment 910 AFROTC at the University of Washington in Seattle, Washington on 12 June 1992. David's family includes his wife, Kristina [Knepshield], whom he married on 6 June 1992 and their son, Quinn, who was born 26 February 1998.

Before arriving at the Air Force Institute of Technology (AFIT) in August 1998, he worked 18 months as a maintenance officer supervising B-52 flight line maintenance at Griffiss AFB, NY, and nearly four years at Kadena AB, Okinawa, Japan, where he filled a variety of positions supervising back-shop and flight line maintenance of F-15, KC-135, HH-60, E-3, and C-130 aircraft. He attended Aircraft Maintenance and Munitions Officers' Course at Sheppard AFB, Texas from September 1993 through January 1994. At AFIT David was assigned to the Department of Electrical and Computer Engineering, Graduate School of Engineering and Management and will be assigned to the Information Directorate, Air Force Research Laboratory at Rome, NY, upon graduation.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 9 March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE FORMAL OBJECT STATE MODEL TRANSFORMATIONS FOR AUTOMATED AGENT SYSTEM SYNTHESIS	5. FUNDING NUMBERS
6. AUTHOR(S) David Wesley Marsh	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Bldg 640 Wright-Patterson AFB OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/00M-03
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NM Attn: Captain Freeman Alex Kilpatrick 801 North Randolph Street, Room 732 9-65 Arlington VA 22203-1977 Phone: (703)-696-6565	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Dr. Thomas C. Hartrum, ENG, Phone: (937) 255-3636 ext. 4581

12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Automated agent system synthesis is the process of generating code from a requirements specification with appropriate inputs from the software engineer. Object-oriented (OO) specifications are frequently used to model intelligent software agent systems and software requirements in general; formal representations capture precisely the intentions of the specifier. Portions of OO specifications can be classified as the structural, functional, and state (or dynamic) models; major strides have been taken in the development of transformations for creating code from formal OO specifications, specifically the structural and functional aspects, and are captured within the AFIT Wide-Spectrum Object Modeling Environment (AWSOME). This research creates a methodology for the automatic transformation of the dynamic model into structural and functional components which can then be exploited for the generation of executable code exactly reflecting the original intent of the requirements specification. The integration of agent communication protocols within this context is addressed, providing a methodology for the incorporation of various agent-to-agent and agent-to-human interaction schemes. Feasibility is demonstrated through the application of transformations to a formal requirements model within AWSOME resulting in executable code.

14. SUBJECT TERMS software engineering, transformation systems, artificial intelligence agents, specification analysis, code generation, state model, formal methods	15. NUMBER OF PAGES 139
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL