



**A COMPONENT BASED APPROACH TO AGENT
SPECIFICATION**

THESIS

David J. Robinson, 1st Lieutenant, USAF

AFIT/GCS/ENG/00M-22

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

A COMPONENT BASED APPROACH TO AGENT SPECIFICATION

THESIS

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

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

David J. Robinson, B. S.

1st Lieutenant, USAF

March 2000

Approved for public release, distribution unlimited.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

A COMPONENT BASED APPROACH TO AGENT SPECIFICATION

THESIS

David J. Robinson, B. S.
1st Lieutenant, USAF

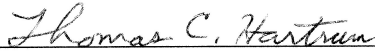
Approved:



Maj Scott A. Deloach

2 Mar 2000

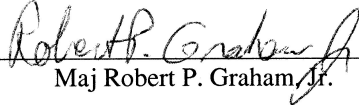
date



Dr. Thomas C. Hartrum

6 Mar 2000

date



Maj Robert P. Graham, Jr.

6 Mar 2000

date

ACKNOWLEDGMENTS

I would like to sincerely thank my faculty advisor, Maj Scott DeLoach. His guidance, support, relentless badgering, and the fact that he outranked me by quite a bit allowed for the authoring of this quality thesis. In all seriousness, it was truly an honor and a privilege to be his student. His incredible focus and energy was often contagious and made me a better student in spite of myself. Thank you. I would also like to thank Dr. Thomas Hartrum for his advice and guidance throughout the thesis process and Maj Robert Graham for serving on my Thesis committee. I would like to thank my fellow classmates for making this journey to all encompassing knowledge bearable and often quite fun.

Most importantly, I would like to thank my wife Andrea, whom without, this thesis would have not been written. Her love, support, and willingness to go shopping when I needed to study allowed me to focus on the task at hand. Thank you and I love you.

David J. Robinson

TABLE OF CONTENTS

ACKNOWLEDGMENTS.....	II
TABLE OF FIGURES	V
I. INTRODUCTION	1
<i>1.1 Background.....</i>	<i>2</i>
<i>1.2 Problem Statement.....</i>	<i>4</i>
<i>1.3 Thesis Overview.....</i>	<i>5</i>
II. BACKGROUND	6
<i>2.1 Overview.....</i>	<i>6</i>
<i>2.2 Knowledge Representation</i>	<i>6</i>
<i>2.3 Architectural Representation.....</i>	<i>12</i>
<i>2.4 Internal Agent Representation</i>	<i>36</i>
<i>2.5 Summary</i>	<i>52</i>
III. APPROACH	53
<i>3.1 Language Requirements.....</i>	<i>54</i>
<i>3.2 Selecting a Language.....</i>	<i>55</i>
<i>3.3 Object Model Definition</i>	<i>57</i>
<i>3.4 Language Validation.....</i>	<i>59</i>
<i>3.5 Chapter Summary</i>	<i>63</i>
IV. DESIGN AND IMPLEMENTATION	64
<i>4.1 Defining the Requirements.....</i>	<i>64</i>
<i>4.2 Review of Existing Languages</i>	<i>66</i>
<i>4.3 Language Definition</i>	<i>73</i>
<i>4.4 Object Model Definition</i>	<i>85</i>

4.5 <i>Language Implementation</i>	93
4.6 <i>Chapter Summary</i>	96
V. ARCHITECTURAL STYLES	98
5.1 <i>Reactive Agent Architectural Style</i>	98
5.2 <i>Knowledge Based Architectural Style</i>	105
5.3 <i>Planning Architectural Styles</i>	110
5.4 <i>BDI Architectural Style</i>	115
5.5 <i>Generic Components</i>	119
5.6 <i>Summary</i>	120
VI. CONCLUSIONS AND FUTURE WORK	122
6.1 <i>Conclusions</i>	122
6.2 <i>Language Usage</i>	123
6.3 <i>Possible Future Work</i>	125
6.4 <i>Thesis Summary</i>	126
REFERENCES	129
APPENDIX A OCL GRAMMAR [WK99]	134
APPENDIX B FIGURE 19 OPERATOR DEFINITION	137
APPENDIX C LANGUAGE IMPLEMENTATION	138
APPENDIX D REACTIVE STYLE OPERATOR DEFINITION	152
VITA	153

TABLE OF FIGURES

FIGURE 1 GENERIC ELEMENTS OF ADL [AES95]	14
FIGURE 2 PRS-CL ARCHITECTURE [PRS99].....	17
FIGURE 3 ARCHITECTURE OF A TYPICAL EXPERT SYSTEM [LS98].....	22
FIGURE 4 RETSINA ARCHITECTURE [SYC96].....	25
FIGURE 5 PRODIGY EXAMPLE [PROD92].....	29
FIGURE 6 ACT DIAGRAM EXAMPLE [WM97].....	32
FIGURE 7 OCL OBJECT MODEL.....	43
FIGURE 8 Z REPRESENTATION.....	46
FIGURE 9 SEMANTIC NETWORK EXAMPLE	48
FIGURE 10 FRAME EXAMPLE.....	51
FIGURE 11 APPROACH.....	53
FIGURE 12 STEP ONE.....	54
FIGURE 13 STEP TWO	55
FIGURE 14 STEP THREE	57
FIGURE 15 A GENERIC AGENT ARCHITECTURE.....	58
FIGURE 16 STEP FOUR.....	59
FIGURE 17 GENERIC BDI ARCHITECTURE.....	61
FIGURE 18 REFINED BDI ARCHITECTURE	62
TABLE 1 LANGUAGE REPORT CARD.....	67
FIGURE 19 REACTIVE ARCHITECTURE.....	72
FIGURE 20 COMPONENT GROUPING	74
FIGURE 21 AGGREGATION REMOVAL.....	76
FIGURE 22 INHERITANCE AND AGGREGATION REMOVAL.....	79
FIGURE 23 MESSAGEINTERFACE STATEDIAGRAM	80

FIGURE 24 INNER-AGENT CONNECTORS	82
FIGURE 25 OUTER-AGENT CONNECTORS.....	83
FIGURE 26 BASIC OBJECT MODEL.....	85
FIGURE 27 OBJECT MODEL REFINEMENT	86
FIGURE 28 COMPONENT ATTRIBUTES	87
FIGURE 29 ATTRIBUTE CLASS ATTRIBUTES	89
FIGURE 30 OPERATOR CLASS ATTRIBUTES	91
FIGURE 31 DATA STRUCTURE	92
FIGURE 32 COMPLETE OBJECT MODEL	93
FIGURE 33 COMPONENT & CONNECTORS.....	94
FIGURE 34 ATTRIBUTE INTERFACE.....	95
FIGURE 35 METHOD INTERFACE.....	95
FIGURE 36 COMPLETED DYNAMIC MODEL.....	96
FIGURE 37 REACTIVE ARCHITECTURAL STYLE	99
FIGURE 38 REACTIVE ARCHITECTURAL OBJECT MODEL.....	100
FIGURE 39 REACTIVE ARCHITECTURE COMPONENT MODEL.....	104
FIGURE 40 IO_INTERFACE SUBSTRUCTURE.....	105
FIGURE 41 KNOWLEDGE BASED ARCHITECTURAL STYLE	107
FIGURE 42 KNOWLEDGE BASE OBJECT MODEL	108
FIGURE 43 KNOWLEDGE-BASED COMPONENT DIAGRAM	110
FIGURE 44 PLANNER ARCHITECTURAL STYLE.....	111
FIGURE 45 DYNAMIC PLANNER OBJECT MODEL	113
FIGURE 46 DYNAMIC PLANNER COMPONENT DIAGRAM	114
FIGURE 47 STATIC PLANNER OBJECT MODEL	115
FIGURE 48 STATIC PLANNER COMPONENT DIAGRAM	116
FIGURE 49 BDI ARCHITECTURAL STYLE.....	117
FIGURE 50 BDI OBJECT MODEL.....	118

FIGURE 51 BDI COMPONENT DIAGRAM	119
FIGURE 52 PLANNER COMPONENT SUBSTRUCTURE	119
FIGURE 53 COMPONENT BASELINE.....	120
FIGURE 54 CREATING AN AGENT	139
FIGURE 55 INTERNAL AGENT PANEL.....	140
FIGURE 56 COMPONENT MENU	141
FIGURE 57 COMPONENT PROPERTIES	141
FIGURE 58 COMPONENT ATTRIBUTE	142
FIGURE 59 RULECONTAINER ATTRIBUTES	142
FIGURE 60 RULE CONTAINER METHOD.....	143
FIGURE 61 RULECONTAINER COMPONENT.....	144
FIGURE 62 REACTIVE AGENT COMPONENTS	145
FIGURE 63 OUTER AGENT CONNECTOR.....	146
FIGURE 64 COMPONENT & CONNECTORS.....	147
FIGURE 65 COMPONENT STATE DIAGRAM.....	148
FIGURE 66 WAIT STATE ADDED.....	148
FIGURE 67 WAIT STATE WITH TRANSITIONS	149
FIGURE 68 DUMMY STATE	150
FIGURE 69 TRANSITION MENU	150
FIGURE 70 COMPLETED DYNAMIC MODEL.....	151

ABSTRACT

The Air Force, as well as all of industry, is currently faced with the problem of having to produce larger and more complex software systems that run efficiently and reliably as well as being extensible and maintainable. This research addresses this problem by developing a knowledge representation language that can be used to unambiguously specify and design software systems in a verifiable, efficient, and understandable manner. The language is a combination of object-oriented and component-based methodologies and makes use of both graphics and text to represent information. Although designed for the development of any type of software system, the language has been implemented in *agentTool*, a multi-agent development environment.

A COMPONENT BASED APPROACH TO AGENT SPECIFICATION

I. Introduction

With the size and complexity of software systems increasing, the overall design, specification, and verification of the structure of systems turns into a crucial concern for software engineers. Software engineering as a whole has made very few significant advances in the past two decades in the realm of software development [MM97]. Structured programming, declarative specifications, object-oriented programming, formal methods, and visual programming were supposed to bring software development to a level capable of implementing software systems correctly and efficiently. Unfortunately, this has not happened and software systems have not kept pace with the rest of the computing industry. While processor performance increased at a rate of 48% per year, and network capacity increased at a 78% annual rate, software productivity increased by only 4.6% and the power of programming languages and tools has been growing 11% per year [MM97].

The Air Force, as well as all of industry, is currently faced with the problem of having to produce larger and more complex software systems that run efficiently and reliably as well as being extensible and maintainable. The lack of a well-established notation upon which software engineers may agree has made solving this problem even more difficult. The goal of this research is to develop a knowledge representation language that can be used to unambiguously specify and design software systems in a verifiable, efficient, and understandable manner. To ensure

maximum understandability and ease of use, the language should make use of both graphics and text to represent information.

1.1 Background

A newer paradigm that is gaining in acceptance and use in the realm of software engineering is multi-agent systems design. Multi-agent systems are computational systems in which several semi-autonomous software agents interact or work together to perform some set of tasks or to satisfy some set of goals. These systems vary from those involving computational agents that are homogeneous or heterogeneous, to those including participation on the part of humans and intelligent computational agents. Research and use of these systems generally focuses on problem solving, communication, and coordination aspects, as opposed to low-level parallelization or synchronization issues that are more the focus of distributed computing. A significant advantage seen to using multi-agent systems is their ability to handle distributed problem solving. Large, complex problems that were normally solved by single applications on single machines can now be broken down and distributed to multiple applications running on multiple machines. Parallelism can be achieved by assigning different tasks or sub-problems to different agents. Possible coordination and information sharing can take place between agents in a manner that is totally transparent to the user. Another advantage to using multi-agent systems is their scalability. Since they are inherently modular, it is a simple to add a new agent to a multi-agent system in order to adapt to new problems without having to rewrite or redesign the whole system.

A methodology being revived to help improve the development of software systems is formal methods. Formal methods involve the use of mathematically based languages, techniques, and tools for specifying and verifying systems. Although formal methods have existed for quite

some time, a lack of usable tools and general acceptance has left the methodology a virtually untapped resource in software development. The majority of past research has been concerned with developing formal notation and inference rules while very little effort has been put toward the development of methodology and tool support [FKV94]. The correct use of formal methods allows requirements to be better understood, contradictions and ambiguities to be removed, specifications verified for correctness, and allows for a much smoother transition from specification to design to implementation [FKV94].

Graphical modeling is very much the opposite of formal methods. Although it is said that a picture is worth a thousand words, it should also be added that a picture is worth a thousand interpretations. A problem with modeling using graphical tools is user understanding. Often pictures and diagrams do not contain enough detail to eliminate ambiguities in how the designer wishes to represent something. The advantage of visual descriptions is seen in their ability to communicate complex relationships. Figures and diagrams are usually much easier to convey to others and are also easier for others to understand. These two points make graphical modeling a very attractive addition to software engineering.

The combination of multi-agent systems, formal methods, and graphical modeling is not a silver bullet that will take software engineering to a realm of creating correct software the first time every time. However, the correct use of certain aspects of each methodology offers a means to achieve robust reliable software capable of solving a large number of complex problems. The Air Force Institute of Technology (AFIT) is currently conducting research in the development of a multi-agent design tool called agentTool. The overall methodology is based on the *Multiagent Systems Engineering* (MaSE) [MASE99] approach to agent system design proposed by DeLoach. This thesis will define the Agent Definition Language (AgDL) used in MaSE. DeLoach's AgDL

is based on the development of a knowledge representation language for completely describing the internal behavior of individual agents of a system.

1.2 Problem Statement

The Air Force currently does not implement any type of formal methodology for developing software systems. As systems are becoming larger and more distributed, the need to integrate complex systems in distributed environments requires that systems have some common methodology. This statement has led to a two-fold problem the Air Force is currently struggling to correct. First, the problem of finding a useful, easy to use methodology that can be implemented Air Force wide and allows for software reuse, easy documentation, and the removal of ambiguities. Second, the problem of finding a way to formally represent this methodology so verification and maintenance of software can be done easily and cheaply. Many methodologies currently exist and are well defined for the development of software. However, in choosing any one of these methodologies to solve the first problem, the second remains almost completely intact and vice versa. The solution to this dilemma does not involve abandoning current software practices to use one methodology to solve all of these problems, but rather an integration of formal methods within a software design methodology to create a formal, more verifiable software system [BH95].

The area that can benefit most from this integration is the specification phase of software development. The creation of a formally based knowledge representation language that is also easy to read and write will allow for unambiguous specifications to be written quickly and efficiently. Ensuring the language is also easily decomposable allows for maximum reuse. The use of such a language in a multi-agent development system will allow for individual agents to be formally specified, allowing for the verification of all components of the system.

1.3 Thesis Overview

Chapter 2 provides a review of literature including a definition of what a knowledge representation language is, what it is composed of, types of knowledge that must be represented in a knowledge representation language, and representation languages used in the past and present. Chapter 3 outlines the specific methodology followed to allow the transition from problem definition to design. Chapter 4 presents the design decisions made relating to the definition of the language as well as presents an implementation of the language. Chapter 5 defines software architectural styles relating to the field of artificial intelligence. Chapter 6 provides conclusions and areas requiring future work.

II. Background

2.1 Overview

The goal of this chapter is to review knowledge representation and how it may be used in Artificial Intelligence (AI) agent-based systems. To effectively design a knowledge representation language for a multi-agent system such as agentTool, past and present work in the area must be reviewed. A knowledge representation language needs to be defined and broken down into its most basic parts to determine what a representation language is and what is necessary to clearly and completely define such a language.

Section 2.2 addresses what knowledge representation languages are, why they are needed, and possible requirements. This thesis views representation languages at two levels of abstraction: architectural and internal. Section 2.3 outlines various aspects of architectural languages and representations while Section 2.4 does the same for internal languages and representations. Section 2.5 summarizes the chapter.

2.2 Knowledge Representation

Newell and Simon argued in their Turing Award lecture in 1976 that intelligent activity in a human or machine is achieved through the following:

1. Symbol patterns to represent significant aspects of a problem domain.
2. Operations on these patterns to generate potential solutions to problems.
3. Search to select a solution from among these possibilities.

The *physical symbol system hypothesis* is based on these three assumptions and outlines the major focus of most AI research: specifying the symbol structures and operations required for problem solving and defining efficient and correct strategies to search the plausible solutions generated by the structures and operations [LS98]. Brachman and Levesque [BL85] state that knowledge representation “simply has to do with writing down, in some language or communicative medium, descriptions or pictures that correspond in some salient way to the world or some state of the world.” At first, the task at hand does appear “simple”. Write things down in some language, be it graphical or textual (or both), in such a manner that the pictures and descriptions match what you are trying to represent. Unfortunately, the problem is not so easily solved. A number of issues must be addressed when defining a knowledge representation language, such as: [LS98, CM87, REICH91]

- what must be represented by the language
- how to formally specify the semantics of the language
- how to deal with incomplete knowledge
- how to maintain current knowledge
- how to control multiple inferencing
- how to represent meta-knowledge

These topics will be looked into more closely in the remainder of this thesis.

2.2.1 Representation Language Defined

Knowledge representation languages consist of two main components, syntax and semantics [RN95]. The *syntax* of a language deals with the manner in which information is

stored in an explicit format and describes the possible configurations that constitute a valid sentence. *Semantics* define the meaning of the symbols. If the semantics of a language are formally and completely specified, then a group of symbols can only be interpreted in one way.

Specification is the act of writing down information in a precise manner. A formal specification uses a representation language with precisely defined syntax and semantics in order to specify what a system is supposed to do. Because specifications eliminate distracting detail and only provide a general description, they become resistant to future system modifications [NASA95]. Regardless of the manner in which it is accomplished, specification is a representation process. In the realm of software engineering, these requirements are depicted in a manner that will lead to the successful implementation of a piece of software [PRES97]. The quality, timeliness, and completeness of any software project can usually be traced back to how much time was spent developing a specification of the system [PRES97].

Once it has been determined what knowledge representation is and why it is necessary, the issue of language requirements needs to be addressed.

2.2.2 Language Requirements

Luck and d'Inverno [DAF97, SZS95] describe three requirements they believe must be met in order for a language to be used to specify agents.

1. A language must precisely and unambiguously provide meanings for common concepts and terms and do so in a readable and understandable manner
2. A language should enable alternative designs of particular models and systems to be explicitly presented, compared and evaluated. It must provide a description of the common abstractions found within that class of models as well as a means of further refining these descriptions to detail particular models and systems.
3. A language should be sufficiently well structured to provide a foundation for subsequent development of new and increasingly more refined concepts. In other words, practitioners should be able to choose the level of abstraction suitable for their purpose.

The first requirement states the language must have a formally defined syntax and semantics as well as be easy to understand. The first half of this requirement does not pose a significant problem since a number of formally defined languages exist today. However, requiring something to be “readable” and “understandable” is a rather vague requirement since both words are subject to interpretation. Computer engineers often get caught up in defining information so that it is easier for a machine to understand, but often forget there is a human element interacting at some point in the process. The use of visual descriptions often aids in the ability to communicate complex relationships and concepts. Therefore, using graphics along with text should yield a much more clear and understandable description of a system rather than just using either approach independently of the other.

The second requirement states the language should have the flexibility to allow designers to represent a model in multiple ways. If the language is not flexible enough to allow multiple representations of a problem, the designer may be constrained to a single solution. The ability to represent and model numerous solutions to the same problem often provides insight that would otherwise be difficult to attain. This flexibility can lead to possible performance gains and improved resource usage not otherwise available. Multiple representations may greatly impact the implementation of the system.

The last requirement implies the language must have ability to represent multiple levels of abstraction. Any time a modular approach to a software problem is taken, as is the case in multi-agent systems, multiple levels of abstraction are found. At the highest level, a solution is stated in broad terms in a domain specific language. This may be as simple as a box and line diagram depicting the overall software system. At lower levels, a more procedural approach is taken and specific details are addressed [PRES97].

As with the flexibility, abstraction also allows issues such as performance and resource usage to be examined at various levels. For example, the designer may look at a design at the agent interaction level and find no way to improve performance. Upon closer inspection of the internal details, the designer may find several redundant communication requests that can be removed, thus reducing network congestion and increasing the overall performance of the system.

Another benefit of multiple levels of abstraction is the ability to hide information. If a design is being developed by multiple teams, each of which is working on a different aspect of the same problem, certain teams may not need to look at all levels of the design. The team interested in the agent communication structure is not concerned with the internal representation of each component making up the design. They do not need to know the internal representation to correctly model their portion of the design.

The final advantage of multiple levels of abstraction is communication and understandability. A design is much easier to communicate if it can be presented in multiple layers of abstraction. Software designs are often very large and complex. Multiple layers of abstraction allow a design to be communicated at a more incremental pace instead of having to understand the whole thing at once.

2.2.2.1 Multi-agent Requirements

The language requirements imposed thus far are fine if someone is just defining one agent, but in order to define agents in a multi-agent system, a number of additional requirements must be taken into account. A paper written by d'Inverno, Fisher, Lomuscio, Luck, de Rijke, Ryan and Wooldridge defines specific requirements a multi-agent language should contain [FMS97].

1. The multiplicity of agents
2. Group properties of agent systems, such as common knowledge and joint intention
3. Interaction among agents, such as communication and cooperation

The first requirement addresses the fact that a single agent type may be defined for a particular problem requiring multiple instances of the agent type to exist at any one time. The specification language chosen must be able to represent this attribute. Requirement two states that the language must allow for the representation of shared agent properties. Agents must be able to have common data stores that can be accessed and manipulated by all agents. The last statement implies that the interaction occurs strictly between agents. Interaction must not only exist between agents, but also between an agent and its environment.

2.2.2.2 Additional Requirements

Although all requirements mentioned thus far imply that the language needs to be expressive, the actual word is never used. The expressiveness of the language used to define any type of software system should not be overlooked. A language must be able to represent complex data types and operators. Many existing languages offer a number of predefined data types and operators a designer may use, but the key is being able to define new data types and operators not available. If a chosen language cannot be adapted to represent new language requirements, it will quickly become outdated.

A general requirement desirable in most languages is the ability to represent knowledge in a modular and reusable fashion. Although modularity does not guarantee reuse, for software to be reusable it is beneficial if it is modular. It has been stated that “modularity is the single attribute of software that allows a program to be intellectually manageable” [MYE78]. A “divide and conquer” approach offers a means of breaking a problem into smaller pieces that can be more

easily solved. If some thought is put into what should be considered a module, patterns may be identified which may in turn lead to reuse of existing modules, or the definition of new modules to be reused in the future.

A final language requirement not specifically addressed by any of those previously mentioned is the ability to capture the dynamic behavior of the agents. All requirements mentioned to this point have applied to the static structure of an agent, rather than its ability to express the dynamics of a system. Describing a system's static structure reveals what a system is composed of and how the parts are related, but it does nothing to explain how all of these parts work together to make the system more functional [EP98]. Because of this, an important requirement that should be considered when defining a language is the ability to capture the static and dynamic behavior of a system.

2.3 Architectural Representation

As stated in the chapter introduction, this thesis views knowledge representation at two levels of abstraction, architectural and internal. Maes defines an agent architecture as “a particular methodology for building [agents]. It specifies how the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact” [MAE91]. Agent architectures provide a higher level of abstraction for constructing and viewing agents and agent systems. Section 2.3.1 outlines the most common languages used for representing architectures. Sections 2.3.2 through 2.3.6 review the most common types of agent architectures and the components they are constructed from. Section 2.3.7 reviews the communication component that is common to all multi-agent architectures.

2.3.1 Architecture Description Language (ADL)

Software architectures define the high-level structure of a software system, showing its overall organization as a collection of interacting components [ACME97]. The structure of the system may include “global control structure; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; physical distribution; the composition of design elements; scaling and performance; dimensions of evolution; and selection among design alternatives”[GS96]. Simply stated, a software architecture defines elements from which software systems are built, the communication that takes place between these elements, the patterns of combination between one another, and the constraints that exist on these patterns [GS96]. Software architectures are important because of their ability to make complex systems manageable by describing them at a high level of abstraction and by allowing designers to take advantage of recurring patterns of system organization [AES95]. The basic elements comprising any software architecture are components and connectors. *Components* represent the repository for computation and state while *connectors* explicitly describe the interaction between these components. Each component has an interface specification that describes its properties. These properties may include, but are not limited, to the signature and functionality of its resources as well as performance properties [GS96]. Components are classified and named according to function. Examples of component types are filters, memory, and server. Connectors have protocol specifications that may include hookup rules, ordering rules, and performance properties as well as others [GS96]. Like components, connectors are also categorized by function and contain the following types; remote procedure call, pipeline, broadcast, etc.

Architectural design of software systems is not a totally new concept in the realm of software engineering, but until recently the process had been rather adhoc, lacking in formality and re-invented for each new design. Because of this, “architectural designs are often poorly

understood by developers; architectural choices are based more on default than solid engineering principles; architectural designs cannot be analyzed for consistency or completeness; architectural constraints assumed in the initial design are not enforced as a system evolves; and there are virtually no tools to help architectural designers with their tasks.” [ACME97] In response to these problems, architecture description languages (ADLs) were developed to act as a formal notation for representing and analyzing architectural designs. ADLs generally consist of a language and an environment. The language is used to define the component as well as the interconnection between components while the environment provides a medium to make the descriptions usable and reusable. The generic elements of an ADL are shown in Figure 1.

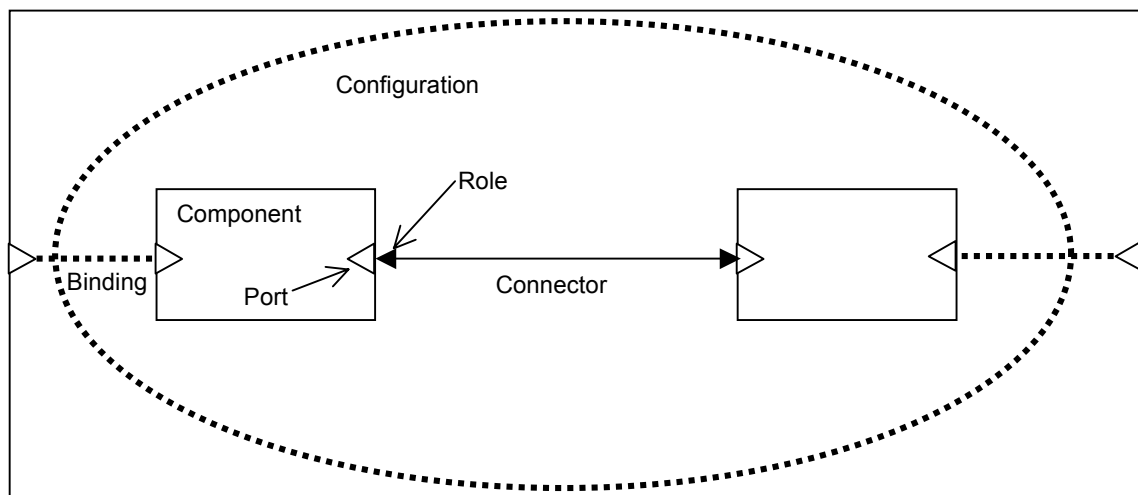


Figure 1 Generic Elements of ADL [AES95]

Although the above figure is specific to the Aesop [AES95] ADL, most ADLs are similar. Components and connectors were described above. *Configurations* define topologies of components and connectors. *Ports* define the component interfaces and determine the component’s points of interaction with the environment. *Roles* are the connector interfaces and identify the participants of the interaction. *Representations* (not shown in figure) refer to the descriptions of the “contents” of components and connectors. *Bindings* are used to define the

correspondence between elements of the internal configuration and the external interfaces of the component or connector.

ADLs also typically provide parsing, unparsing, displaying, compiling, and analyzing tools. Some of the more popular ADLs include Aesop, Adage, UniCon, and Wright [AES95, CS93, UNI95]. Although the purpose of each language is to represent the architectural design of software, each provides slightly different capabilities than the next. Aesop is primarily concerned with the use of architectural styles; Adage allows for the description of architectural frameworks for avionics navigation and guidance; UniCons use of a high-level compiler allows for the support of a mixture of component and connector types; and Wright allows for the specification and analysis of interactions between components [ACME97].

ADLs do not offer the user the ability to completely specify software systems. They do offer a means to analyze the overall structure of the system and the interaction that takes place between its parts before the internals of these parts are specified.

2.3.2 BDI Architecture

A widely used agent architecture is the Belief, Desire, Intention (BDI) architecture. This architecture consists of four basic components: beliefs, desires, intentions, and plans. The exact definition of these components varies slightly from author to author, but most generally agree that all four need to be present in one form or another when using this architecture. In this architecture, the agent's *beliefs* represent information that the agent has about the world, which in many cases may be incomplete or incorrect [DMAR97]. The content of this knowledge can be anything from knowledge about the agent's environment to general facts an agent must know in order to act rationally. The *desires* of an agent are a set of long-range goals, where a goal is typically a description of a desired state of the environment. An agent's *goals* simply represent

some desired end state. These goals may be defined by a user or may be adopted by the agent. New goals may be adopted by an agent due to an internal state change in the agent, an external change of the environment, or because of a request from another agent. State changes may cause rules to be triggered or new information to be inferred that may cause the generation of a new goal. Requests for information or services from other agents may cause an agent to adopt a goal that it currently does not possess. An agent's *desires* provide it with motivations to act. When an agent chooses to act on a specific desire, that desire becomes an *intention* of the agent. The agent will then try to achieve these intentions until it believes the intention is satisfied or the intention is no longer achievable [DMAR97]. The intentions of an agent provide a commitment to perform a plan. Although not mentioned in the acronym, plans play a significant role in this architecture. A *plan* is a representation outlining a course of action that, when executed, allows an agent to achieve a goal or desire. Plan representation will be discussed in greater detail in Section 2.3.6.

2.3.2.1 PRS-CL

One of the best established agent architectures currently available [DMAR97, WJ94], PRS has been used for several significant real world applications ranging from fault diagnosis on the space shuttle to air traffic control management [DMAR97, WJ94]. PRS is a BDI architecture that includes a plan library as well as explicitly defined symbolic representations for beliefs, desires, and intentions. PRS provides an architectural framework to express and execute knowledge in an easy and efficient manner. First described by [LG87], the architecture has evolved over the years to a number of usable implementations. One implementation of the PRS architecture is PRS-CL developed at SRI International.

PRS-CL consists of the following components [PRS99]:

- A database containing the agent's current beliefs or facts of the world

- A set of goals to be realized
- A set of plans describing how goals are achieved and how situations are reacted to
- An intention structure containing the plans chosen for eventual execution

Figure 2 shows a graphical depiction of the architecture.

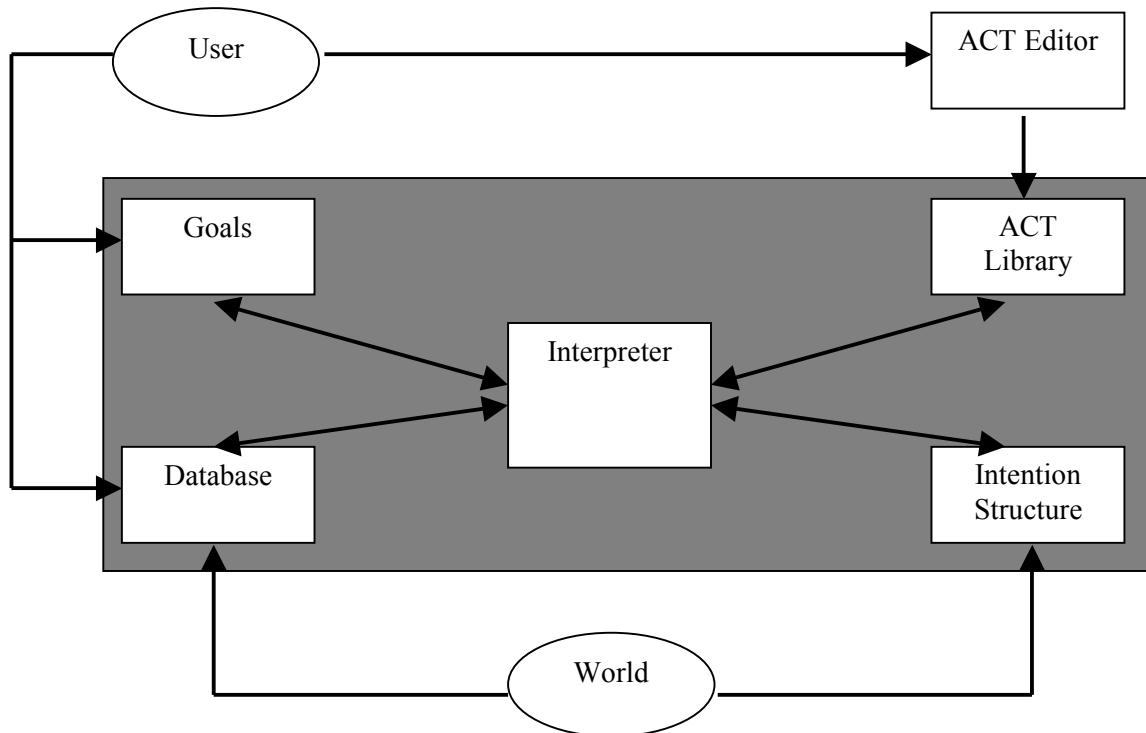


Figure 2 PRS-CL Architecture [PRS99]

The PRS-CL database contains an agent's beliefs, which include facts about the static and dynamic properties of the domain. The database knowledge is represented in Lisp syntax although it is not evaluable by a Lisp interpreter. New beliefs are acquired by the system dynamically as the agent executes its plans, which are referred to as Acts.

In PRS-CL, "goals are expressed as conditions over some interval of time (i.e., over some sequence of world states) and are specified as a combination of a goal operator applied to a

logical formula” [PRS99]. For example, the goal of (USE-RESOURCE(A B C)) indicates resources A, B, and C are needed for the completion of the Act to occur.

The backbone of the PRS-CL architecture is its use of Acts (Acts will be discussed in greater detail in Section 2.3.6.2). Acts are declarative procedure statements containing knowledge on how to accomplish goals and how to react to given situations. So although PRS is procedural by definition, the majority of the knowledge is represented in a declarative format. All Acts stored in PRS consist of a body and an invocation condition. The body of the Act describes the steps of the procedure to be taken. The Act body is considered a plan or plan schema in this architecture. The invocation condition contains triggering information describing all events that must occur before an Act is executed. In PRS-CL, Acts are represented as graphs containing a start node and one or more end nodes. The nodes of the graph are labeled with all subgoals to be achieved in carrying out the plan. The successful completion of a plan is realized when all subgoals between the start node and end node are achieved.

The final portion of the PRS-CL architecture is the intention structure. The intention structure holds the tasks the agent has chosen to execute. Intentions may either be executed immediately or at some time in the future depending on the invocation condition. Because of this, the intention structure of an agent may contain zero to many intentions, some of which are not currently active due to deferment or the waiting for certain conditions to become true.

2.3.2.2 Distributed Multi-Agent Reasoning System (dMARS)

D’Inverno, Kinny, Luck, & Wooldridge propose another PRS-based BDI architecture that is very similar to the one proposed by SRI except that it is based on formally specifying the agents [DMAR97]. One of the drawbacks to using PRS-CL is that the agent specification has no formal backbone. Ambiguities and program correctness cannot be easily verified until the agent system has been completely constructed. A significant advantage offered by dMARS is that an

agent specification is written using the Z formal specification language. Agent's beliefs, goals, intentions, plans, and actions are all specified using Z. Doing this allows system verification to be accomplished before implementation as well as the possibility of going from specification to implementation in a systematic manner.

2.3.3 Reactive Architectures

Perhaps simplest and among the most widely used agent architectures are reactive architectures. Wooldridge and Jennings [WJ94] describe a reactive architecture as an architecture that does not have a central world model and does not use complex reasoning. Unlike knowledge-based agents that have an internal symbolic model from which to work, reactive agents act by stimulus-response to environmental states. A simple example would be "If it starts raining, close all windows". The agent perceives an environmental change and reacts accordingly. Reactive agents can also react to messages from other agents. An example of a simple reactive agent may be a database "wrapper". The "wrapper" surrounds the database intercepting and interpreting all interactions with the database. Whenever a query for data comes in, the agent collects the appropriate data and returns it to the requestor. Although reactive agents are basic and can only perform simplistic tasks, they do form a building block from which other more complex agents can be built. By adding a knowledge base to a simple reactive agent, you now have an agent capable of making decisions that take into account previously encountered state information. By adding goals and a planning mechanism, you can create a rather complex goal directed agent. One of the more elaborate uses of reactive agents was seen in Brook's Subsumption Architecture [BRK85]. The Subsumption Architecture is based on Brook's belief that the Artificial Intelligence community need not build "human level" intelligence directly into machines. Citing evolution as an example, he claims we can first create simpler intelligences,

and gradually build on the lessons learned from these to work our way up to more complex behaviors. Brooks uses a layered finite state machine for representation of the agent's function. Although complex patterns of behavior can be developed using reactive agents, their primary goals usually consist of being robust and having a fast response time. Most agent architectures contain a reactive component of some kind, but are not actually truly reactive agents. The majority of reactive architectures can be modeled using a basic "IF-THEN" rule structure.

2.3.4 Knowledge-Based Architectures

Although the BDI architecture has a knowledge base, a large number of architectures exist built around a centralized knowledge store. In general these are referred to as knowledge-based or expert systems. Knowledge-based systems use data structures consisting of explicitly represented problem-solving information. This knowledge can be viewed as a set of facts about the world. Three aspects of knowledge-based systems making them powerful are:

1. They can accept new tasks in the form of explicitly described goals
2. They can achieve competence quickly by being told or learning new knowledge about the environment
3. They can adapt to changes in the environment by updating the relevant knowledge. [RN95]

In general, knowledge-based systems represent knowledge using a formal declarative language. Using a declarative language allows knowledge to be added or deleted from the knowledge base quickly and easily without affecting the rest of the system. Using a declarative language such as first-order logic also allows new information to be derived from the current knowledge stored in the system using inference mechanisms. An inference mechanism can perform two actions. First, given a knowledge base, it can generate new sentences that are necessarily true, given that the old sentences are true. Second, given a knowledge base and a

sentence, it can determine whether or not the sentence was generated by the knowledge base [RN95]. The relation just described between sentences is called entailment and is used a great deal in knowledge-based systems.

A common use of standalone knowledge-based systems is seen in expert systems. Although the name is often used synonymously with knowledge-based systems, expert systems are really a specific instantiation of a knowledge-based system. The first expert system, DENDRAL, was used to interpret the output of a mass spectrometer, an instrument used to analyze the structure of organic chemical compounds. A major result of the research done on expert systems has been the development of techniques that allow users to model information at increasing levels of abstraction. These techniques allow programs to be designed closely resembling human logic thus allowing for easier development and maintenance. Rule-based programming is the most common technique for the development of expert systems. Rules are used to represent heuristics specifying a set of actions to be performed for a given input. The foundation behind an expert system is its inference engine, which automatically matches facts against patterns and determines which rules are applicable. Once the inference engine finds an applicable rule, the actions of the rule are executed. The execution of the particular actions may affect the list of applicable rules by adding or removing facts. The inference engine then selects another rule and executes its actions. This process continues until no applicable rules remain. Figure 3 shows a general expert system architecture.

The **general knowledge base** in Figure 3 holds the problem-solving knowledge normally represented as a set of if-then rules. The **inference engine** is used to determine which actions to execute based on the information provided by the user. **Case specific data** is used to hold facts, conclusions, and other information relevant to the particular case being analyzed.

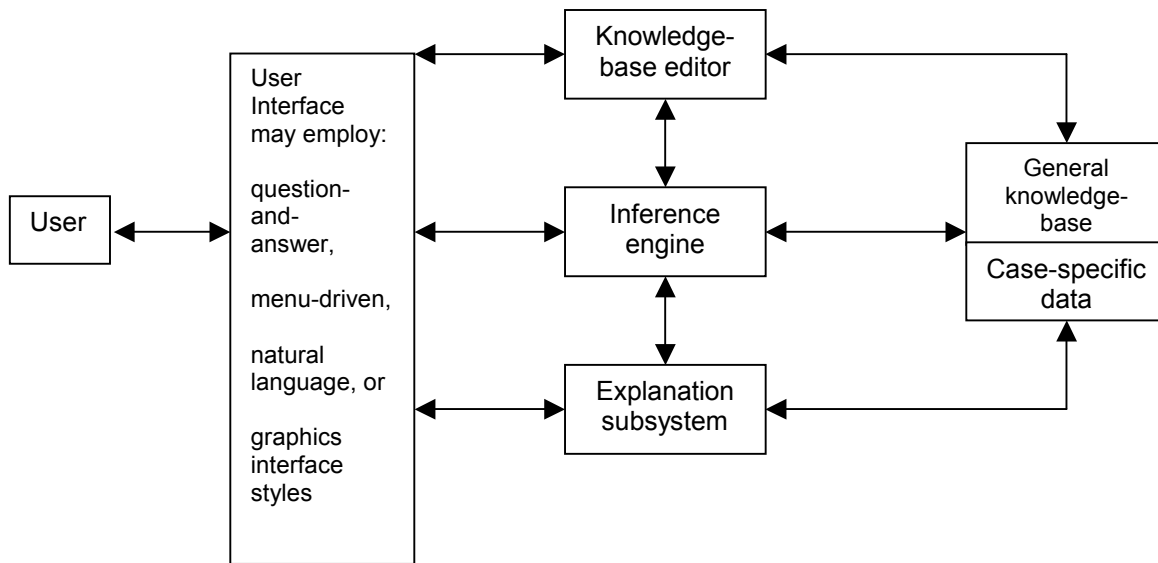


Figure 3 Architecture of a typical expert system [LS98]

An explanation subsystem is usually used to display to the user the reasoning used by the inference engine. The knowledge-based editor is simply a programming tool to allow a programmer to correct bugs in the knowledge base.

A significant problem in the design and use of expert systems is developing the knowledge needed to populate the knowledge base. The MYCIN expert system used for diagnosing spinal meningitis was developed in approximately 20 person-years. A number of tools were developed to aid in the design of expert systems. Two of these will be addressed in the remainder of this section.

2.3.4.1 C Language Integrated Production System (CLIPS)

CLIPS is an environment for the construction of rule and object-based expert systems. It is currently used by all NASA sites, all branches of the military, numerous federal bureaus, government contractors, universities, and many companies [CLIP98]. CLIPS allows for the handling of a variety of knowledge and includes support for rule-base, object oriented, and

procedural programming paradigms. Knowledge and queries to the system are formatted in a declarative Lisp-like syntax. CLIPS can be embedded within procedural code, and integrated with languages such as C, FORTRAN and ADA. This makes CLIPS ideal for either the embedding of knowledge within an agent or as a standalone expert system. CLIPS also provides a number of tools to support the verification and validation of expert systems. These tools provide support for modular design and partitioning of a knowledge base and semantic analysis of rule patterns to determine if inconsistencies could prevent a rule from firing or generating an error [CLIPS98].

A rule in CLIPS is similar to an IF THEN statement in a procedural language like Ada, C, or Pascal. An example of an actual rule would be as follows:

```
(defrule duck "Here comes the quack"      ; Rule header
  (animal-is duck)                        ; Pattern
=>                                         ; THEN arrow
(assert (sound-is quack)))                ; Action
```

Rules normally start with an optional rule-header comment which is specified in quotes. There can be only one rule-header comment and it must be placed after the rule name and before the first pattern. Following the rule-header is one or more patterns and actions. The number of patterns and actions do not have to be equal, which is why different indices, N and M, were used for the rule patterns and actions as seen in the above example.

Each pattern consists of one or more fields. In the duck rule, the pattern is (animal-is duck), where the fields are animal-is and duck. CLIPS attempts to match the patterns of rules against facts in its fact list. The fact-list consists of one or more declaratively defined CLIPS facts. Pattern matching can be done against a pattern entity, which is either a fact or an instance of a user-defined class.

An action in CLIPS is a function which typically has no return value, but performs some useful action, such as an assert or retract. The above example shows the function named

`assert` and its argument `sound-is quack`. This function returns no value, but instead asserts the fact `sound-is quack`. As with most programming languages, CLIPS recognizes a number of reserved keywords. The keyword `assert` is used to add data to the CLIPS fact-list.

2.3.4.2 Java Expert System Shell (JESS)

JESS is an expert system shell written in Java to develop rule-based expert systems that can be integrated in other Java code [JESS98]. Jess started as a clone of CLIPS, but evolved into a distinct environment of its own. In spite of this fact, JESS is still compatible with CLIPS and many JESS scripts are valid CLIPS scripts or vice-versa. JESS may be used to build Java applets and applications possessing the ability to reason over knowledge supplied by the user. Like CLIPS, Jess uses a declarative Lisp-like syntax for defining knowledge and rules.

2.3.5 Reusable Task Structure-based Intelligent Network Agents (RETSINA)

RETSINA was developed at Carnegie Melon as a technique “for developing distributed and adaptive collections of agents that coordinate to retrieve, filter and fuse information relevant to the user, task and situation, as well as anticipate a user’s information needs.” [SYC96] One reason that RETSINA is an important architecture is that it does not follow any of the “major” architectures covered in this chapter. In developing a robust knowledge representation language, one must take into account information in well-known architectures as well as those that are not as widely used. This section will show how even lesser known architectures contain the same knowledge components that the more common architectures possess.

RETSINA is a multi-agent infrastructure consisting of reusable agent types that can be adapted to address a variety of different domain-specific problems. RETSINA has three types of agents; interface agents, task agents, and information agents. In order for all of the agents to

effectively use and coordinate their resources, a common architectural backbone is needed for the agents to interact with. This architecture is shown in Figure 4.

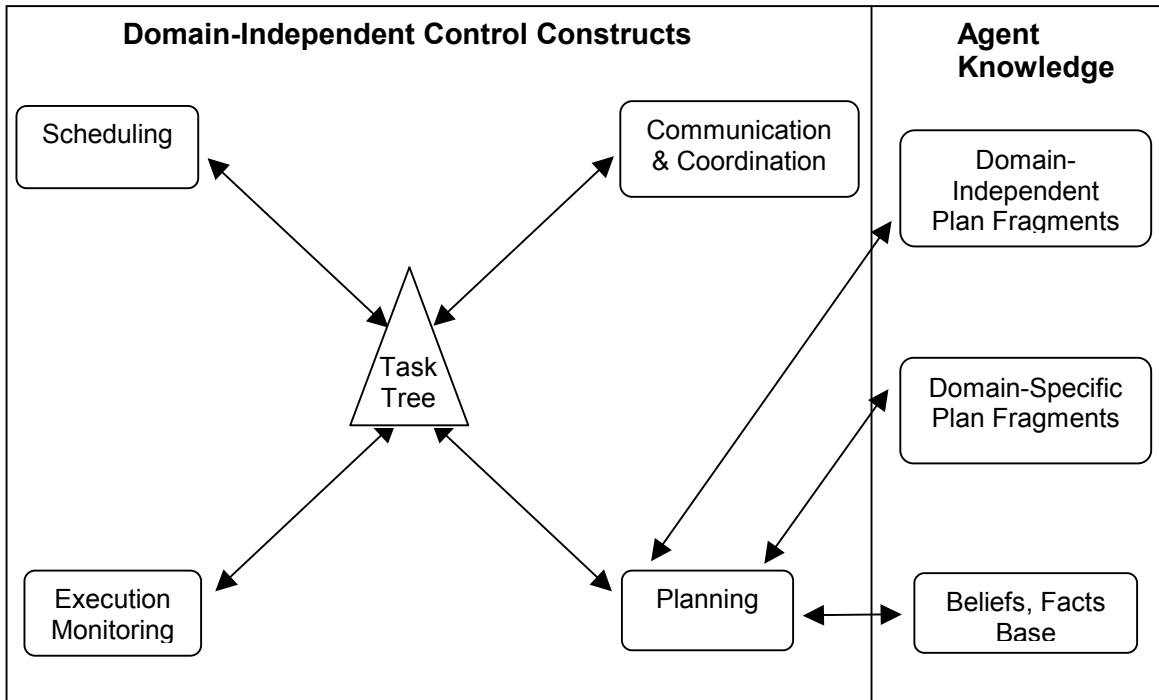


Figure 4 RETSINA Architecture [SYC96]

The communication and coordination module accepts and interprets Knowledge Query Manipulation Language (KQML) messages from other agents. In addition to this, interface agents also accept and interpret e-mail messages. Messages can contain requests for services that become goals of the recipient agent.

The planning module takes a set of goals as input and produces a plan that satisfies those goals. In RETSINA, the agent planning process is based on a hierarchical task network (HTN) planning formalism. Agents have a domain-independent library of plan fragments, which can be indexed by goals, as well as a domain-specific library of plan fragments that can be retrieved and incrementally instantiated according to input parameters.

The scheduling module is used to schedule each of the steps in a given plan. The scheduling process takes an agent's current set of plan instances (the set of executable actions) and decides which are to be executed next (if any). The action is identified as a fixed intention (as seen in BDI architectures) until it has been satisfied.

The execution monitor takes the agent's next intended action and prepares, initiates, and monitors its execution until completion.

2.3.6 Planning Architectures

A number of authors give various definitions for planning, but all boil down to the same essential facts. Planning is the process of formulating a list of actions in order to achieve a specified goal [MP92, RN95, PB94, LS98]. In Artificial Intelligence, a planner uses knowledge about the actions it may perform and their consequences, as well as knowledge about the environment, in order to formulate a list of acceptable state transforming operators that can transform the agent from an initial state to a goal state. As seen in BDI and RETSINA, planning architectures are usually embedded in other agent architectures to determine actions an agent will perform. Within a given agent architecture, plans may be either synthesized dynamically or defined ahead of time and placed in a plan library. In general, plans come in two varieties; total order and partial order. Total order plans simply consist of a list of steps that an agent must follow to accomplish some goal. These steps have a definite order that must be followed for the goal to be achieved. Partial ordered plans may have some steps ordered while the order of other steps is arbitrary and inconsequential to reaching the goal. An example of this would be putting on your pants. Given a goal to put your pants on, and a plan with the unordered steps "put left leg in pants" and "put right leg in pants", the order in which these are carried out does not matter as long as both of them are accomplished. If the step "zip up pants" is added to the list of steps, a

partial order now needs to be imposed. The first two mentioned steps can still be done in any order, but now they must both be accomplished before the zipping can occur. At one more level of abstraction, plans can be fully or partially instantiated. The steps of a plan are generally operators containing parameters that need to be set to some value in order for the operator to function. A fully instantiated plan is one in which all of these parameters are set to some value. Sometimes committing a variable to early in the planning process may overly constrain or limit the planner. For this reason, variables are often left uncommitted until a later time.

Russell and Norvig [RN95] state that a plan is a formally defined data structure that contains the following components:

- A set of plan steps. Each step is one of the operators of the problem.
- A set of step ordering constraints.
- A set of variable binding constraints.
- A set of causal links to record the purpose(s) of steps in the plan

The goal of this portion of the thesis is not to understand how a planner works, but to understand what knowledge a planner needs to function properly.

2.3.6.1 PRODIGY

One of the best known planning architectures is PRODIGY. PRODIGY has been used primarily as a research testbed in the areas of planning, machine learning, and knowledge acquisition.

In the simplest terms, PRODIGY searches for a list of actions that will transform an initial state into a goal state. In order to do this transformation, PRODIGY requires a certain

amount of information be available to it. All knowledge entered into PRODIGY must be in the PRODIGY4.0's Description Language (PDL4.0), which is essentially a watered down version of first order logic.

The first, and perhaps most critical piece of information PRODIGY needs is the domain theory, also called the domain specification. The *domain specification* consists of a type hierarchy for all entities in the domain, a set of operators, inference rules, and search control rules [PROD92]. The *type hierarchy* defines all of the objects that may be able to have operators applied to them. Once the type hierarchy is entered, operators and inference rules are defined. *Operators* are representations of actions leading to changes in the state. Operators have a set of effects, commonly referred to as postconditions, describing changes to the world that take place when the operator is applied. *Inference rules* represent all of the legal inferences that can be made for a given domain. For example, in a domain specified for a robot, there may be a single inference rule used to indicate that the robots arm is empty (represented by (arm-empty)). Every operator and inference rule has a precondition, which must be satisfied before the operator can be applied. An effects list describes how the application of the operator changes the state of the environment [PROD92]

Once the domain is totally described, an initial state and goal expression (any arbitrary PDL expression) can be defined. Prodigy then uses a backward chaining mechanism to find a sequence of operators that produce a state satisfying the goal expression. Figure 5 shows a simple example of how this works using the Extended-Strips Domain. Figure 5a shows the initial state of the world and figure 5b displays the goal state. The initial state consists of a robot, three boxes, and four doors. The robot starts in room 1, boxes 2 and 3 are in room 3, and box 3 is in room 4. The doors between rooms 1 and 3 and rooms 1 and 2 are open, while the doors between rooms 3 and 4 and rooms 2 and 4 are closed.

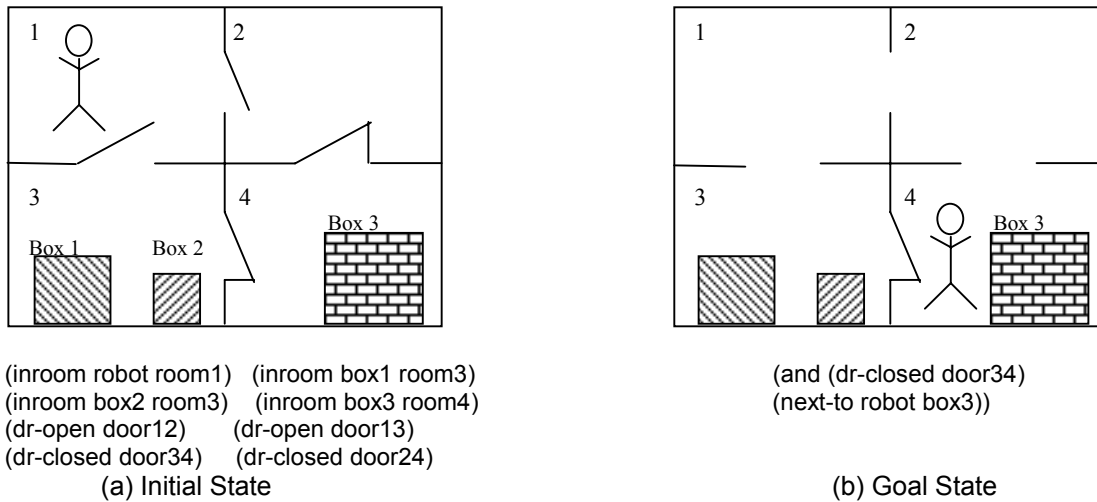


Figure 5 Prodigy Example [PROD92]

The goal is for the robot to end up next to box 3 with the door between rooms 3 and 4 closed.

Prodigy's solution is shown below.

```
Solution:
<goto-dr door13 room1>
<go-thru-dr door13 room1 room3>
<goto-dr door34 room3>
<open-door door34>
<go-thru-dr door34 room3 room4>
<close-door door34>
<goto-obj box3 room4>
```

To use PRODIGY in an agent system, the domain specification for each agent would need to be entered into PRODIGY. When an agent is presented with a goal inconsistent with its given state, it would have to transform its initial state and goal state into PDL and submit the information to PRODIGY. The agent would then be returned a solution, which would contain a list of actions the agent would have to sequentially execute in order to achieve its goal state.

2.3.6.2 Act Formalism

While PRODIGY is a domain independent planning architecture, the Act formalism is a domain independent language for the representation of plans. The Act formalism is a language used to represent knowledge necessary to develop complex plans.

For the most part, plan generation and reactive execution are considered separate entities in Artificial Intelligence [WM94]. The purpose of Act is to serve as a general representation language for sharing knowledge between multiple planning and execution systems. As discussed in Section 2.3.1.1.1, the basic unit used in the Act formalism is the Act. An *Act* “describes a set of actions that can be taken to fulfill some designated purpose under certain conditions” and can describe procedures, planning “operators”, or plans [WM97]. *Environment conditions* define the purpose and applicability criteria for a given Act. *Action specifications* are referred to as plots and consist of a partially ordered set of actions and subgoals. The environment conditions and plots are both specified using goal expressions.

Goal expressions define all requirements on the planning/execution process and the desired states to reach [WM97]. Goal expressions consist of predefined Act metapredicates applied to a logical formula built from predicates specified in first-order logic, connectives, and Act names. The predicates of the expressions describe possible goals and beliefs of the system. These goals and beliefs are interpreted in the same manner as they would be in a BDI architecture. *Metapredicates* allow for the specification of many modes of activity to include goals of achievement, maintenance, and testing [WM97]. Once defined, a goal expression specifies both the applicability conditions for environment conditions and subgoals for plot nodes. Metapredicates such as TEST or ACHIEVE can be used in conjunction with logical formula to specify wants or needs of the system. For example, (TEST P) in the precondition

means P must be true in order for the Act to be applied, while (ACHIEVE G) means the system must currently have G as a goal in order for the Act to be applied.

Figure 6 is an example of an Act describing an operator for deploying an air force to a particular location. The environment conditions are listed on the left side of the screen and the plot nodes (discussed below) are on the right. When the overall system achieves the goal of executing this deployment, the Cue will be invoked.

The precondition ensures various constraints on the intermediate locations get enforced, while the setting is used to look up the cargo to send by air and sea for this particular deployment.

The plot (right side of figure 6) shows the activities for accomplishing the purpose of an Act and consists of a directed graph whose nodes represent actions and whose arcs impose a partial order of execution [WM94]. A plot contains one start node and one or more termination nodes. Each plot node also contains a list of goal expressions (subgoals) that must be satisfied for the action to take place. Plots are composed of conditional (displayed as rectangles) and parallel (displayed as rounded squares) nodes. Arcs coming into and going out of parallel nodes are conjunctive and must be executed, while arcs coming into and going out of conditional nodes are disjunctive and only one arc needs to be executed. If a conditional node has multiple successor nodes, the system will execute all successor nodes until one is found whose goals are satisfied. Execution will now commit to that branch in the tree and will ignore all other branches. Completion of a plot requires the successful execution of all nodes along a given branch of plot from the start node to some terminal node. A significant advantage Act has over many other plan representation languages is the Act-Editor. The Act-Editor, developed by SRI, is a graphical tool used for the interactive viewing, creating, modifying, and verifying of Acts. This offers the user not only a graphical user interface for the creation and modification of plans, but more importantly, a way to verify their correctness.

DEPLOY-AIRFORCE

Cue:

(ACHIEVE (DEPLOYED AIR.1
AIRFIELD.2 END-TIME.1))

Preconditions:

(TEST
(AND (LOCATED AIR.1 LOCATION.1)
(NEAR AIRFIELD.1 LOCATION.1)
(NEAR SEAPORT.1 LOCATION.1)
(PARTITION-FORCE AIR.1
CARGOBYAIR.1 CARGOBYSEA.1)
(TRANSIT-APPROVAL AIRFIELD.2)
(TRANSIT-APPROVAL SEAPORT.2)
(NEAR SEAPORT.2 AIRFIELD.2)
(ROUTE-ALOC AIRFIELD.1
AIRFIELD.3 AIR-LOC.1)
(ROUTE-SLOC SEAPORT.1
SEAPORT.2 SEA-LOC.1)))

Setting:

(TEST
(AND (NOT (= AIRFIELD.2 AIRFIELD.1))
(NOT (= SEAPORT.1 SEAPORT.2))))

Resources:

- no entry -

Properties:

((AUTHORING-SYSTEM SIPE-2(CLASS
OPERATOR))

Comment:

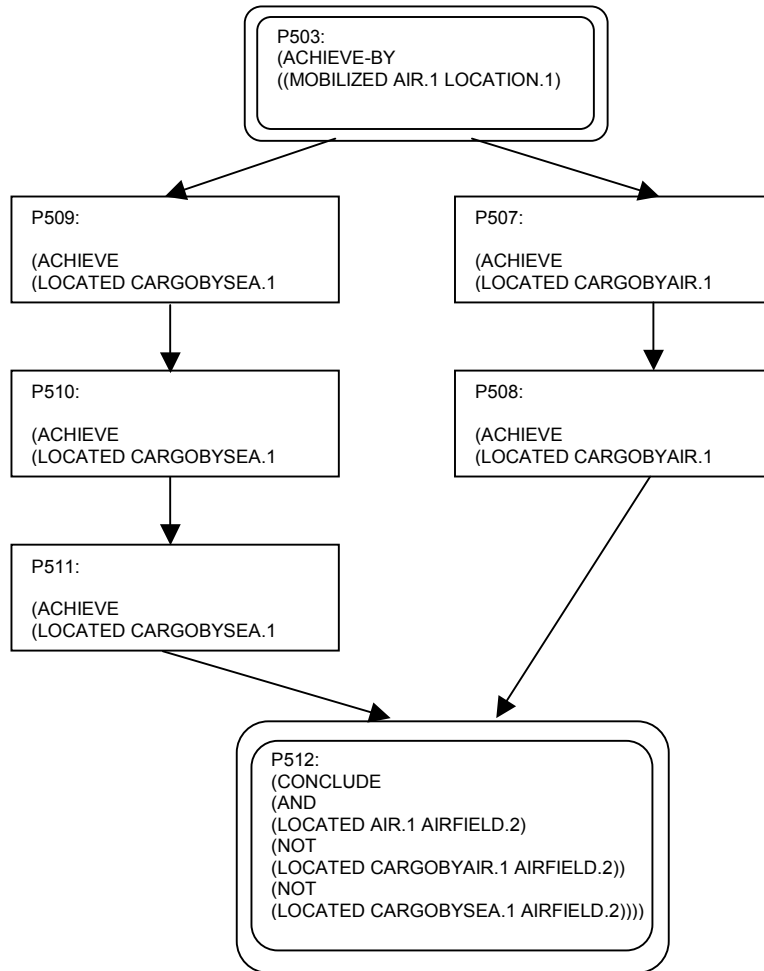


Figure 6 Act diagram example [WM97]

2.3.7 Multi-Agent Communication

When dealing with multi-agent systems, an area of critical importance is communication. Communication is the basis for interaction and organization without which agents would be unable to cooperate, coordinate, or sense changes in their environment. There are three forms of agent communication that may take place: agents may communicate directly with other agents via message passing, agents may communicate with the environment using sensors and effectors, and

agents may communicate with other agents by effecting changes to the environment using effectors.

2.3.7.1 Message Passing

A multitude of languages exist allowing for the communication needs of agents. Telescript, Java, Tcl, and Python name just a few. These language capabilities combined with infrastructural capabilities (such as CORBA) form complete agent communications mechanisms [KJ98]. However, because more than just agent state and simple messages may need to be passed, more is needed. Because agents act on complex rules and ontologies, specific protocols need to be defined for message passing between agents. The Knowledge Query and Manipulation Language (KQML) from the Defense Advanced Research Project Agency (DARPA) and the Agent Communication Language (ACL) from the Foundation for intelligent Physical Agents (FIPA) represent the two most utilized representations to date. These languages provide the designer with a standard syntax for forming messages as well as a number of performatives that define the intent of the message. A *performative* is a keyword that designates the type of illocutory act associated with a message. Using these agent communication languages allows agents to share knowledge and information to cooperate with each other to perform complex tasks. These languages used in conjunction with the aforementioned communication protocols provides a means for agents running on disparate systems to communicate effectively and efficiently.

Regardless of communication format, the overall message-passing model can be simplified to two methods; *send* and *receive*. Although explicit details of how these methods are implemented will be strictly dependent on the type of messaging protocol being used, they do have a number of general characteristics. An agent *send* function will generally take a formatted

agent message and extract specific “address” information in order to deliver the message to the correct agent. This may consist of an internet protocol (IP) address or could simply be a name that can be interpreted by the agent. *Receive* functions are generally event driven and are invoked by the act of a message arriving from another agent. What actually happens once the *receive* method is called can be specified by the designer of the system.

2.3.7.2 Communicating with the Environment

Communication does not only take place between agents, but can also occur between an agent and its environment. Agents can be implemented with sensors and effectors to allow the agent to interact with its surroundings. *Sensors* are required in order for an agent to interpret changes in the environment, while *effectors* are used to impose a change to the environment. Both sensors and effectors may be implemented in hardware or software but will normally have a software interface to the agent itself.

Agent *sensors* can be either functional or persistent. *Functional* sensors allow an agent to get a piece of information on demand. An example of this type of sensor could be used by an agent system deployed on a satellite. Periodically temperature readings may need to be taken and reported back to earth. The agent would query the temperature sensor and then send a message containing the results. *Persistent* sensors are those that perceive changes in the environment and continuously update the agent. Using the example just stated, the same agent system may need to keep track of the internal temperature of the satellite. If certain components overheated they could be permanently damaged. For this reason, the sensor would continually update the agent with internal temperature readings of the satellite components.

Agent *effectors* can be looked at as the actions that an agent may perform on its environment. If an agent system has a robotic arm, it may use this arm to make changes to its surroundings.

2.3.7.3 Communicating Through the Environment

A less direct model of agent communication has to do with one agent effecting a change on the environment and other agents perceiving this change and acting accordingly. Instead of sending a message directly to another agent, an agent may use its effectors to change the environment in a way that can be perceived and interpreted by the sensors of other agents. This method is not as effective as message communication since it assumes that all agents will interpret this environmental change in the same manner. However, it does offer a significant advantage if message passing capability between agents is lost but coordination between agents is still required.

2.3.8 Architectural Summary

This section reviewed some of the most popular architectures currently being used in the field of agents. Although seemingly disparate, many similarities exist in the types of knowledge represented in each architecture.

Not explicitly addressed in any of the architectures but of critical importance in agent systems is communication. If a representation language is to be used in any type of multi-agent based system, some means must exist to represent how agents communicate with other agents as well as their environment. The majority of agent architectures designed on paper or implemented in the real world utilize some form of communication.

2.4 Internal Agent Representation

In order to completely describe the internal behavior of an agent a representation language is needed. Knowledge representation languages in artificial intelligence and software engineering are not new, and a great deal of research has been done concerning this issue. It would be nice if natural language could be used as a knowledge representation language since it is well known and for the most part well understood. However, a number of issues regarding the use of natural language quickly dispel any thoughts of its use. Bigus and Bigus [BB97] point out that natural language is very expressive, but is more appropriate for communication than representation since the meaning of natural language is very dependent on the context in which it is spoken or written.

A *formal language* is a language having a vocabulary, syntax, and semantics that are formally defined and usually have a mathematical basis. Ideally you want a knowledge representation language combining the advantages of both natural and formal languages.

An issue that must be addressed when selecting a knowledge representation language is determining the type of representation scheme needed. *Knowledge representation schemes* categorize the general types of knowledge represented in a program. Two schools of thought exist: declarative and procedural [WIN75].

2.4.1 Declarative Representations

Architectures whose internal representation is based on *declarative knowledge* are formatted in a way that may be manipulated, decomposed and analyzed by a reasoning engine independently of the content. Declarative knowledge deals with representing knowledge as a set of facts. The most well known declarative representation is logic. A significant advantage of a declarative representation is the ability to use knowledge in ways the system designer did not

foresee. Since a piece of knowledge is not inextricably linked to the manipulation of that knowledge, multiple uses can be obtained from a single statement. Ease of modification also represents a major advantage of the declarative approach. A single fact may be easily added, deleted, or modified without affecting any other facts being stored. A final advantage of the declarative approach described by Winograd is that much of the information we know can be more easily stated using declarations. He states that human communication in general consists of breaking information into statements that are passed from person to person.

2.4.2 Procedural Knowledge Representation

In contrast to declarative knowledge, which can be read and modified by humans and all processes that know the format, *procedural knowledge* is in a machine-optimal representation that cannot be read or modified by humans or other internal processes. Procedural representations are characterized by their ability to store knowledge in a faster-to-access but less flexible format.

An advantage procedural representations have over declarative is the representation of second order or metaknowledge. “In applied AI, metaknowledge is that part of the knowledge which the system has about its internal knowledge.” [PB94] In its most general terms, metaknowledge is knowledge about the domain knowledge. This is an important concept since it essentially gives the ability to acquaint a system with what it knows.

2.4.3 Declarative and Procedural Summary

The arguments for declarative and procedural approaches have been made, but there is no clear winner. Genesereth and Ketchpel [GK94] believe that although functional representations may be winning the battle due to their familiarity, declarative representations will overall win the war. Genesereth and Ketchpel are referring to agent communications languages but the statement seems just as applicable in knowledge representation languages.

Once all aspects of the problem have been accounted for, the basic approach to be taken is to look at the overall problem and try to determine if the knowledge needed is more easily represented declaratively or procedurally. Once a scheme has been chosen, the next decision to be made is the language that it will be implemented in.

2.4.4 Languages

The majority of research done in knowledge representation languages fall into four major areas: logic, semantic networks, frames, and production systems.

2.4.4.1 Logic

Formal logic was one of the first attempts to model knowledge unambiguously. It is one of the most studied and best-understood schemes to date [PB94]. Based on predicate calculus, logic is used primarily to communicate declarative rather than procedural knowledge. The ability to reason over sentences is obtained by using axioms and inference rules that specify how new sentences can be derived from given sentences [HODG91].

The advantages of using logic are its clear semantics and expressiveness [REICH91]. Having a clearly defined mathematical semantics ensures a lack of ambiguity in all expressions. The expressiveness of logic can be seen in its ability to express incomplete knowledge. First order logic “determines not so much what can be said, but what can be left unsaid.” [REICH91] This means that details not yet known do not have to be represented. There are numerous logics to choose from. One may use temporal logic to represent and reason about time or epistemic logic to represent and reason about knowledge and belief.

Since logic represents declarative knowledge it also has all of the advantages and disadvantages discussed in Section 2.4.1. In spite of its limitations, logical representation is still

one of the most studied knowledge representation languages for use in artificial intelligence [PB94]. Three well-known logic-base knowledge representation languages are surveyed next: the Object Constraint Language (OCL), Z (pronounced “zed”), and the Knowledge Interchange Format (KIF).

2.4.4.1.1 Object Constraint Language (OCL)

OCL was designed to be used with the Unified Modeling Language (UML) to describe constraints on object-oriented models [WK99]. Rumbaugh et al. [RUM91] defines a *constraint* as a “functional relationship between entities of an object model” while Booch [BOO94] defines it as “the expression of some semantic condition that must be preserved.” Instead of focusing on any one definition, “OCL tries to express the common factor, thereby setting a standard that is understandable and easy to use and allows the modeler to specify what is necessary.” [WK99]

OCL is considered to be a “side effect free” language since OCL expressions cannot change anything in a model. An OCL expression can never change the state of the system even though an OCL expression can be used to specify such a state change (e.g., in a post-condition) [OCLW99]. OCL is also considered to be a “reasonably” formal language, since although OCL is based on well-known and defined set-theoretic concepts, a formal semantics has not yet been defined. Because of its basis in these concepts, it is not considered a difficult task to define the semantics [KWC98]. A metamodel for OCL has been created that defines the syntax of all OCL concepts such as types, expressions, and values in an abstract way and by means of UML features [RG99]. This also aids in bringing OCL closer to being considered a completely formal language. Many arguments exist for not using formally based languages due to their inability to be written and read by non-mathematical experts. Unlike many formal languages, OCL is touted as being “a formal language, which remains easy to read and write” [OCLW99]. OCL has a

familiar look and feel to it that can easily be learned by anyone familiar with programming notations [KWC98]. Using OCL with traditionally non-formal graphical models provides the precise, unambiguous information normally lacking in graphic based models. OCL was written to give users the power of a formal language as well as an easy to read and write format. Because OCL is a modeling language, it is not possible to write program logic or flow control and therefore, it is not directly executable.

OCL expressions are used to specify pre- and postconditions of operations and methods [WK99]. As is normally the case, a precondition must be true at the moment that the operation is going to be executed while a postcondition must be true at the moment that the operation has just ended its execution. Besides pre- and postconditions, OCL is also used to define invariants. Used in this context, an *invariant* is a constraint that states a condition that must always be satisfied.

OCL has declaratively defined semantics and contains a large number of predefined data types to represent collections, enumerations, strings, booleans, integers, and reals. OCL has the predefined types **set**, **bag**, and **sequence** to work with collections of objects and supplies a wide range of predefined operations ranging from manipulation to checking the status of these collections. OCL is also very robust in terms of how it handles characters and strings. Concatenation, size, substring, equals, and not equals operations allow a user the flexibility to handle most string and character problems that may be defined. An equally robust set of predefined operators is also seen in dealing with **boolean**, **integer**, and **real** types. Besides the “standard” data types just mentioned, a user also has access to a number of unique data types particular to OCL. **OclType** refers to all predefined OCL types as well as any type defined in a UML model. Access to this type allows a modeler access to the meta-level of the model. For example, **type.attributes** returns the set of names of the attributes of **type** as defined in the UML

model (where `type` is the instance of `OclType`). `OclAny` is the supertype of all types in a model. All classes in a UML model inherit all of the features defined on `OclAny`. Like the previously mentioned types, a useful set of predefined operations exists for each type.

A significant difference between OCL and many other modeling languages is the fact that it is dependent on predefined graphical models. In order to use an attribute or method, that attribute or method must first be defined in an object oriented (UML based) model. This offers a significant advantage in that the model can easily be viewed from varying levels of abstraction. The graphical model alone offers a “big picture” view of the system while the OCL offers the ability to specify precise details about the system.

Another advantage offered by using OCL in conjunction with UML is the ability to model communication between objects using events, signals, and messages. In UML, an *event* is something that occurs in the system or environment that the system must react to and handle. Four different types of events exist in UML: a condition becoming true, receipt of an explicit signal object, receipt of an operation call, or passage of time [EP98]. A *signal object* is sent from one object to another and may contain attributes and operations. Events may occur at the software or hardware level. [MASE99] and [COOL95] both address agent communication by using event diagrams to model agent coordination. *Signals* are a special case of events. In UML, signals are defined as named events that may be raised. A signal is described as a class (with the `<<signal>>` stereotype) representing an event that occurs in the system. Signals are passed as messages synchronously or asynchronously between objects in a system. *Messages* are used to communicate between passive objects, active and passive objects, or between active objects. Messages are implemented by operation calls or as signal objects placed in a mailbox or queue [EP98]. In real-time systems, message receipt is normally considered an event. To depict

communication between objects, messages are shown in the following UML diagrams; sequence, collaboration, state, and activity.

A simple example of how to specify a problem using OCL would be modeling non-military and military students at a university. Both military and non-military students contain the following attributes: first name, middle initial, last name, date of birth, social security number, gender, height, weight, and age. All students have a grade point average (GPA) that is between 0.000 and 4.000. Persons in the military have an associated rank and must also meet weight requirements. Military students must be between the ranks of 1LT and Capt, must be under the age of 35, and must maintain a GPA greater than or equal to 3.000. The object model shown in Figure 7 captures the main classes needed, their relationships to one another, and the attributes of each class (there are no methods needed for this example). Figure 7 shows the main class Person contains all of the attributes shared by a student and military member. Student inherits all of those attributes and adds the attribute GPA. Military inherits the same attributes as student but adds the attribute military_rank. MilitaryStudent inherits all of the attributes from Student, Military, and Person. OCL can now be used to specify the constraints needed. The context to which an OCL expression applies is always underlined and is followed by OCL expressions. The constraints shown below satisfy the original conditions specified in the problem.

```
Student  
GPA >= 0.000 and GPA <= 4.000
```

```
Military  
weight <= 3*height
```

```
MilitaryStudent  
military_rank = 1LT or 2LT or CPT  
gpa >= 3.000  
age <= 35
```

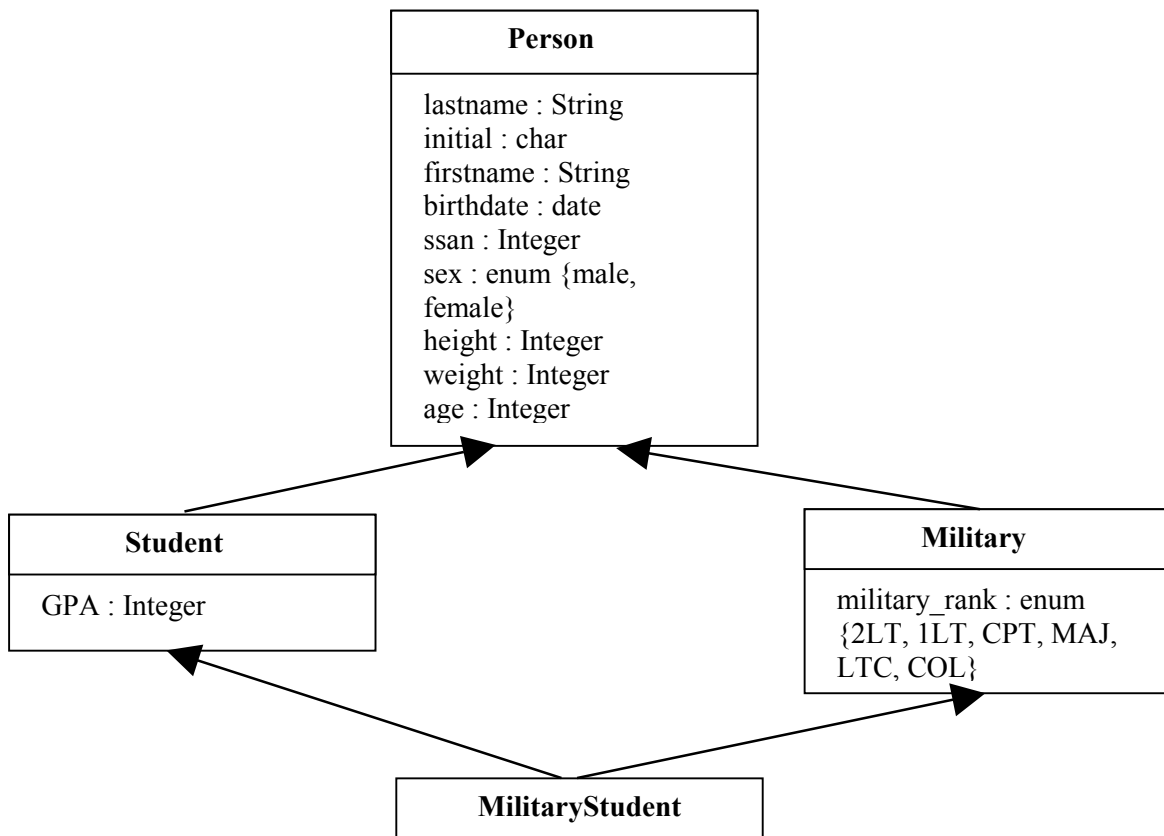


Figure 7 OCL Object Model

A problem often seen when using any type of graphical model in a knowledge representation language is finding a way to represent the model so that the computer can interpret it. A parser has been developed by IBM to read in the models and OCL using a simple syntax. Type declarations start and end with the keyword `<type` and can be followed by optional supertypes starting with the key word `<supertype`. An example of how to convert the Person and Student classes for Figure 7 is shown below:

```

<features
  <type Person
    lastname : String;
    initial : char;
    firstname : String;
    birthdate : date;
    ssan : Integer;
    sex : enum {male, female};
  
```

```
height : Integer;  
weight : Integer;  
age : Integer;  
  
<type Student <supertype Person  
GPA : Integer
```

The constraints themselves are written in the standard OCL syntax described earlier and are stored in a separate file. The object model along with the OCL constraints allows the problem to be specified in an easy to read unambiguous manner that can be easily parsed and interpreted by a computer.

2.4.4.1.2 Z

The formal specification notation Z (pronounced “zed”) is based on Zermelo-Fraenkel set theory and first order predicate logic. It has been developed by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL) and elsewhere since the late 1970s. Z offers essentially the same expressive power seen in OCL. Luck and d’Inverno [LD95] provide three primary arguments for the use of Z in agent based systems.

1. Z allows for precise and unambiguous definition of common concepts and terms in a readable way.
2. Z is sufficiently expressive to allow for a consistent, unified and structured account of a computer system and its associated operations.
3. Z provides a foundation for development of new and increasingly more refined concepts.

Given the reasons cited by Luck and d’Inverno, one would think that Z would be a universal standard for knowledge representation in AI as well as software engineering. One of the reasons Z has not been globally adopted is that although Luck and d’Inverno find Z very readable and easy to use, most of the rest of the general population does not. Z is based on first order logic, which in itself is mathematically very simple. The problem comes when complex information must be represented and/or manipulated. The syntax and semantics of Z tend to collapse everything together making it difficult to correctly interpret the specification even

though it has been formally written. Another problem is that computers do not easily represent much of the notation needed to write Z specifications and the shortcut notation needed by most interpreters makes the knowledge virtually unreadable in a non-compiled format. Point two and three made by Luck and d’Inverno are very strong and make Z a viable contender when choosing a representation language, as long as all parties involved are aware of the overhead that comes along with using it. Using the same student example as seen in Section 2.4.4.1.1, a Z specification can be shown from start to finish. Figure 8 shows a Z depiction of all of the attributes, constraints, and associations specified in the original problem. From this simple example it is easy to see a small problem can quickly become unmanageable using Z as a specification language. A problem not brought out by the example is the fact that Z constraints are difficult to understand and ambiguous at first glance.

2.4.4.1.3 Knowledge Interchange Format (KIF)

Another alternative to OCL or Z is KIF. KIF was developed as part of the Advanced Research Project Agency (ARPA) sponsored Knowledge Sharing Effort to aid in the sharing and reuse of knowledge. KIF was specifically designed to be an “interlingua”, which is essentially a mediator to translate other languages.

KIF is a prefix version of first order predicate calculus that added a number of extensions to support non-monotonic reasoning and definitions. Although not specifically designed as a knowledge representation language, KIF possesses all of the attributes one would want in a language. KIF has declarative semantics giving the advantages discussed in Section 2.4.1. KIF is also logically comprehensive in that it provides for the expression of arbitrary sentences in the first-order predicate calculus.

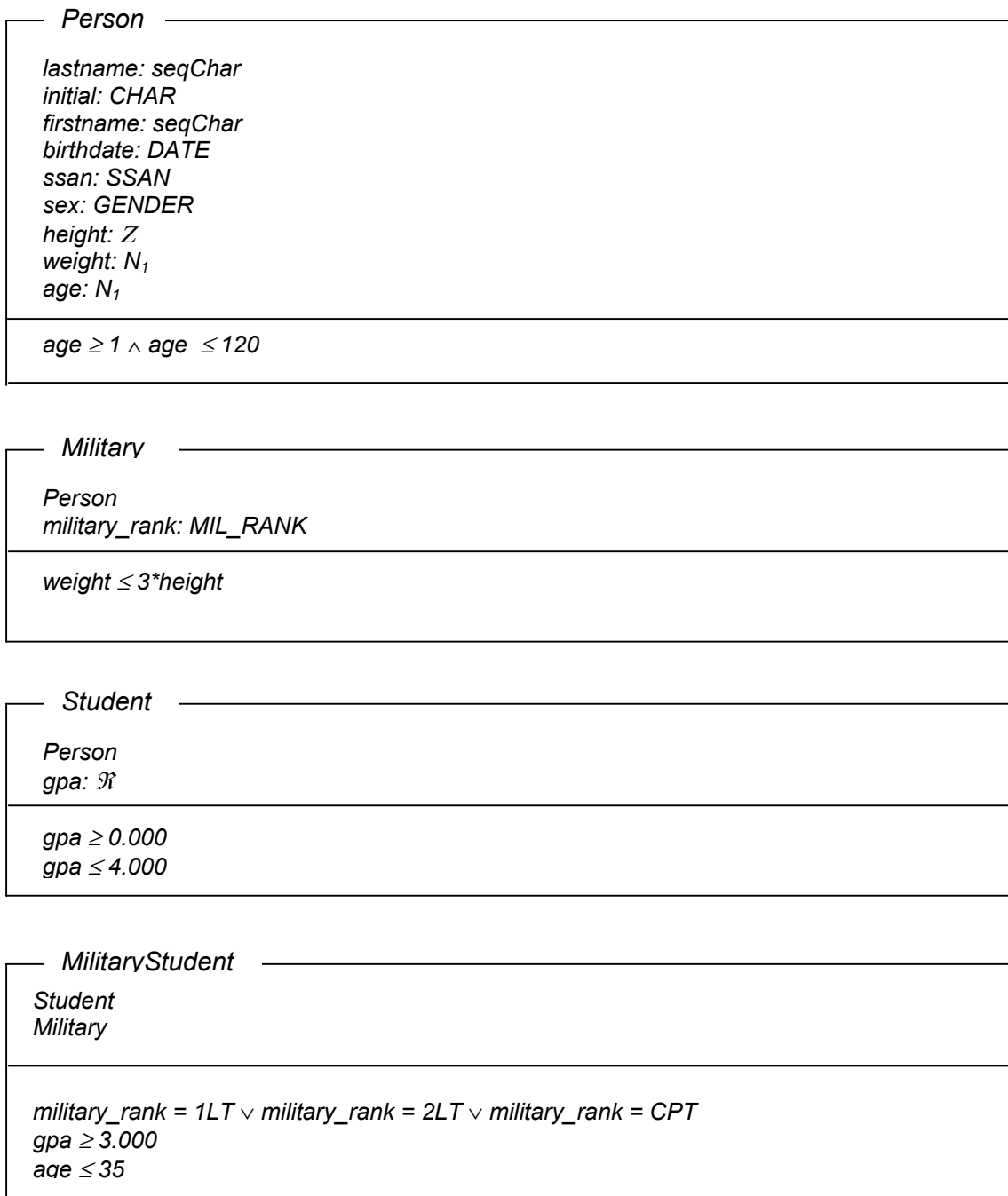


Figure 8 Z Representation

The language provides for the representation of knowledge about knowledge, allowing a user to make representation decisions explicit, as well as permitting users to introduce new representation constructs without changing the language. In terms of representing logic, KIF can handle all standard logical sentences containing any combination of negations,

conjunctions, disjunctions, implications, equivalences, existential quantifications, and universal quantifications. KIF is also very expressive in its ability to represent numbers. KIF has predefined types for integer, real, and complex numbers as well as an extensive list of operators and functions that can be performed on each type. KIF's ability to represent sequences and sets is also robust enough to deal with most problems a user may want to specify. A downside to KIF is its lack of functions for dealing with characters and strings. Characters and strings are easily represented in KIF, but there are no functions for manipulating these strings once created.

Another disadvantage of using KIF is that knowledge must be represented in prefix notation, which does not always lend itself to readability and understanding. A possible solution to the problem is using infix KIF. Every expression written in infix KIF can be translated to a logically equivalent expression in prefix KIF. The opposite is not true in that every prefix KIF expression cannot be translated to infix.

The student example used for OCL and Z cannot be easily represented in KIF since KIF offers no predefined constructs for handling object-oriented models. A KIF representation of the internal structure of the person class and its constraints is shown below.

```
string lastname
char initial
string firstname
integer birthdate
integer ssan
string sex
integer height
integer weight
integer age
(and (>= age 1)
     (<=age 120))
(>= (height 1))
```

Although not as challenging as Z, even moderately complicated expressions can quickly become difficult to read even to those very familiar with prefix notation.

2.4.4.2 Semantic Networks

Semantic networks allow for the graphical representation of relationships based on patterns of nodes that are interconnected by arcs. Knowledge is depicted in labeled, directed graphs by the *nodes*, representing the objects or concepts, and the *arcs* representing the relationships between the elements. An important point to make is that no information is stored in the nodes themselves; all knowledge is represented by the links between the nodes. Semantic networks offer the advantage of being represented graphically as opposed to the textual representation of Z or KIF. The biggest disadvantage seen in using semantic network representations is that there is no formal semantics of what a given representational structure means. A partial representation of the student example of Section 2.4.4.1.1 using a semantic network is shown in Figure 9.

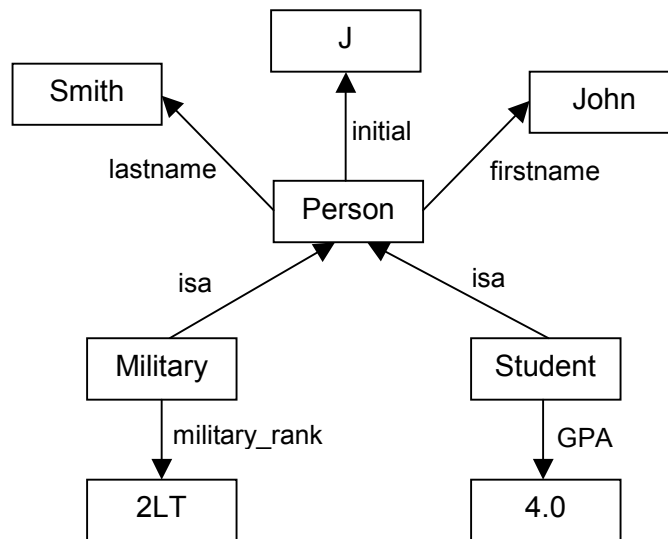


Figure 9 Semantic Network Example

The most well known semantics network system is the Semantic Network Processing System (SNePS) [KM98, SHAP99]. The OK BDI architecture [KS94], is an evolved version of SNePS based on the BDI formalism.

2.4.4.3 Frame-Based Systems

Frame-based systems in AI are usually composed of mutually linked frames that are ordered hierarchically. Frames extend semantic networks by giving the ability to organize knowledge in a hierarchical and descriptive manner. An individual frame may be viewed as a “record” data structure containing information relevant to specific entities. A frame is composed of *slots* that contain attributes of the object being represented by the frame itself. The slots of the frame may contain the following information [LS98]:

1. Frame identification
2. Relationship of current frame to other frames
3. Descriptors of requirements for frame match
4. Procedural information on use of the structure described
5. Frame default information
6. New instance information

The need for each of the slots and the amount of detail needed in each is dependent on the particular situation being addressed. The ability to attach procedures to frames is important since it gives the ability to represent knowledge not easily represented in a declarative format. Procedures called demons are invoked as side effects of some other action in the knowledge base [LS98].

An advantage seen in using frames is its ability to represent declarative as well as procedural knowledge [PB94]. General frames allow knowledge that is common to a class of objects to be stored while specific frames describe specific objects. Frames allow for an object-oriented approach to knowledge representation where frames represent objects, slots represent attributes, and daemons represent the methods of an object. The use of general frames as well as inheritance has made frames one of the most efficient tools for knowledge representation [PB94]. Although not intuitively obvious, both semantic nets and frames are closely related to predicate logic. Russell and Norvig [RN95] describe a procedure to transform both semantic networks and frames into first-order logic [BB98]. It is because of this that semantic networks and frames offer all of the major advantages and disadvantages associated with logic. The primary down side seen to using frames is the lack of a formal theory for representing the knowledge. This fact has significantly limited the use of frames in AI systems. A representation of the student example of Section 2.4.4.1.1 using a frame representation is shown in Figure 10.

2.4.4.4 Production Rules

Production rules are a knowledge representation formalism consisting of two parts, a left hand side and a right hand side. The conditional part (the left side) is used to show when a rule should be applied. The transformational part (the right side) shows what actions should be taken when the rule is applied. Production rules are used extensively in expert systems since expert knowledge can be easily represented using these IF-THEN types of rules. The greatest advantage to using production rules for expert systems is the modularity offered by representing data in this manner. Rules can be added, modified, or deleted independently without effecting the rest of the knowledge base. A complete description of these systems is outlined in Section 2.3.4.

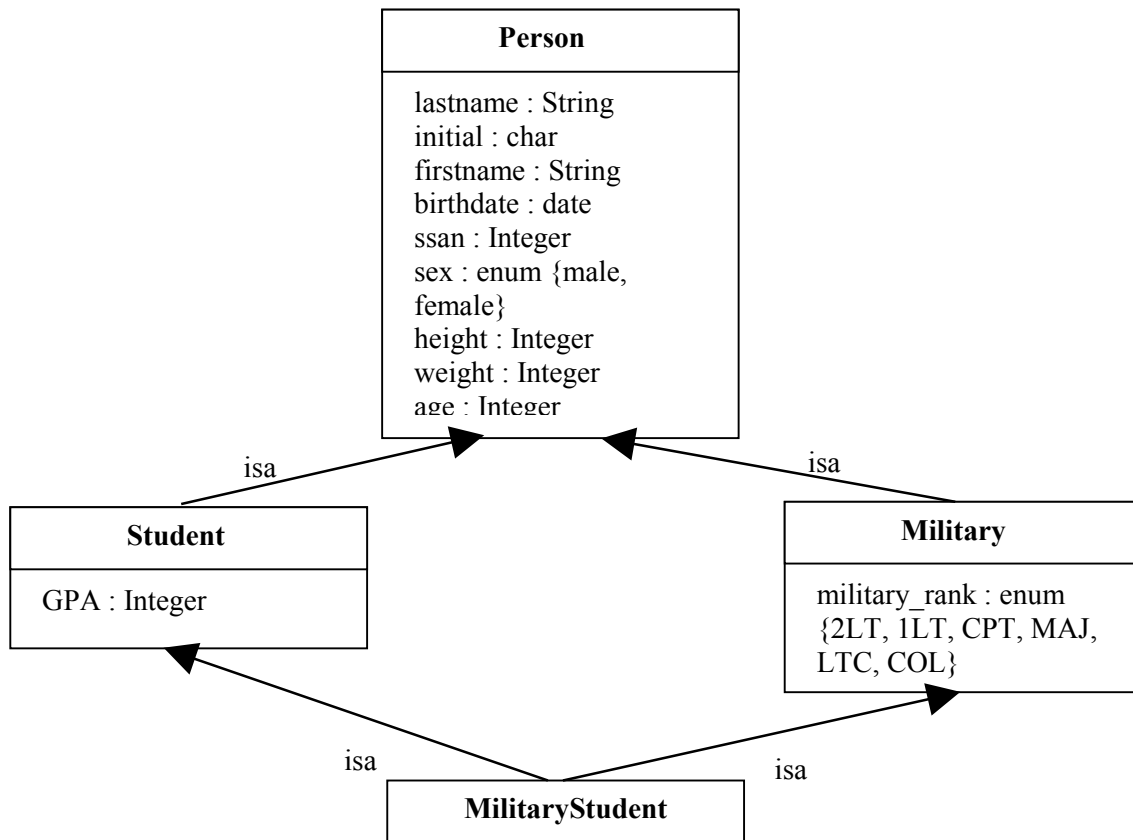


Figure 10 Frame Example

The main drawback to using this type of representation is the limited scope in which these systems can be used.

2.4.5 Section Summary

Although arguments have been made for the use of declarative and procedural representation schemes, it seems that of most research is directed toward the declarative approach. The stated benefits along with a formal foundation allow agent systems to be specified easily and unambiguously. Logic languages represent the most direct and implementable way of

representing declarative knowledge. The majority of current agent implementations use some form of first order logic for the representation of certain knowledge.

2.5 Summary

Knowledge representation is needed in all AI systems in one aspect or another. Despite the research that has been done, a formal knowledge representation standard still does not exist. Many specific representation languages have been defined for specific architectures, but no effort has been made to define a language that will work with multiple architectures. By examining agent architectures and looking for common knowledge representation components, a generic language can begin to take shape. Continued research is needed to review past and present work in order to determine a way to effectively incorporate knowledge representation and formal methods.

III. Approach

As stated in Chapter 1, the goal of this research is to create a precise specification language for defining the structure and behavior of agents in a multi-agent development environment. When developing a representation language for use in a multi-agent (or any software) system, a number of considerations must be taken into account. This chapter describes the approach followed in developing the language. Figure 11 depicts this approach graphically.

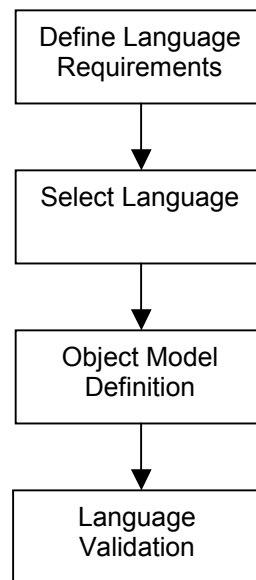


Figure 11 Approach

Section 3.2 defines how requirements are chosen for the language to specify agents in a multi-agent system. Section 3.3 outlines the process of selecting a language based on the requirements of Section 3.2. Section 3.4 details how the problem syntax and semantics are defined and Section 3.5 outlines the process of creating templates to test the language.

3.1 Language Requirements

Figure 12 depicts the first step of the approach.

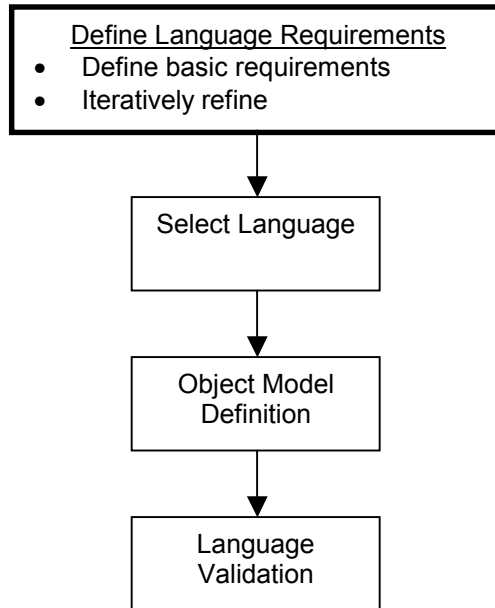


Figure 12 Step One

The first step in defining a knowledge representation language for a multi-agent design system is to determine all requirements the language must satisfy. This thesis follows an iterative refinement approach in developing the requirements. The first step is to look at the problem domain and try to determine the most basic requirements the language must satisfy. For example, if the language is to be used to specify multi-agent systems, a basic requirement may be the ability to define an agent. Basic requirements may also be “pre-defined” by users or designers of the system. An example of this could be that users want to interact with the system using a graphical interface. An extensive review of literature should also be accomplished since the development of specification languages is well researched and a number of known issues may have already been identified. Once all basic requirements are determined, each of these should then be re-examined to decide if a lower level of requirements exist. Building off the above

example, an additional requirement might be that agents should be viewed at multiple layers of abstraction. For example, an agent may need an external graphical view as well as an internal textual view. This refinement process is repeated until a complete listing of language requirements is developed.

The importance of this first step should not be underestimated since it forms the foundation from which the language will be built. Once completed, this first step is repeated at least once more to verify the completeness of the list of requirements.

3.2 Selecting a Language

Figure 13 depicts the second step of the approach.

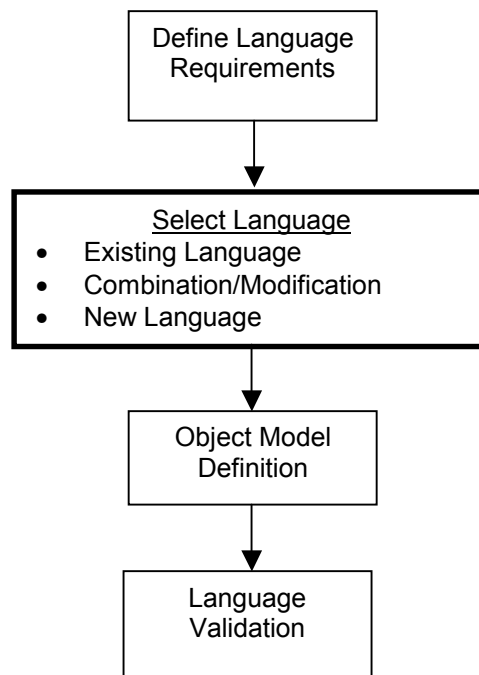


Figure 13 Step Two

Once the requirements for the language are specified, the next step is to choose a language or languages to satisfy them. The three options available are to choose an existing

language, modify or adapt an existing language, or define a new language. Of these choices, using an existing language offers the most potential advantages, which include documentation, wide use and acceptance, and rigorous testing. Because of this, the first step in this portion of the approach is to examine as many existing languages as possible. Languages used to solve similar types of problems should be examined first. In regards to agent specification, the two best areas to concentrate on are artificial intelligence and software engineering. Both fields address the software specification problem at varying levels of abstraction and a large amount of research has been focused in this area. Documentation and literature of each language should then be carefully reviewed to determine the requirements the language will satisfy. The best approach is to make a table listing all language requirements and all languages examined. It is then possible to check off which languages meet which requirements. If no one language can meet all requirements, the next step is to see if multiple languages can be used to meet the requirements. Often, different aspects of the same problem may need different languages to satisfy all the requirements. Using the table, it is easy to see which languages may be used to do this. If numerous languages don't meet all requirements, then the next step is to examine the requirements not met. Choose the language or languages satisfying the most requirements and then examine the specific requirements that still need to be met. It may be possible to modify or extend an existing language so that it is able to meet the remaining requirements. If this is not possible, then the last resort is to define a new language.

Defining a new language is by far the most undesirable solution but must be addressed in case all other attempts fail. [FMS97] states that "at least 90% of the next 700 formalisms for reasoning about agents will have no impact whatsoever on the development field." A serious problem currently being faced in software engineering is one of standardization. Software engineering is a relatively new discipline and as such, contains very few (if any) agreed upon

practices in the area of software development. Writing a new language tailored to a specific problem will solve the problem at hand, but at the same time will add to the problem at large. Because of the mentioned downsides, as well as the fact that defining a new language can be a time consuming and difficult task, this option should only be used as a last resort to solving the problem.

3.3 Object Model Definition

Figure 14 depicts the third step of the approach.

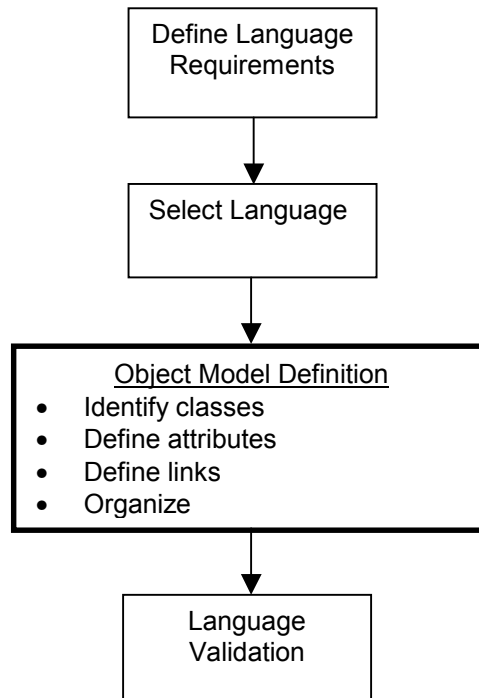


Figure 14 Step Three

Once a language is selected, the problem syntax and semantics must be specified. Defining an object model can do this. Selecting a language that meets all the problem requirements provides a medium to solve the problem, but it does not provide the means. This portion of the approach is based on Rumbaugh's object-oriented analysis (OOA) techniques for

creating an object model [PRES97]. The goal is to separate the problem into its most basic parts so it may be seen what these parts consist of and how they are connected. The steps to follow are shown below.

1. Identify classes relevant to the problem
2. Define attributes and associations
3. Define object links
4. Organize classes using inheritance

An example of using these steps to define a generic solution to specifying agent architectures follows. Using Maes definition (Section 2.3) of an agent architecture, the relevant classes identified in step one consist of components and connectors. Figure 15 shows an object-oriented representation of these classes along with the associations between them. Definitions of attributes are left out of this example for simplicity, but would normally be done at this time.

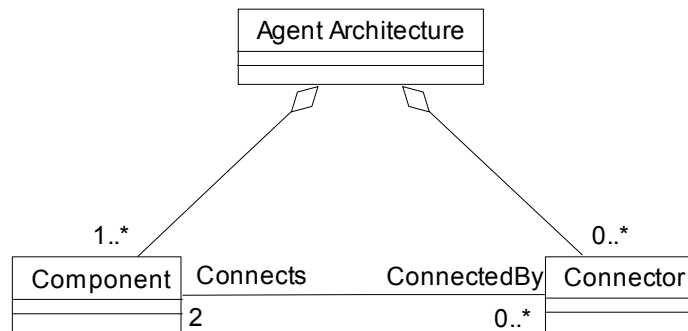


Figure 15 A Generic Agent Architecture

According to the template, an agent architecture consists of one or more components linked together by connectors. Each component is connected by zero or more connectors and each connector joins exactly two components. Once completely defined, the classes along with their required attributes represent the *problem syntax*. How the classes can be constructed and

connected together to solve the problem constitute the *problem semantics*. Together, the syntax and semantics define the problem object model.

The representation of a truly generic solution is important for a number of reasons, the first of which is extensibility. By defining a template that all solutions are based upon, new components can easily be created and integrated within existing solutions. This leads to another advantage of a generic solution, which is compatibility. Any component built according to the template should be able to be integrated with any other components previously defined.

3.4 Language Validation

Figure 16 depicts the fourth step of the approach.

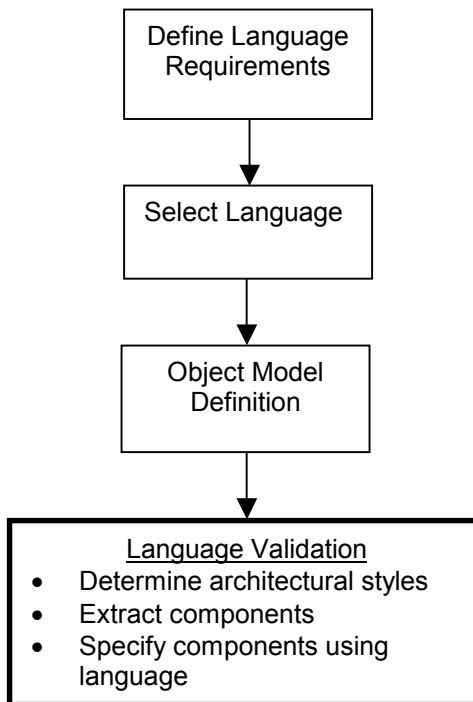


Figure 16 Step Four

When defining an agent specification language, an issue of significant importance is ensuring most agent types can be specified using the language. In order to test for this, it must

first be determined what knowledge is needed to define any agent type. This portion of the approach outlines the testing needed to verify the completeness of the language chosen. A beneficial side effect of the testing is the creation of agent templates that can be used for the design of agent systems.

One methodology to testing the language is to look at all multi-agent implementations and try to extract required agent knowledge from each. This information could then be grouped according to function and made into generic components that could be used to construct agents. If the language chosen could specify all of the components, then it would obviously be a wise choice. While this proposal may seem like an obvious answer, it is definitely not the easiest. Besides the fact that there are an extremely large number of agent implementations in existence at this time, this is more a brute force solution than one based on sound engineering principles. In order for a component to be a reusable asset, more must be done than just taking a monolithic design of complete solutions and then just breaking it down into fragments [CS98]. Descriptions must first be carefully generalized in order to allow reuse to be possible in many contexts. Looking at the problem at a higher level of abstraction, another approach is to examine the more generic agent architectures from which the implementations were derived. Using the Maes definition of agent architecture from Section 2.3, it should be possible to decompose each architecture into a set of components and connectors, which can then be specified using the chosen language. A problem seen to following this methodology is the fact that the recent popularity of agents has created a large number of distinct agent architectures. The solution involves looking at the problem at its absolute highest level of abstraction; the architectural style.

An *architectural style* defines a class of systems in terms of a pattern of structural organization. It determines the vocabulary of the components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [SWA94].

To determine the basic agent architectural styles, architectures must be reviewed and categorized according to the below criteria proposed by Garlan and Shaw [GS96].

- Design vocabulary – types of components and connectors
- Allowable structural patterns
- Underlying computational model
- Invariants of the style

To go about this review and categorization in the most comprehensive manner possible, each architecture must be modeled using object-oriented techniques. Object-oriented design and component-based design are closely related to one another. A component is likely to come into existence through objects and will normally consist of one or more classes or constant prototype objects [SZY98]. Modeling each architecture as a collection of classes and associations significantly aids in identifying the basic components and connectors that each is composed of. It also shows the basic structure of the specific architecture.

The first step in decomposing an architecture is to determine the basic classes needed to model the architecture. An example of this using the PRS-CL architecture of Section 2.3.2.1 can be seen in Figure 17.

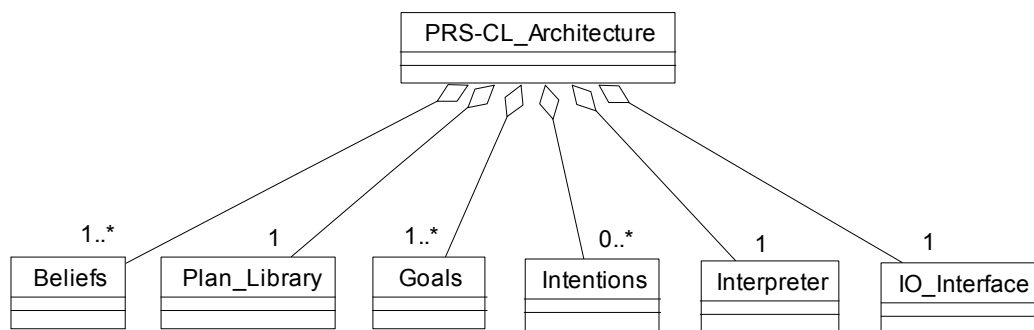


Figure 17 Generic BDI Architecture

As described in the previous chapter, the architecture consists of beliefs, goals, and intentions as well as a plan library, interpreter, and interface to the environment. Each class must

then examined to see if it can be further decomposed. A possible decomposition of the Plan_Library, Goals, and IO_Interface can be seen in Figure 18.

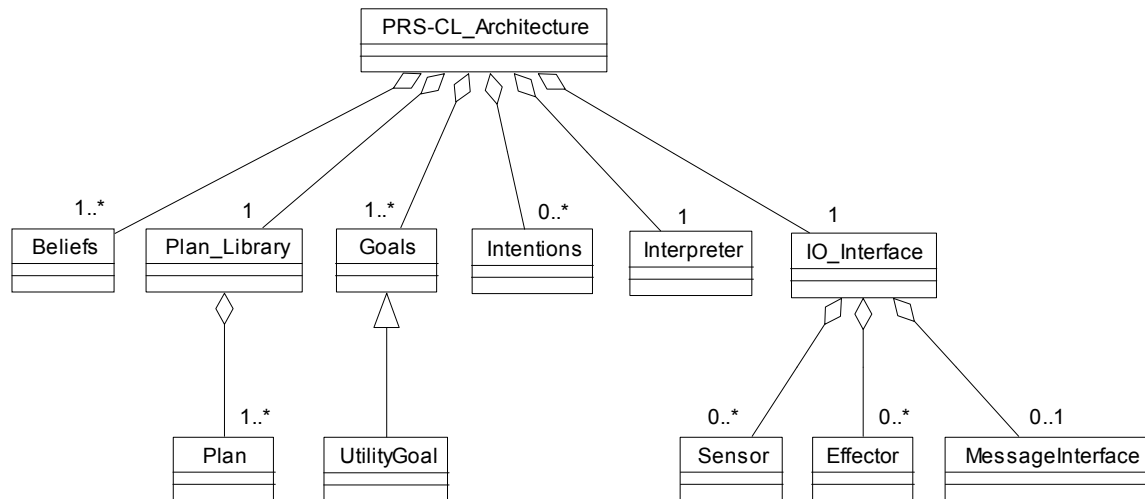


Figure 18 Refined BDI Architecture

This process is repeated for each class until no further breakdown of classes is appropriate or needed. Once all classes are defined, iterative refinement of each class is done to specify attributes, methods, and data structures. Dependencies and interactions between classes are then examined to determine what can be modeled as a component. Using Figure 17 as an example, candidate components are Beliefs, Plan Library, Goals, Intentions, Interpreter, and IO_Interface. The whole process must then be repeated for a different architecture. Architectures should be grouped according to the types of components and connectors they contain as well as how these components and connectors are organized. After reviewing a number of specific architectures, a general set of agent styles should begin to take shape. Once a listing of styles is completed that the majority of architectures belong to, a general set of components are extracted. The components should then be specified in the language selected in Section 3.3 to verify its applicability.

3.4.1 Template and Testing Summary

Determining a set of generic architectural styles and generic components not only allows the language to be thoroughly tested, but it also provides a set of templates a designer can use in creating agents. The templates are not meant to limit the designer in any way, only aid in the rapid design of agent-based systems.

3.5 Chapter Summary

This chapter has outlined the general approach to be followed in order to define a knowledge representation language for a multi-agent system. Section 3.2 discussed the method of determining the requirements that should be imposed on the language. Section 3.3 elaborated the steps to be followed in selecting a language once the requirements have been defined. Although three options were presented, the third was highly discouraged because of significant drawbacks. Section 3.4 defined the method for determining the language object model and Section 3.5 outlined the approach for validating the language through the use of architectural styles.

IV. Design and Implementation

The purpose of Chapter 3 was to outline the generic approach that can be used to reproduce the results of this research. This chapter covers the first three steps of the approach. Chapter 5 is dedicated to the final step of the approach. Section 4.1 defines the requirements the language must possess. Section 4.2 evaluates two existing languages against the requirements defined in Section 4.1. Section 4.3 completely defines the language and Section 4.4 uses this definition to create the language object model. Section 4.5 describes how the language was implemented within the agentTool multi-agent design environment.

4.1 Defining the Requirements

This section of the thesis outlines the language requirements chosen and the reasoning behind their selection. The complete list of language requirements is shown below.

1. The language must precisely and unambiguously provide meaning for common concepts and terms and do so in a readable and understandable manner.
2. The language must allow agents to be specified using a combination of graphics and text.
3. The language must allow for interaction among agents as well as interaction between agents and their environment.
4. The language must support design modularity.
5. The language must allow for the development and viewing of multiple levels of abstraction for a given design.
6. The language must allow for the representation of complex data structures and operations.
7. The language must allow alternative designs of particular models and systems to be presented, compared, and evaluated.
8. The language must allow for the static and dynamic behavior of the system to be captured.

The remainder of this section describes the rationale behind the selection of each language requirement.

The first two requirements were originally addressed as thesis goals in Chapter 1. The first states that the internal agent representation must be specified using a precise and unambiguous language. This requirement is also addressed by Luck and d’Inverno and is covered in Section 2.2.2. The reasoning behind this requirement are the possibility of agent verification, automated code generation, and overall ease of understanding. The second requirement outlined in Chapter 1 states that agents must be specified using a combination of graphics and text so as to aid in understanding and composition of agent systems.

The next requirement was imposed by the multi-agent characteristics of the problem. Because the language will be used to specify agents in a multi-agent system, it is assumed an agent system may contain multiple agents capable of interacting with one another. To allow for this, the third language requirement states that the language must have the ability to represent interaction among agents as well as interaction between agents and their environment.

The next requirements were realized after closer examination of what an agent can represent. Agents can range in complexity from simple reactive agent to complex reasoning agents. Because of this, a number of additional requirements were needed to ensure agents of varying degrees of complexity could be specified. The first of these requirements is derived from Section 2.2.2.2 and states that the language must support design modularity. In order to efficiently and effectively specify agents, the language must allow for large and difficult problems to be partitioned into manageable pieces. The next requirement is derived from Section 2.2.2 and states that the language must allow agents to be developed and viewed at multiple levels of abstraction. As agents become more complex, it may not be desirable or even possible to

represent the internal details of an agent at one level of abstraction. The final requirement in regard to agent complexity is derived from Section 2.2.2.2 and states that the language must allow for elaborate data structures and operations to be represented. Even the most basic agent may need complex data structures to store information and complex operations to manipulate it.

Not all language requirements were extracted by examination of the problem. After reviewing literature on agent specification languages, an additional requirement was realized. Derived from Section 2.2.2.2, this requirement states that a multi-agent language should enable alternative designs of particular models and systems to be presented, compared, and evaluated. This requirement is important when defining a language to specify any type of software system since it provides the designer with options. Just because a design solves a problem does not mean it is the best design. Options give the designer the ability to compare and contrast designs and then choose the most appropriate.

The final language requirement specified was the ability to represent an agent dynamically. Until now, all requirements dealt with the static structure of the agent and have not addressed agent execution. As stated in Section 2.2.2.2, a dynamic representation is needed to address how the structure or system configuration may change due to external events [PRES97].

4.2 Review of Existing Languages

Using the language descriptions of Section 2.4.4.1.1 and 2.4.4.1.2, *Z* and the combination of UML and OCL were evaluated against the requirements specified in Section 4.1. After reviewing a number of existing languages, *Z* and the combination of UML and OCL were determined the best candidates for satisfying the requirements of Section 4.1. A report card of how well each language met the requirements is shown in Table 1. Grades are based on the standard academic grading scheme.

Requirement	Z	OCL & UML
1	B+	B+
2	D	A
3	D	A
4	A	A
5	B	A
6	A	A
7	A	A
8	B	A

Table 1 Language Report Card

An A is excellent, B is above average, C is average, and D is below average. Plus and minus symbols may be added to further delineate levels of good and bad appropriately. Sections 4.2.1 and 4.2.2 review the reasoning behind the grading of each language.

4.2.1 Z

Z is a formal specification language adopted by a number of researchers [DAF97, DMAR97, SZS95] to construct formal agent frameworks. As stated in Section 2.4.4.1.2, Z is based on set theory and first-order logic.

The first requirement for the language was that it be able to represent information precisely, unambiguously, and do so in the most readable and understandable manner possible. Without question Z offers the ability to represent information precisely and unambiguously by having clearly and completely defined syntax and semantics. However, the ability for Z specification to be considered readable and understandable is arguable (Section 2.4.4.1.2), which is the reason the requirement was rated B+.

In regards to the language using graphics and text for representation of information, Z meets the textual portion of the requirement with ease, it is the graphical portion that is lacking. The only graphics used in Z are the lines bordering a schema (see Figure 8). Z may be used in

conjunction with graphical languages such as object-oriented diagrams, but because *Z* was not designed to be used in this fashion, it is questionable if the overall interpretation could still be considered precise and unambiguous, thus further compromising Requirement 1.

Requirement 3 addresses the ability to model agent interaction. A downside to using *Z* to specify agent systems is that it is inappropriate for modeling interactions between agents [FMS97]. When it is necessary to model some sort of communication structure, a formalism such as the Communicating Sequential Process (CSP) algebra may be more appropriate [FMS97]. CSP allows a system to be modeled as a collection of processes that communicate with one another. CSPs lack of use in the realm of multi-agent systems is based on the fact that CSP was developed in relation to distributed processes and the semantics of parallel languages. These are not always well suited to the problems of multi-agent systems, especially those in which agents carry out actions in an environment [JF99].

In regards to modularity, *Z* schemas allow Requirement 4 to be met. As stated in Section 2.4.4.1.2, the schema is the main element in *Z* to decompose a specification into smaller, more manageable pieces.

In a paper written by Luck and d’Inverno, they specifically address the ability of *Z* to satisfy Requirements 5 and 7, the ability to support multiple levels of abstraction and the ability to compare and evaluate a design in multiple ways. They state that through the use of schemas and schema inclusion, *Z* depicts a system description at different levels of abstraction, thus satisfying requirement five [FMS97]. Something not stated by Luck and d’Inverno, is that although multiple levels of abstraction are possible, they are not always beneficial. Because *Z* is a textual language, it is difficult to visualize multiple layers of abstraction. Luck and d’Inverno also state *Z* is expressive enough in representing a consistent, unified, and structured view of a

computer system and its operations to allow for alternative designs of agent models to be presented, compared, and evaluated [FMS97]. This satisfies Requirement 7.

Z's ability to represent complex data structures and operations was addressed in Section 2.4.4.1.2, and is adequate to satisfy Requirement 6.

The final requirement states the language must be able to represent the static and dynamic behavior of a system. Although [ZRM98] states that Z schemas are used to describe both static and dynamic aspects of a system, dynamic representation in Z can be hard to follow. Z offers no "big picture" view of the dynamics of a system, making interpretation difficult.

4.2.2 UML & OCL

Although UML and OCL are two separate languages, OCL was designed for use with UML diagrams. Because of this, UML and OCL were evaluated together against the requirements of Section 4.1.

In regards to representing information in a precise and unambiguous manner, OCL has selected ideas from formal methods and combined them with diagrammatic, object-oriented modeling thus resulting in a precise, robust, and expressive notation. Like Z, OCL is based on predicate logic. As stated in Section 2.4.4.1.1, OCL currently has no complete formal semantics defined, but because it is based on logical and set-theoretic concepts, it is not seen as a significant problem to define them. Addressing the readability and understandability portions of this requirement, OCL is intended to be simple to read and write, having a familiar syntax that can be readily learned by anyone comfortable with programming notation. Because of its lack of formal semantics, OCL is rated B+ in satisfying Requirement 1.

Requirement 2 specified the language must be able to represent information using a combination of graphics and text. OCL itself is a textual language, but because it was designed for UML, OCL contains a tightly coupled graphical level of expressiveness, thus meeting this requirement.

As outlined in Section 2.4.4.1.1, UML offers a variety of options to meet the interaction ability addressed in Requirement 3. Events, signals, and messages represent the most common communication techniques available using UML.

Because the object-oriented paradigm is based on the decomposition of a problem into objects, and because UML and OCL are based on this paradigm, the modularity support of Requirement 4 is easily satisfied.

As stated in Chapter 2, a modeler can view just the UML diagram to understand the overall relationships represented, or can examine low-level details of the model specified in OCL. This allows a specification to be examined at various levels of abstraction and satisfies language Requirement 5.

Requirement 6 states the language must allow for the representation of complex data structures and operations. Through the use of inheritance and aggregation, one or more classes can be connected in such a manner as to create the most simple to the most complex data structures. In regards to representing operations, OCL expressions can be used to define the pre and postconditions of an operation. Because of the expressiveness of the language, operations can be defined ranging from simple to very complex.

The same *Z* argument allowing the representation of alternate designs that satisfied requirement 7 in Section 4.2.1, also applies to UML and OCL. By using UML case tools, designers can manipulate and view alternative representations of a design quickly and easily.

The final requirement states the language must be able to represent the static and dynamic behavior of a system. Regarding this requirement, UML provides diagrams to capture both the static and dynamic aspects of a system. Class diagrams are used to document and express the static structure of a system, while state, sequence, collaboration, and activity diagrams can all be used to express the behavior (dynamics) of the system.

4.2.3 Existing Language Summary

As seen in Table 1, the combination of UML and OCL clearly seem like the best languages for modeling agents. However, after some preliminary testing, a number of problems became apparent. A significant advantage of using UML and OCL to specify agents is its ability to represent information at various levels of abstraction using both graphics and text. The problem was that the graphical level of abstraction was not high enough to allow for the manageable construction of agents. Figure 19 shows a simple reactive agent modeled using UML. The purpose of the agent is to receive messages from other agents and react to them based on the content and performative. When a message is received by the `MessageInterface`, the `checkRules` method of the `ReactiveAgent` is called. The `checkRules` method iterates through all of the rules in the `RuleContainer`, and executes any that are appropriate. Because the execution of some rules may involve sending reply messages or invoking effectors, the `Rule` class needs associations to both `MessageInterface` and `Effector`. The specifics of the operation of the agent are written in OCL and embedded in the operators. The OCL code for all operators can be seen in Appendix B. The diagram depicts how even the most basic agent quickly becomes large and complicated using UML. The answer to the problem required a higher level of graphical descriptions be used.

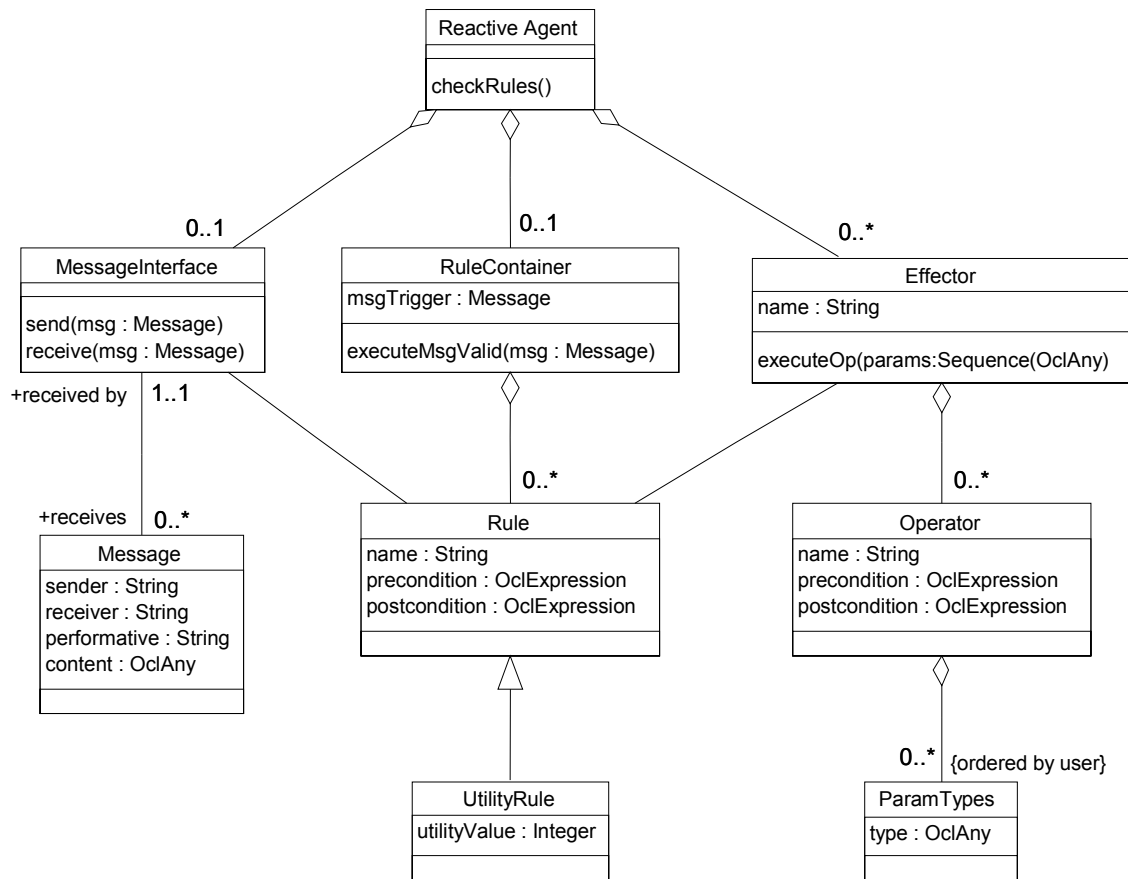


Figure 19 Reactive Architecture

Following Step 2 of the approach outlined in Section 3.2, the next step was to see if an existing language could be modified to meet all requirements. As described in Section 2.3, software architectures separate and represent information at a higher level of abstraction, but don't provide the low-level detail offered by a language such as OCL. The solution was to use an architectural representation for the graphical depiction of internal agent components, and use OCL as the low-level specification language to represent the explicit details of each component. Because UML and OCL met all language requirements, the goal was to find a way to capture the UML diagrams in a more abstract, architecture-based diagram. The approach followed was to transform the class diagram into a software architecture, thus preserving the expressiveness, but doing so at a higher

level of abstraction. The result would be an architectural specification language having the high level graphical abstraction of an ADL and the low-level details of OCL.

4.3 Language Definition

This section outlines how the UML class diagrams are converted into an architectural specification language and defines the syntax and semantics of the language. The language object model is defined in Section 4.4 and the implementation of the language is described in Section 4.5.

As stated earlier, the main elements of software architectures are components and connectors while the main elements of a class diagram are classes and relationships. The overall approach was to find a way to represent classes and relationships as components and connectors. The first step was to define what would be considered a component. Using the definitions of Section 2.3.1, a component represents an independently deployable repository of computation. Therefore, any class or classes with computational ability that can work independently is a component candidate. Using this definition as a guideline, Figure 20 is an example of how the classes of Figure 19 were grouped into components. The grouped classes shown in Figure 20 will be referred to as *component classes* for the remainder of this section. Because some level of computation must take place (per the definition), a component class must contain at least one operator. Once a component identification process was determined, the next step was to transform the class relationships.

Two types of relationships have to be examined, component class relationships and component relationships. *Component class relationships* exist between the classes making up the components and *component relationships* exist between the selected components.

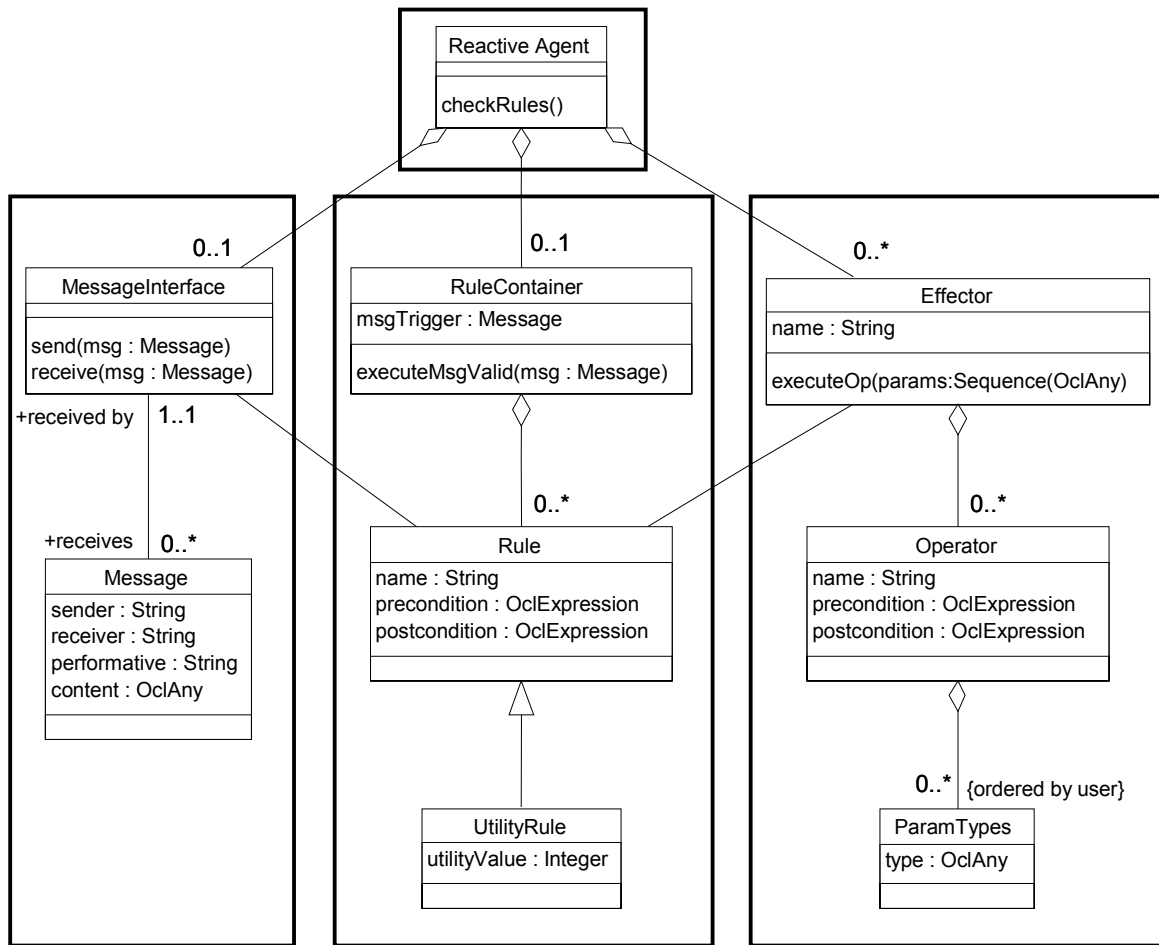


Figure 20 Component Grouping

For example, the relationship between **Message** and **MessageInterface** is a component class relationship while the relationship between the **ReactiveAgent** and the **MessageInterface** is a component relationship. It is important to note the order in which associations are transformed. Component class relationships must always be transformed first. Component class associations are transformed in a bottom up manner until the user is left with one class per set of component classes. Which set of component classes are done first is not important. Once all component class relationships have been transformed, component relationships can then be addressed. The order in which these relationships are transformed is not important.

4.3.1 Component Class Relationships

Numerous types of relationships exist in class diagrams, but the two most encountered are associations and generalizations (also known as inheritance). Section 4.3.1.1 outlines the transformation of associations and Section 4.3.1.2 presents the process to transform inheritance from a class diagram.

4.3.1.1 Associations

The most common associations are normal associations and aggregation. *Aggregation* is often referred to as the “consists of,” “contains,” or “is part of” association and for those reasons is the easiest to eliminate. For example, in Figure 20, the `RuleContainer` “consists of” zero or more `Rule` where `Rule` is the aggregate class. All aggregate classes are transformed by simply moving them into the parent class and representing them as attributes. An example of this is seen in Figure 21. Note that the aggregate `Rule` class cannot be transformed until the inheritance class `UtilityRule` is transformed. This is done in Section 4.3.1.2.

If the cardinality of the aggregation is greater than one, it is represented as a set, unless the aggregation is ordered, in which case it is represented as a sequence. An example of this is the `ops` attribute of `Effector` from Figure 21. Because `Effector` consisted of an aggregate class, which itself consisted of an aggregate class, aggregation was removed in a bottom up fashion. As previously stated, the lowest level of aggregation is transformed first followed by each higher level of aggregation. In Figure 20, `ParamTypes` was first moved into `Operator` and represented as the attribute `params:Sequence(OclAny)`. Because of the ordering imposed on the aggregate class, the information had to be captured as a sequence. `Operator` was then moved into `Effector` and represented as the attribute `ops:Set(Operator)`. Because the cardinality was greater than one and not ordered, the class was captured as a set with type `Operator`. Although the aggregation

within the components is removed, the collection element types must still be stored somewhere so the component may access them.

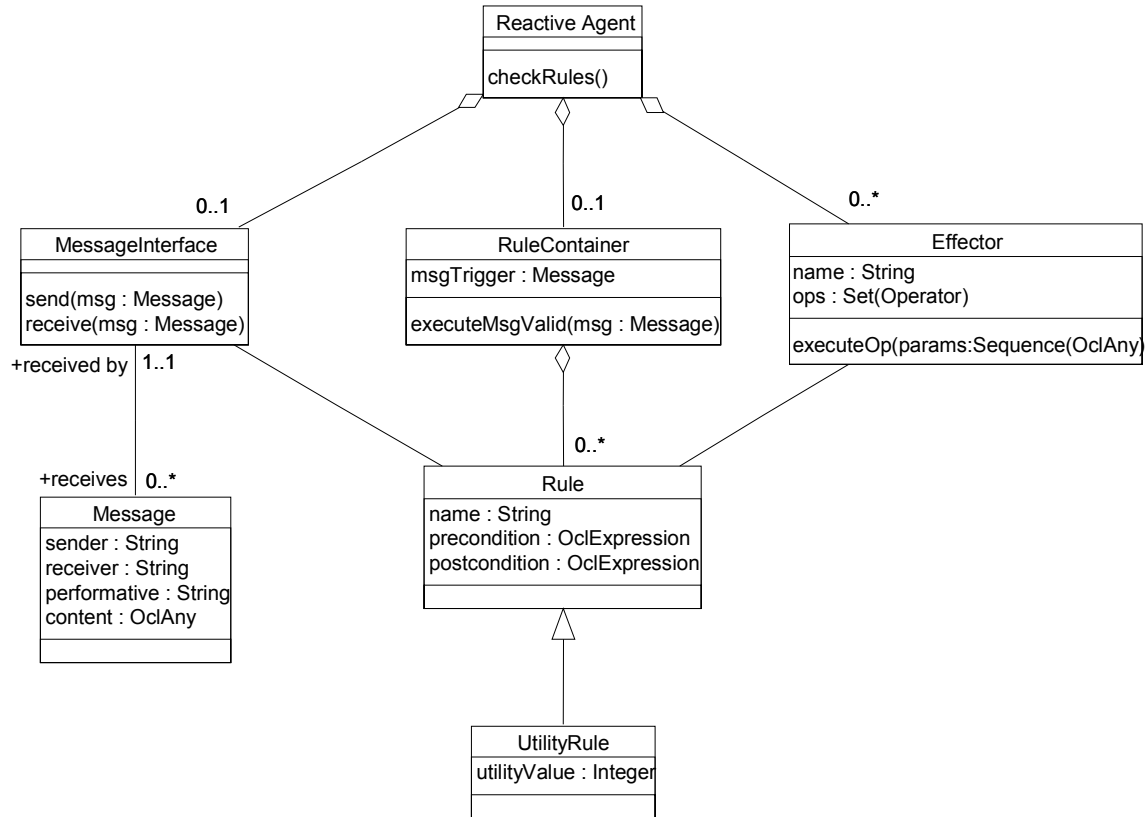


Figure 21 Aggregation Removal

In Figure 21, the collection is a set containing elements of type **Operator**. Collection element types that in and of themselves meet the definition of a component, must be stored as substructures. A *substructure* represents the internal architecture of a component. Just as a system may be composed of one or more components, so may a given component. A substructure is one or more components that are “inside” a component. Referred to as compositional knowledge [KJ98], this “has-a” relationship allows for the organization of information that makes up complex components. A component can directly access all attributes and operators of any components in its substructure. However, if a component needs access to

the substructure of another component, an interface must be defined. For example, if **Operator** were a component and was stored as a substructure of **Effector**, any other component needing access to attributes of **Operator** would need to call an operator in **Effector**, which would then access the information. In order for a user to know if a component has substructures, the asterisk symbol (*) is appended to the name of any component containing substructures. A substructure may be used at any time at the designer's discretion, to embed varying levels of complexity into a component. Any component within a substructure may also contain a substructure (and so on).

All other collection element types not considered to be substructures are stored as data structures. A *data structure* is a representation of the logical relationship among individual elements of data [PRES97]. Each component may have a set of data structures associated with it. All data structures specified are stored in a common library. Data structures are used in the same manner as basic OCL types (integer, real, boolean, etc.), but unlike the standard set of OCL types, which are available to all components, data structures are only available to components having access to them. A component only has access to the data structures associated with it. When the *OclAny* type is used as an attribute type, the attribute value can be of any basic OCL types as well as any data structures the component has access to. The component does not have access to all data structures stored in the library. Data structures are not visible in the component diagram but can be accessed by the user.

Two types of normal associations exist that must be removed: normal associations between component classes and normal associations within component classes. Normal associations between component classes will eventually be replaced by connectors. This issue will be addressed in Section 4.3.2. All other normal associations in a class diagram are used simply to give one class access to the attributes of another. Within component classes, normal associations are also treated as data structures since if the associative class had any operators, it

would either be a component or be part of a component. The **Message** class of Figure 21 is an example of an associative class that is converted to a data structure. The sole purpose of the **Message** class is to allow the component access to the format of **Message**. The class does not do anything; it simply provides a representation of the logical relationship of the data.

4.3.1.2 Inheritance

The last relationship within component classes to be removed is inheritance. If a subclass Y inherits all of the attributes and operations associated with its superclass X, this means that all the data structures and algorithms designed and implemented for X are immediately available to Y [PRES97]. Therefore, inheritance is removed by copying all inherited class attributes and operators into the superclass. Subclass attributes and operators override any attributes or operators in the superclass with the same name as those in the subclass. Two types of inheritance exist that must be removed: *component inheritance* and *data structure inheritance*. If the superclass or subclass meets the component definition, then the combination of classes results in a component. If the superclass and subclass are data structures, then the combination of classes results in a data structure. For example, in Figure 21, the subclass **UtilityRule** becomes a data structure composed of the attributes of both the **UtilityRule** and **Rule** classes. The resulting class is a data structure because both the superclass and subclass are data structures. If the superclass met the component definition, then the resulting class would have been a component containing all of the attributes and operations of the two classes. If both the superclass and subclass need to be accessed individually, the following additional steps must be taken. When there is component inheritance, two components must be created. One component represents the combination of both the superclass and the subclass while the other component contains just the superclass information. The superclass component is then placed in the substructure of the new inheritance

component. Both components now exist individually and can be accessed by other components. If there is data structure inheritance, then a data structure must be defined for the combination of the superclass and subclass and a data structure must be defined for just the superclass. For example, if both `UtilityRule` and `Rule` were accessed individually (i.e., one component uses `Rule` and another component uses `UtilityRule`), separate data structures would be defined. One data structure would be called `UtilityRule` and would contain all of the attributes of both `Rule` and `UtilityRule`. The other data structure would be called `Rule` and would contain all the attributes associated with the rule class. These data structures can then be associated with any components needing access to them. Figure 22 depicts Figure 21 once all component class relationships are removed.

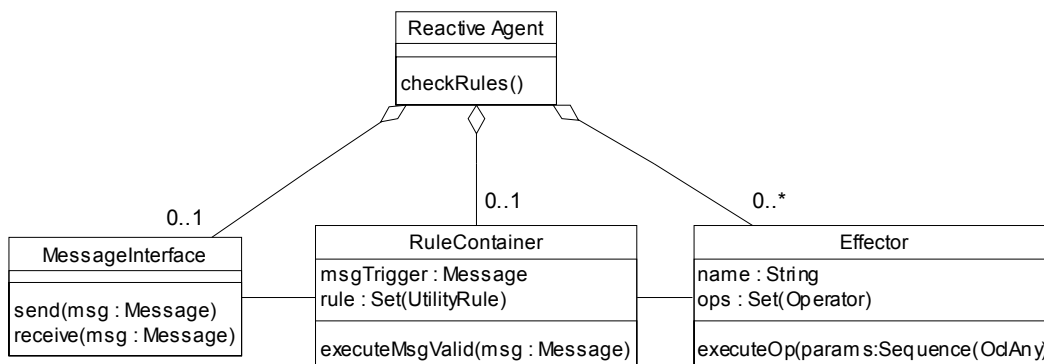


Figure 22 Inheritance and Aggregation Removal

Each class shown in Figure 22 now represents a component. As seen from the figure and the previous sub-sections, the main elements a component consists of are attributes, operators, data structures, and substructures. Because classes normally have state diagrams associated with them to represent the dynamics of the system, each component has a state model to represent the behavior of the component.

4.3.2 Dynamic Representation

State diagrams are used to represent the dynamic aspects of a component. A *state diagram* describes the states a component can be in during its life cycle along with the behavior of the component while in those states. The diagram also describes the events that cause a state change. An *event* is something that happens and that will cause some action. How events are received by a component is addressed in Section 4.3.3.2. An example of a state diagram for the `MessageInterface` component of Figure 22 is shown in Figure 23.

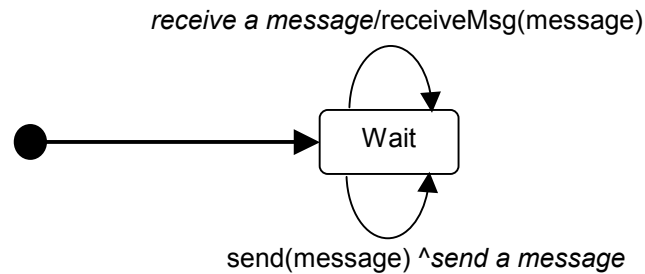


Figure 23 `MessageInterface` StateDiagram

When the component is instantiated, it automatically goes into the `Wait` state. Whenever a message event is received the `receiveMsg` operator is invoked. Any operator called within the state diagram must be defined in the component diagram. When the `receiveMsg` operator has completed execution, the component will transition back to the `Wait` state. Whenever the `send` operator is executed, a send event is generated. There is no OCL code associated with the `send` operator in the component diagram since everything the operator does is depicted in the state diagram. Any operator declared within a component must either have an OCL definition associated with it (as seen in Appendix B) or must be associated with the generation or receipt of an event. If an operator does neither of these things then it does nothing and should be removed from the component. Because both the receive event and send event are implementation

dependent, they are defined on a case by case basis. For the purposes of modeling the events, they are represented in the diagram as **receive a message** and **send a message** respectively.

4.3.3 Component Relationships (Connectors)

Once components are defined, the only relationships left are between the components themselves. Because the language is based on software architecture principles, any relationship between two components is replaced with a connector. To represent all component interaction, two types of connectors are needed: inner and outer agent connectors.

4.3.3.1 Inner Agent Connectors

Inner agent connectors are used to connect two components within an agent. The purpose of the connector is to give components access to the attributes and operations of other components. *Access* in this context, is the ability to read attributes, add data to collections, and invoke operators. Access does not give other components permission to delete or modify the structure of a component. For example, one component cannot delete the attributes or operators of another component. A component also cannot change the definition of any attributes or operators (i.e. cannot change attribute types or operator specification). All remaining class relationships of Figure 22 were replaced with inner agent connectors as shown in Figure 24.

The inner agent connector is represented as a one-way or two-way thin arrow. A one-way arrow indicates that the component at the originating end of the arrow can access the component where the arrow points, but the other component has no access to the originating component. An example of this is seen between the `RuleContainer` and the `Effector`. A `Rule` within the `RuleContainer` may be triggered that has the agent make changes to its environment using its effectors. Because of this, the `RuleContainer` component must have access to the `Effectors`.

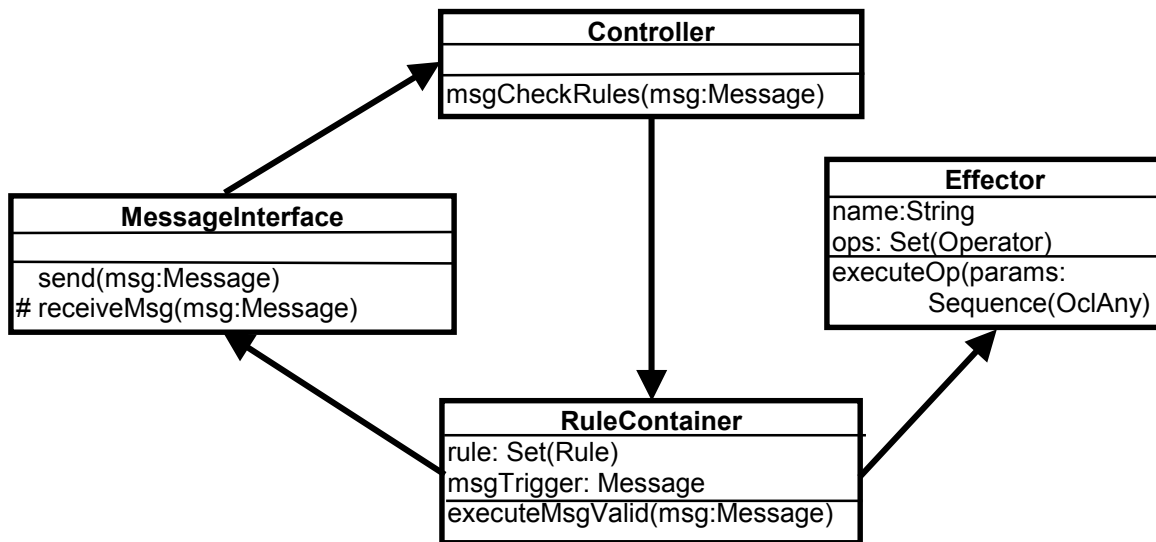


Figure 24 Inner-agent Connectors

There is no reason for an **Effector** to have access to the attributes and operators of the **RuleContainer**, therefore a one way arrow is used. A double arrow indicates that each component has access to the others attributes and operators. As mentioned in Section 4.3.1, a component will only have access to another components substructure if an interface operator is defined.

UML Class diagrams have two levels of visibility to control access to attributes and operators; public and private. *Visibility* describes whether the attribute or operator is visible and can be accessed from other classes. A *public* attribute or operator can be viewed and used outside of the class while a *private* attribute or operator cannot be accessed by any other classes. Components also require this level of protection. Therefore, each attribute and operator of every component can be declared as visible or invisible. Invisibility makes an attribute or operator inaccessible to all other components. Invisible attributes and operators are still depicted in the component diagram but are preceded by the ‘#’ symbol. An example of when this would be necessary is seen in the `receiveMsg` operator of the **MessageInterface** component. The

operator is only triggered by an external message event (discussed in detail in Section 4.3.2.2). Therefore, no other component should ever be able to invoke this operator. To ensure this does not happen, the operator is made invisible. Visible attributes and operators are depicted normally.

4.3.3.2 Outer Agent Connectors

Outer agent connectors allow agents to interact with other agents as well as their environment. Like the inner agent connector, outer agent connectors are also represented as one-way or two-way arrows. In regards to appearance, two differences exist; outer agent connectors are represented as much thicker, dashed arrows and only one end of the outer agent connector is connected to a component as seen in Figure 25.

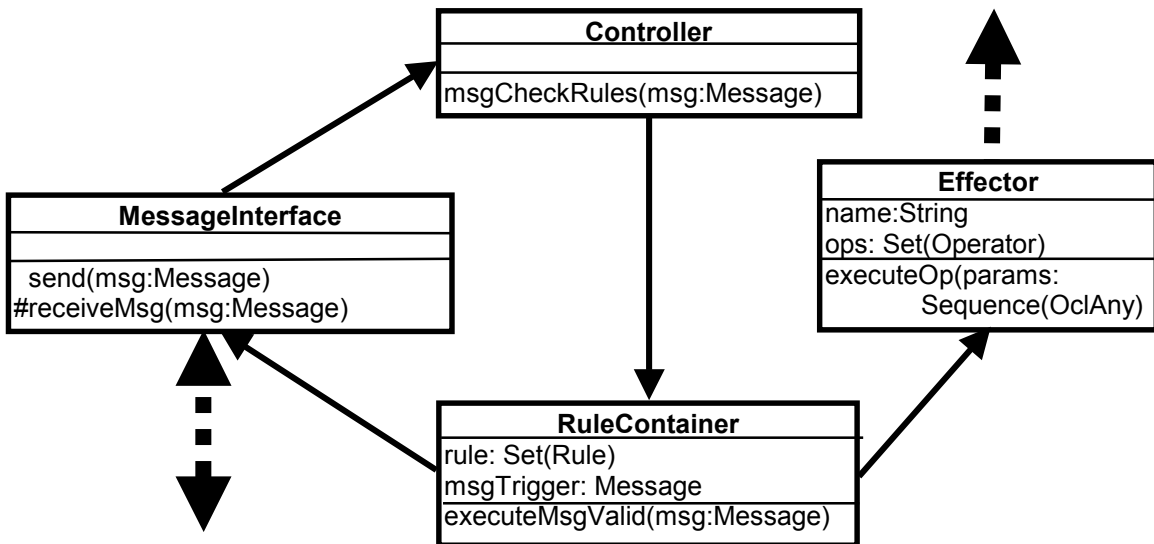


Figure 25 Outer-agent Connectors

The other end of the connector represents the external entity the component is interfacing with. Outer agent connectors are used for sending and receiving external events. Component interaction will only begin taking place in an agent after one of two things occurs: an external event causes an operator to be executed (discussed in Section 4.3.2) thus triggering a sequence of events or the initialization operator is invoked. The initialization operator is an optional method

that can be placed anywhere within a component diagram. Only one operator is allowed per agent and it is executed immediately upon agent instantiation. The operator may be used to start a continuous chain of events or may be used to simply send a message stating the agent is active.

If a one-way outer agent connector points away from a component, the component can only generate events, and cannot be affected by events generated by the entity it is connected to. An example of when this would be useful is an effector component. An effector is going to impose changes on the environment by generating events, it is never going to receive. The opposite of this would be a one way outer agent connector pointing towards a component. This type of connector can only receive events from external entities. An example of this would be a basic sensor component. The sensor component will receive information from the sensing device whenever changes occur. The final connector type is the two-way outer agent connector, which can send and receive events. Examples of this would be a message interface or a sensor that can be periodically queried for changes in the environment. Figure 25 adds outer agent connectors to Figure 23.

4.3.4 Language Definition Summary

This section defined the architectural specification language used in this thesis. It has also shown that the representation captures all information in a UML class diagram. The process outlined simply collapses a class diagram into a higher level of abstraction while maintaining the attributes of a class diagram. Although the language is based on object-oriented principles, this does not imply that an object-oriented design methodology has to be used. The language defined can be used in conjunction with any established design methodology for the creation of software systems.

4.4 Object Model Definition

Once the language has been completely defined, it is then possible to create a generic template defining the syntax and semantics of the language. This is accomplished by creating an object model for the language. Following the approach of Section 3.4, the first step in defining an object model for the language was to determine the basic classes required. Because the overall solution is based on using a software architecture approach, Figure 15 of Section 3.3 depicts an initial decomposition. A flaw in the figure is that it does not take into account the two different kinds of component connectors defined in Section 4.3.2. The diagram specifies that every connector connects two components, however, outer agent connectors only connect one component. A corrected version of the diagram is shown in Figure 26.

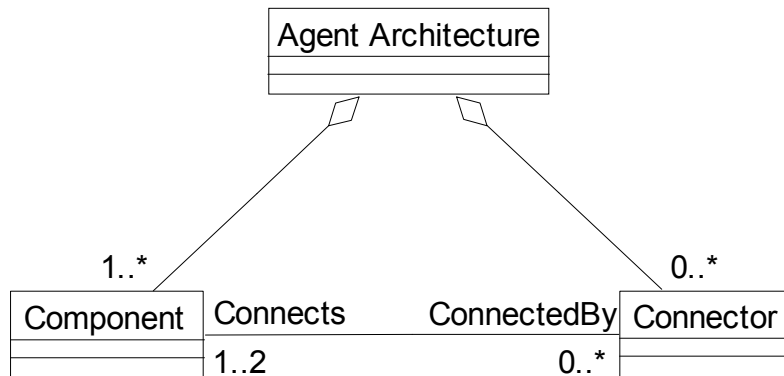


Figure 26 Basic Object Model

The remainder of this section builds on this figure by further refining and defining each class based on the language description outlined in Section 4.3.

4.4.1 Component Class

As stated in Section 4.3.3.1, each component consists of attributes, operators, data structures, and substructures. A refinement of Figure 26 capturing this information is shown in Figure 27.

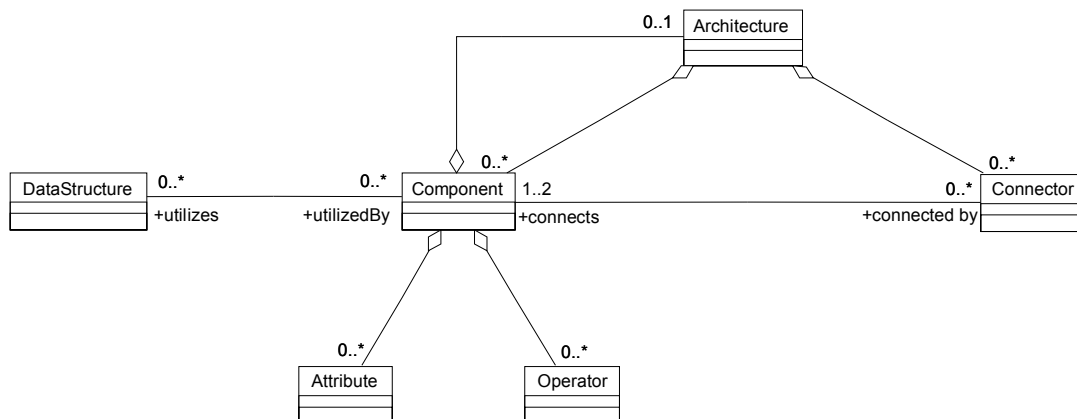


Figure 27 Object Model Refinement

The aggregation from **Component** to **Architecture** states a component may be composed of zero or one architecture. This captures the fact that every component may contain at most one substructure and that substructure may contain zero or more connected components (which may each consist of one substructure, and so on). The association from **Component** to **DataStructure** captures the fact that a component may have data structures associated with it.

Once the basic classes were identified, attributes were determined for each class. The first attribute identified in the component class was the **name** attribute. Like a class in a class diagram, every component must have a unique name to distinguish it from other components. To adequately capture the aggregation between **Component** and **Architecture**, three additional attributes were added to the component class; **hasSub**, **sub**, and **parent**. The **hasSub** attribute is of type boolean, and specifies if the component contains any substructures. The **sub** attribute is also boolean, and defines if the component is part of a substructure itself. If the **sub** attribute is true, the **parent** attribute is used to hold the name of the parent component. As stated earlier, in order to capture the dynamics of a component, each component contains a state diagram. This information is captured in the **stateDiag** attribute. The **stateDiag** type **ATstatetable** represents

the object model for state diagrams used in the component. A more thorough examination of this data type is presented in [WOOD00]. The `constraints` attribute represents any invariant constraints imposed on the component. One or more conjuncted predicates may be used to define all component constraints. For example, a designer may specify a `RuleContainer` component that has `max` and `min` attributes used to represent the maximum and minimum number of rules needed for the agent to function properly. To specify that the minimum number of rules needed is 5 and the maximum number of rules allowed is 50, the designer could specify the constraint `min>=5 and max <=50`. The object model with all component attributes added is shown in Figure 28.

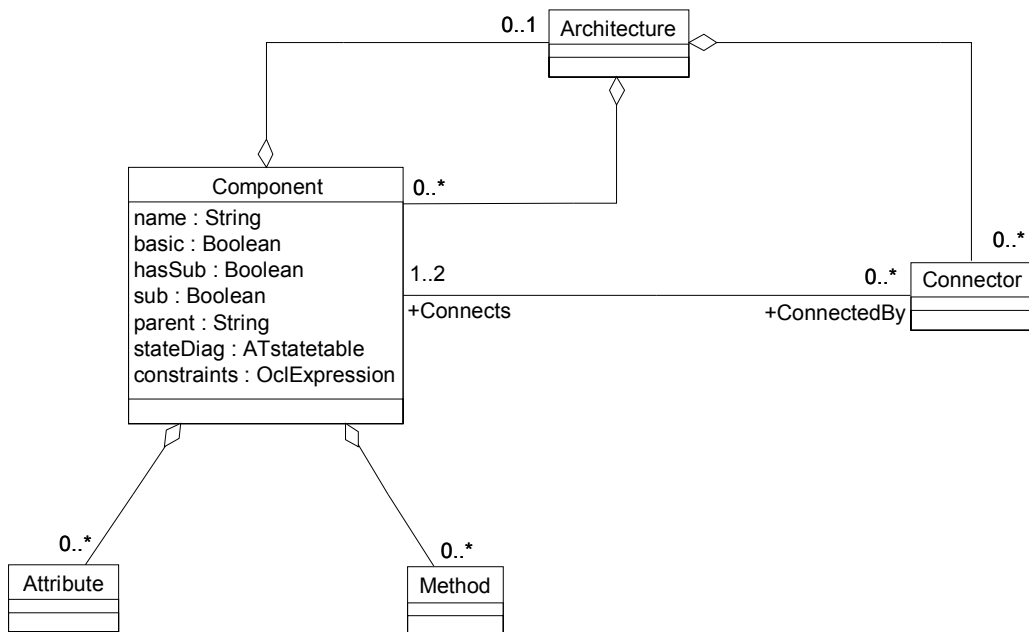


Figure 28 Component Attributes

4.4.1.1 Attribute Class

In a class diagram, *attributes* capture information needed to describe and identify an instance of a class. Attributes also have an associated *type* that tells what kind of attribute it is. In general, agent attributes are used to hold information specifying the agent's state. Given these

descriptions, two basic attributes needing to be represented in the `Attribute` class were `name` and `type`. In order for an attribute to be represented as any of the predefined OCL types as well as any new user defined types, the `type` attribute was made of type `OclAny`.

To limit access to an attribute, the `visible` attribute was added. If `visible` is true, then other components have access to the given attribute, but if `visible` is false, no access is given.

To capture the fact that an attribute may represent a collection of objects, the boolean attributes `set`, `sequence`, and `bag` were added. Only one of these attributes may be true at any time. If one of the three attributes is set to true, then `type` represents the type of the elements in the collection. If all three are false, then `type` represents the type of `name`.

Because the component diagrams act as a template from which executable code may be generated, it must be possible to instantiate the component attributes with specific values. Since an attribute can be user instantiated, system instantiated, or both user and system instantiated, the boolean attributes `userDefined` and `runTimeDefined` were added. If both `userDefined` and `runTimeDefined` are set to true, then the attribute could be instantiated by a user and by the system. An example of this could be a data attribute in a knowledge base component. Some information in the knowledge base will need to be predefined by the designer of the system, while other information will be added by the system once it is running.

In order to store an instantiated attribute value, a generic `Value` class was defined. This generic class inherits from classes `SingleValue`, `SetValue`, `SequenceValue`, and `BagValue` depending on the values of `set`, `sequence`, and `bag`. `SingleValue` has a `value` attribute of type `OclAny`, `SetValue` has a `value` attribute of type `Set(OclAny)`, `SequenceValue` has a `value` attribute of type `Sequence(OclAny)`, and `BagValue` has a `value` attribute of type `Bag(OclAny)`. An example of this would be, if the values `set`, `sequence`, and `bag` were all

false, then the `value` attribute of `SingleValue` would contain a specific instance of the attribute. For example, if `name` were equal to 'Car' then `value` could equal 'Mustang'. The object model showing all attributes of the `Attribute` class is shown in Figure 29.

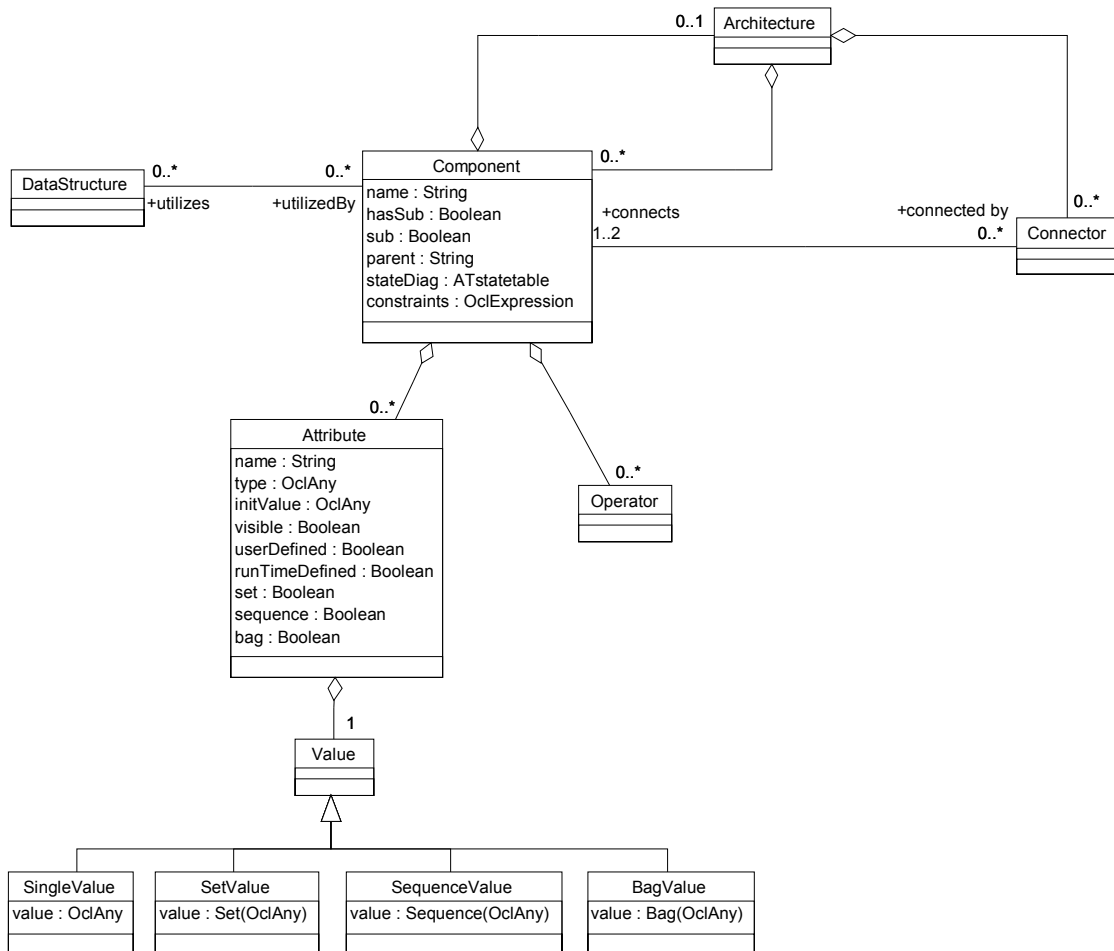


Figure 29 Attribute Class Attributes

4.4.1.2 Operator Class

Operators provide a representation of the behaviors of objects, normally done through the manipulation of attributes. Resembling a function in most programming languages, operators are described with a name, return-type, and zero or more parameters. To capture this information, `name` and `returnType` attributes and a `Parameter` class was added. Operators may

consist of zero or more parameters each having a unique **name** and **type**. Because parameters can be of differing types, the order in which they are evaluated is important. Representing the **Parameter** class using an ordered aggregation was done to take this into account. The attributes mentioned thus far describe everything needed to use the operator, but to capture what the operator does, precondition and postcondition information must be specified. This information is stored in the **pre** and **post** attributes. The **pre** attribute stores the information that must be true in order for the operator to be executed while the **post** attribute stores what must be true after the execution of the operator is complete. Because **pre** and **post** are specified using OCL, both are of type **OclExpression**.

As with the **Attribute** class, a **visible** attribute was added to limit accessibility to each operator. An updated object model is shown in Figure 30.

4.4.1.3 Data Structure Class

Like **Component**, **Attribute**, and **Operator**, **DataStructure** required a unique name for identification purposes. To capture both simple and complex data structures, an aggregate class **DataField** was added. **DataField** consists of **name** and **type** attributes to allow data structure fields to be of varying types. An example of using this would be if the user needed to define an automobile data structure. The data structure would need several fields that could hold information regarding a particular automobile such as make, model, price, year, etc. To model this, a data structure would be defined with **name** set to 'Automobile'. Each piece of information relating to an automobile would be defined as a data field. One data field would have **name** set to 'make' and **type** set to 'String'. Another would have **name** set to 'year' and **type** set to 'Integer'. Once completed, this data structure could be associated with a component and referenced just as any other basic OCL type.

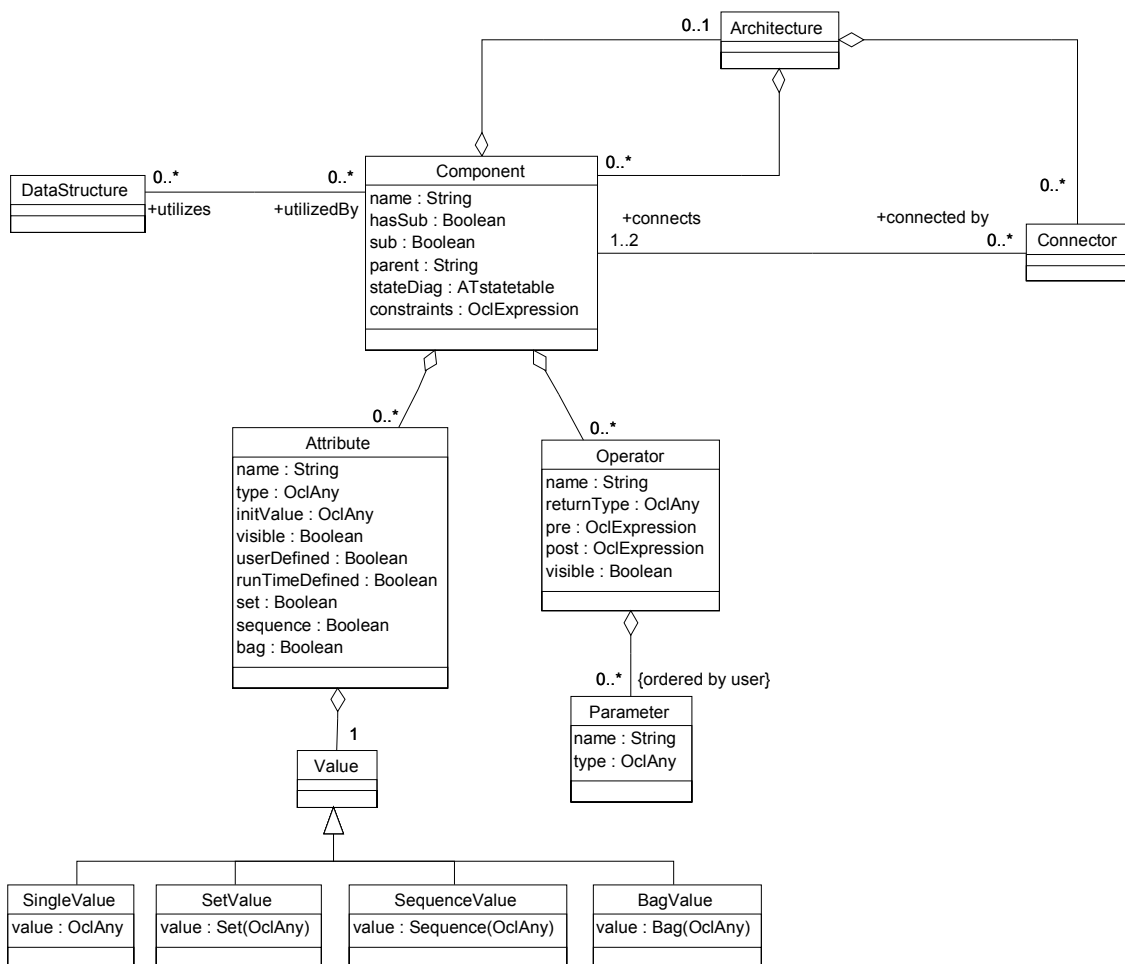


Figure 30 Operator Class Attributes

For instance, if the user needed a variable to store multiple automobiles, they could define an `Automobiles` variable of type `Sequence(Automobile)`.

Because `OclAny` is the supertype of all predefined OCL types as well as any user defined types, in order for a `DataSet` to be used in an OCL diagram, it must be stored as an OCL type. This would allow any attribute having type `OclAny` to have access to the data structure. The object model including data structure information is shown in Figure 31.

4.4.2 Connector Class

As depicted in Figure 31, an architecture consists of zero or more connectors, each of which is connected to one or two components. To more adequately represent the fact that there are two types of connectors, two additional classes were created: InnerConnector and OuterConnector.

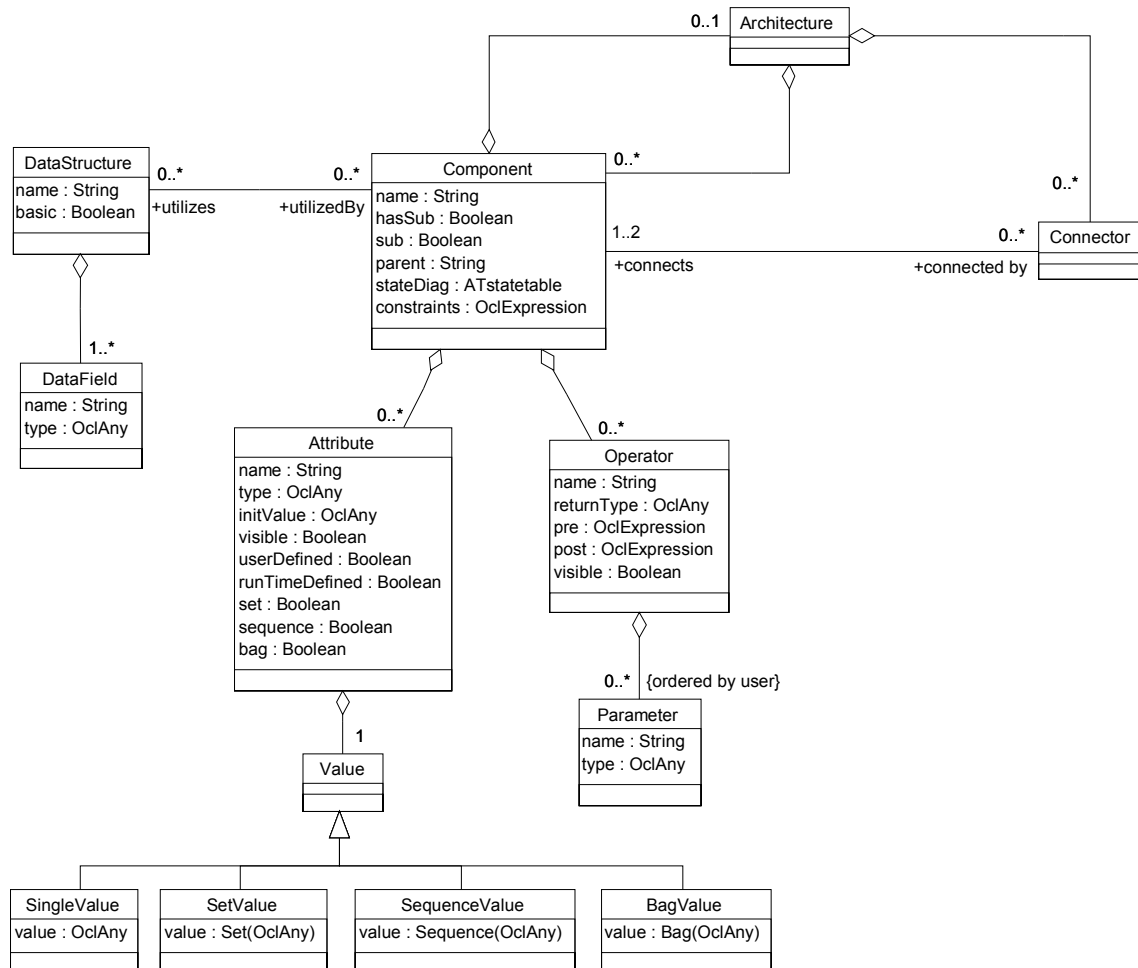


Figure 31 Data Structure

Both classes inherit from the Connector class. Each class has a name and type attribute, where name is used to identify a connector and type is hard coded. For InnerConnector, type is set to 'innerAgent' while for OuterConnector, type is set to

'outerAgent'. The reason behind separating these into two classes was to represent that an InnerConnector must connect two components and an OuterConnector connects one component. This fact was enforced using associations between each class and Component. Figure 32 shows the complete architectural object model.

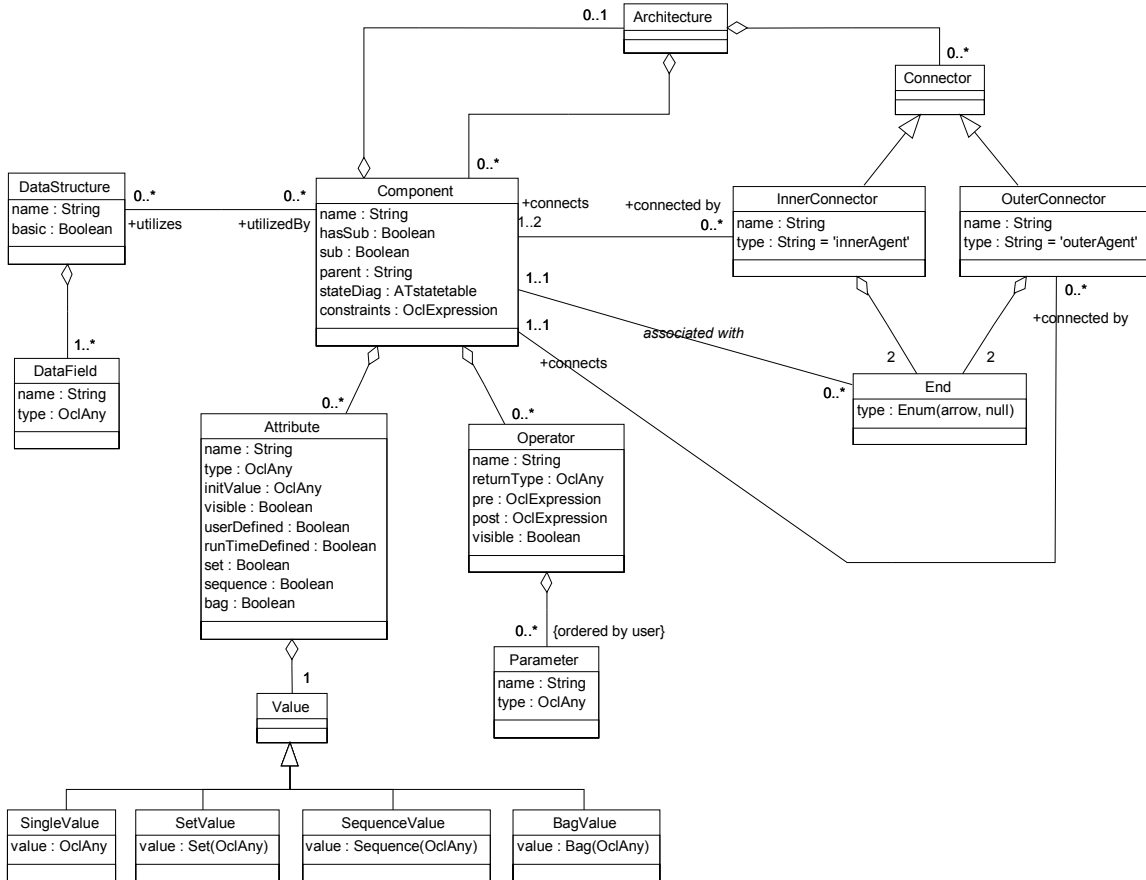


Figure 32 Complete Object Model

4.5 Language Implementation

The language described in the previous sections was implemented in the multi-agent design environment *agentTool*. The purpose of this section is to give an overview of how the

language described in the previous sections was implemented in this environment. A complete description of the implementation is given in Appendix C.

Based on the Multiagent Systems Engineering (MaSE) [MASE99] methodology, the purpose of *agentTool* is to allow multiagent systems to be quickly and easily designed and implemented. Within the system, once an agent has been defined, internal agent components may be added, deleted, modified, and connected to other components or to the environment. Both the static and dynamic representation of components is currently implemented. A static representation of the internal components of a reactive agent is shown in Figure 33.

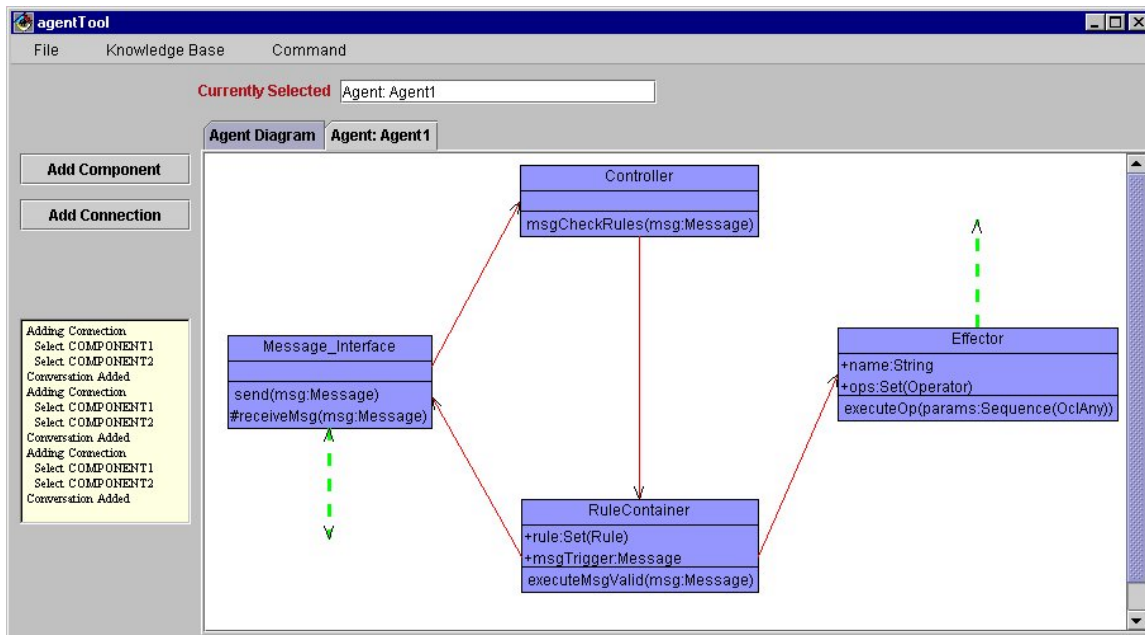


Figure 33 Component & Connectors

Once a component is specified, attributes and methods may be added, deleted and modified for that component. Figure 34 shows the interface used to specify a component attribute, while Figure 35 shows the interface used to specify a component method. Both interfaces coincide with the language object model specified in Figure 32.

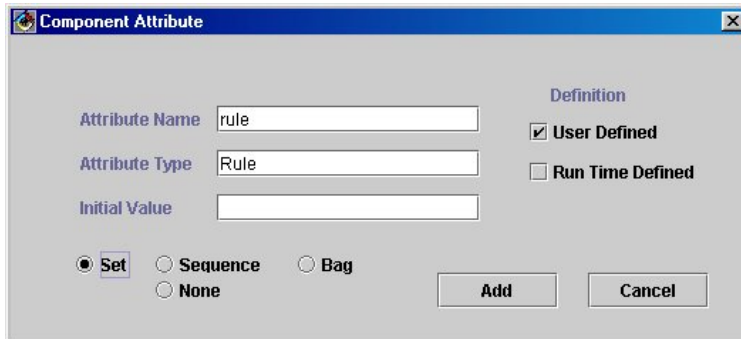


Figure 34 Attribute Interface

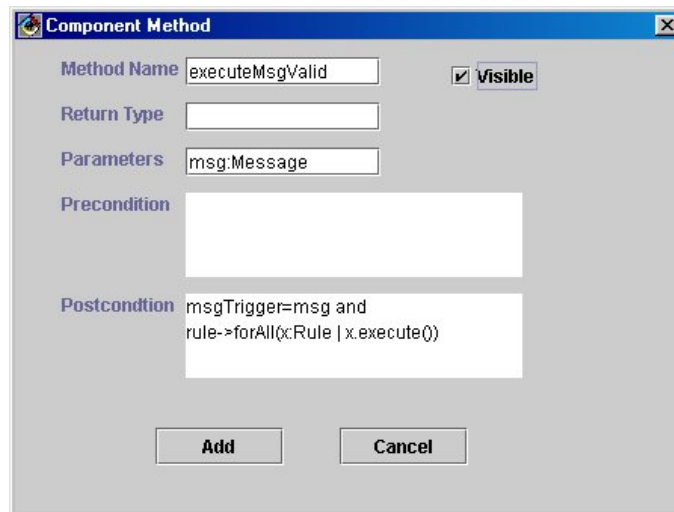


Figure 35 Method Interface

As shown in Figure 34, the user can choose whether the attribute is user defined, run time defined, or both. These selections will be used for the code generation portion of *agentTool* (not yet implemented). As seen in Figure 33, the ‘+’ symbol indicates an attribute is user defined, while the ‘-’ symbol indicates an attribute is run time defined. Although not shown in the figure, the ‘±’ indicates an attribute that is both run time and user defined. Figure 35 shows how the OCL code is specified in the postcondition for a given method. Although not shown in the

component diagram, this information is stored in the language object model to be used for code generation purposes.

To define the dynamic representation of the software, every component defined has an associated state diagram. The representation of the information within these diagrams is the same as that used in UML. The dynamic representation for the `Message_Interface` component is shown in Figure 36.

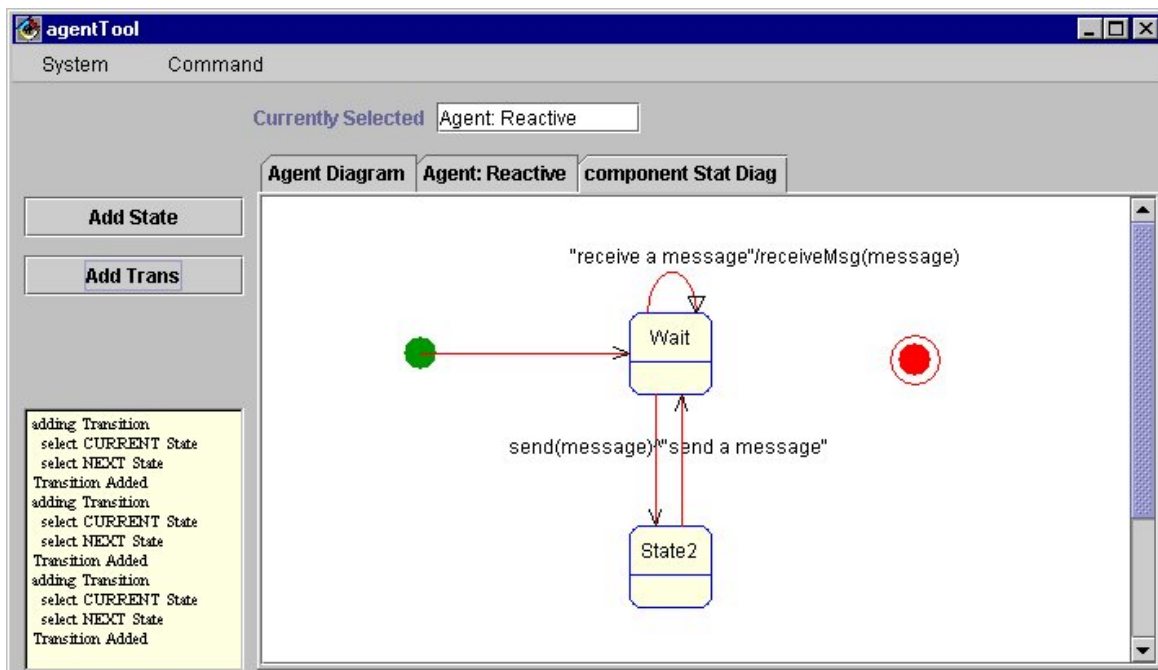


Figure 36 Completed Dynamic Model

Using methods described in [LACEY00], it is possible to automatically verify that a component dynamic model is free of dead locks and infinite loops.

4.6 Chapter Summary

The first section of this chapter defined the requirements that must be met by a language in order to be used to specify agents in a multi-agent environment. These requirements were

based on analysis of the problem as well as a review of literature of existing languages used for similar purposes. Section 4.2 compared two existing languages against these requirements to find out which was better suited. Because neither language adequately met all language requirements, an existing language was modified to satisfy the problem. Section 4.3 defines this modification and completely outlines the language. Section 4.4 uses the description from Section 4.3 to completely define an object model for the problem. Section 4.5 is an overview of the implementation of the object model in the multi-agent definition environment *agentTool*.

V. Architectural Styles

This chapter tests the validity of the language defined in Chapter 4 by using the language to define some of the most common architectural styles used in artificial intelligence. Sections 5.1 through 5.4 describe the architectural styles selected and the process followed in selecting them. The section closes by describing the generic set of components obtained from the selected architectural styles.

A number of existing agent architectures were reviewed, decomposed using object-oriented techniques and then categorized based on their components, connectors, and overall structural pattern. Four architectural styles were found using this approach: reactive, knowledge-based, planning, and Belief Desire Intention (BDI). The remainder of this section follows a four-step process for each architectural style. First, the requirements of each style are described. Second, a generic component model is constructed defining the basic components and outlining the overall structure. Third, based on the information presented in steps one and two, an object model is created. Fourth, the object model is then transformed into a component diagram using the techniques defined in Section 4.3.

5.1 Reactive Agent Architectural Style

As stated in Section 2.3.3, reactive agents come in a number of different forms based on the attributes that the agent may possess. However, all agent architectures that are reactive in nature share a common set of characteristics.

Because of the stimulus-response nature of reactive agents, all architectures in this category are based on a set of rules. The rules are used to interpret the stimulus and generate a response if appropriate. Although various styles are used to represent rules, the same IF-THEN

structure is predominate. To receive signals and generate responses, each architecture also contained an interface to the environment. These interfaces included a message interface, resource interface, sensors, and effectors. *Message interfaces* are used to send and receive messages between agents. *Effectors* are used to send changes to the environment and *sensors* are used to receive changes from the environment. *Resource interfaces* are used to interact with external data sources ranging from HTML files to databases. Architectures in this category also contain a control mechanism to interpret inputs from the interfaces and select the appropriate rule based on the input. Therefore, using the notation defined in Chapter 4, the architectural style can be represented as shown in Figure 37.

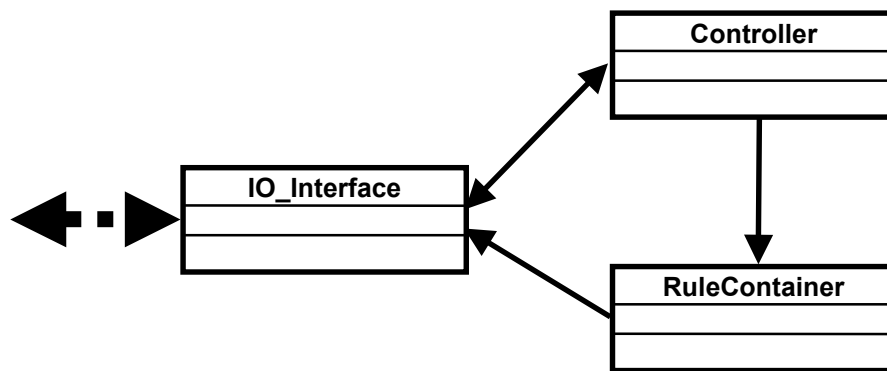


Figure 37 Reactive Architectural Style

In the figure, all interfaces (message interface, sensors, effectors, and resource interfaces) have been combined and represented as an **IO_Interface** component. This captures the fact that only one of these interfaces need be present. This also shows that no matter which interface is present, the same inner agent connectors connect it to the other components. Although not all interfaces have a two-way connector to the environment, the main point of the figure is to stress the structure of the architecture. All interfaces will interact with the environment and the **Controller** component. The **Controller** may take information received from the **IO_Interface**,

and query the rules of the RuleContainer to determine what action needs to be taken. The RuleContainer contains all rules for the agent as well as the operations needed to manipulate the rules. Because a given rule may in turn cause a change to the environment, or cause a message to be sent, there is a one-way inner connector from RuleContainer to IO_Interface. Using Figure 37 along with the initial description, it was possible to determine the basic classes as well as generic attributes and methods. The object model depicting this is shown in Figure 38.

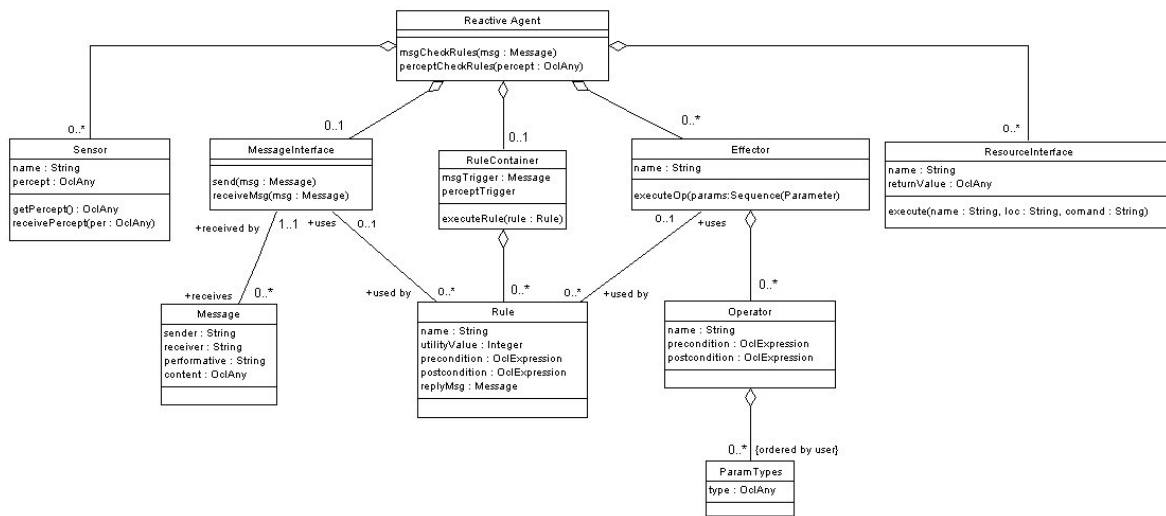


Figure 38 Reactive Architectural Object Model

Each Rule has a name attribute to identify it from any other rule currently defined. Since rules are generally of the IF-THEN format, precondition and postcondition attributes were defined to capture the rule structure. An example of how a rule may be used is shown below.

```

name: "FoodGratuityRule:"
precondition: msgTrigger.performative= "CalculateFoodTip"
postcondition: RuleContainer.ReactiveAgent.replyMsg.content=msgTrigger.content*.15
                and RuleContainer.ReactiveAgent.replyMsg.receiver=msgTrigger.sender
                and RuleContainer.ReactiveAgent.replyMsg.sender= "RestaurantAgent"
                and RuleContainer.ReactiveAgent.replyMsg.performative= "FoodTip"
  
```

An agent could send a message to the “RestaurantAgent” containing the amount of the bill as the content and the performative “CalculateFoodTip”. The agent would then iterate through all of its rules and see that the precondition of the “FoodGratuityRule” is true. Upon finding a valid rule, the postcondition would then be evaluated. As stated in Appendix B, each conjuncted OCL postcondition expression is evaluated in a top to bottom, left to right fashion. Therefore, in the above example, `replyMsg.content=msgTrigger.content*.15` is made true before `replyMsg.receiver=msgTrigger.sender`. How these expressions are made true is dependent on the language that will be used to implement the specification. In the above example, the easiest way to make `replyMsg.content=msgTrigger.content*.15` true is to take `replyMsg.content` and set it equal to the value of `msgTrigger.content*.15`. The specification is not concerned with how the value is made true, only that it is made true.

Because two or more rules may be applicable to a given situation, a `utilityValue` attribute was also defined. This value can be used to determine the overall applicability of a given rule. The `RuleContainer` class acts as a central repository from which all rules may be accessed. The `msgTrigger` and `perceptTrigger` attributes contain the most recent message or percept received by the agent. The `executeRule` operator is used to evaluate rules (i.e. if the precondition is true, then the postcondition is made true).

The control structure of the agent is primarily seen in the `ReactiveAgent` class. When a message is received, the `receiveMsg` operator in `MessageInterface` calls `msgCheckRules`, and if a percept is received, the `receivePercept` operator in `Sensor` calls `perceptCheckRules`. These two operators then use the `executeRule` operator to iterate through all the rules searching for valid preconditions. These operators can be specified in a number of ways depending on the goal of the agent. They can be used to find and execute all valid rules, they can be used to find

and execute the first valid rule, or they can be used to find and execute the rule with the highest or lowest utility value. The operator definition is dependent primarily on the desired result.

The `MessageInterface` class simply contains the `send` and `receiveMsg` operators. These generic operators represent the sending and receiving of messages to other agents. The actual implementation of these operators is based on the type of agent communication protocol selected by the designer. When implemented, these generic operators will be mapped to the particular protocol selected. The `receiveMsg` operator reacts to external message events (as described in Section 4.3) from the environment. As stated previously, the `receiveMsg` operator invokes the `msgCheckRules` operator of the `ReactiveAgent` class whenever a message is received. Just as `receiveMsg` reacts to external events, `send` is used to create external message events. When the `send` operator is invoked, an external message event is created that may be received by other agents with a `receiveMsg` operator. The associative `Message` class contains the general template used to pass messages between agents. This template can be modified at the designer's discretion.

The two attributes comprising the `Sensor` class are `name` and `percept`. As stated in Section 2.3.7.2, sensors can come in two forms: persistent and functional. A persistent sensor is one that is initiated by a change of environmental state. The `receivePercept` operator was defined to model this characteristic. The operator is invoked whenever an external environmental event occurs that the `Sensor` is able to detect. Once invoked, the operator can call the `perceptCheckRules` operator of the `ReactiveAgent` class to determine if any rules have become valid based on the change. The functional sensor was modeled using the `getPercept` operator. Invoking this operator queries the sensor for current information, which is returned and stored in the `percept` attribute.

The **Effector** class contains a **name** attribute to specify the category of effector that the agent will access. For instance, an agent may have five effectors that control a robot arm and another five effectors with the exact same names controlling a robot leg (i.e. move up, move down, etc.). Each set of effectors could be grouped under the type **Arm** and **Leg** respectively without having to rename all operators. The user could then call **Arm.up** or **Leg.up** as appropriate. Each effector class consists of one or more operations that it can perform. In the above example, the **Arm** effector contained five operators to control arm movement. In the majority of the architectures examined, effectors were modeled as agent operators. Keeping with this naming convention, the **Operator** class contains the actual information needed to effect the change to the environment. As described in Section 4.4.1.2, the **precondition** and **postcondition** attributes contain the information that must be true for the operator to be evaluated along with the information that will be true upon completion. These operators may be instantiated by calling the **executeOp** operator of the **Effector** class. This operator takes the name of the operator to be executed as well as a possible list of parameters and evaluates the **precondition** attribute of that operator. If **precondition** evaluates to true, then the **postcondition** attribute is then made true.

The **ResourceInterface** class contains a **name** attribute to delineate between different resources that may be accessed. The main functionality seen in the class is in the **execute** operator. Any type of external data source residing on a computer will always be accessed through some type of application. This application could be an operating system, database, or web browser just to name a few. To interface with the data source, the agent must interface with the application. In order to effectively interact with the application, three pieces of data are required; the name of the file needed, the location of the file, and the command to be executed. To capture this information, the **execute** operator contains the parameters **name**, **loc**, and

command (representing filename, file location, and command). Any return information or feedback from the application is stored in the `returnValue` attribute.

Using the methods described in Section 4.3 for transforming a class diagram into a component diagram, Figure 39 was created.

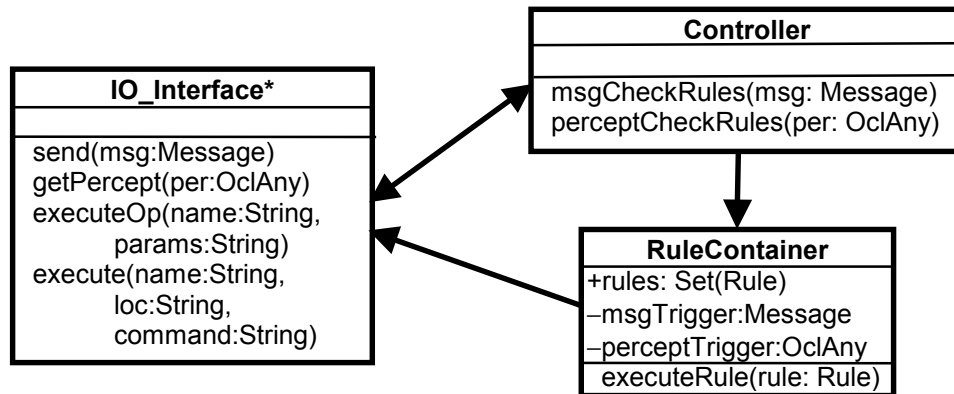


Figure 39 Reactive Architecture Component Model

The components comprising the architecture are a **RuleContainer**, **IO_Interface**, and a **Controller**. The **Controller** component essentially replaces the **ReactiveAgent** class and acts as the control center of the agent. The most interesting component is the **IO_Interface**. The substructure of the **IO_Interface** contains all interfaces the system may have with other agents and the environment. The representation of the substructure is shown in Figure 40. It is important to note that all outer agent connectors are represented in the substructure, hence the reason none appear in Figure 39. Each component of the **IO_Interface** substructure could be represented in the Figure 39 representation. They were moved into the substructure for readability and understandability purposes only. The **Controller** and **RuleContainer** component can still access the components by calling the appropriate interface operator in the **IO_Interface** component. These operators in turn call the operator of the appropriate component.

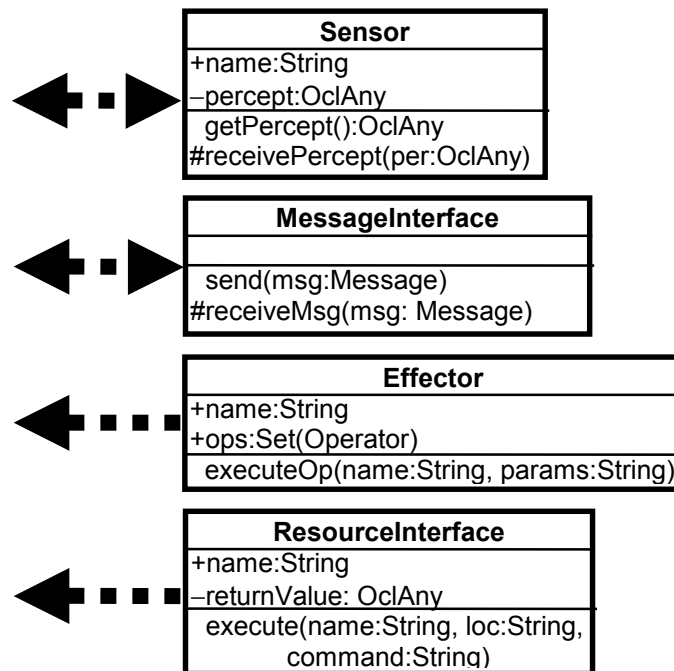


Figure 40 IO_Interface Substructure

Substructure components can access higher-level components by first accessing the parent component. For example, when invoked, the `receiveMsg` operator in turn calls the `msgCheckRules` operator in the `Controller` component. The `MessageInterface` accesses this operator by calling `IO_Interface.Controller.msgCheckRules`. Any return values associated with an operator are returned to the calling component. The `Operator`, `Rule`, and `Message` classes are all implemented as data structures and are associated with the components that use them. The OCL representation of the component operators is defined in Appendix D.

5.2 Knowledge Based Architectural Style

Knowledge based agents have been around for a number of years and although not as popular as they once were, they still represent a fundamental agent style. As the name implies, the central element in knowledge-based architectures is the knowledge base. The knowledge base

contains any predefined information the agent may need as well as any new information derived by the agent. As with reactive architectures, knowledge based architectures also use a set of rules to make decisions. Like the reactive architecture, these rules are also of an IF-THEN structure. Although the knowledge base is considered the key element in a knowledge-based system, the second most important component is the inference engine. The inference engine provides the facilities for navigating through and manipulating the knowledge in order to deduce something from it. The inference engine will use the rules and the knowledge base to reason over information and deduce results in an organized manner. Because the knowledge-based system normally has the ability to accept external queries and updates from other sources as well as react to changes in the environment, an input/output interface is also required. After analyzing the structure of a number of knowledge-based architectures, the architectural style of Figure 41 was defined. The main components comprising this style are `IO_Interface`, `RuleContainer`, `Controller`, `KnowledgeBase`, and `InferenceEngine`. Like the reactive style, the `IO_Interface` component encompasses all components that allow the agent to interact with other agents and the environment. Again, the definition of the component is to show how the interfaces are connected to the other components in the diagram. Similar to the reactive style, incoming information is forwarded to the `Controller` component so that it may be properly directed. The `Controller` is then able to direct the information to the `RuleContainer` (for updates) or the `InferenceEngine` (for queries) as appropriate. Because a query will normally need to be responded to, two-way inner connector is needed between the `Controller` component and the `IO_Interface` component. Once the `Controller` has accessed the `InferenceEngine` to solve the requested problem, it can take the solution and send it to the requestor of the message using the `MessageInterface`.

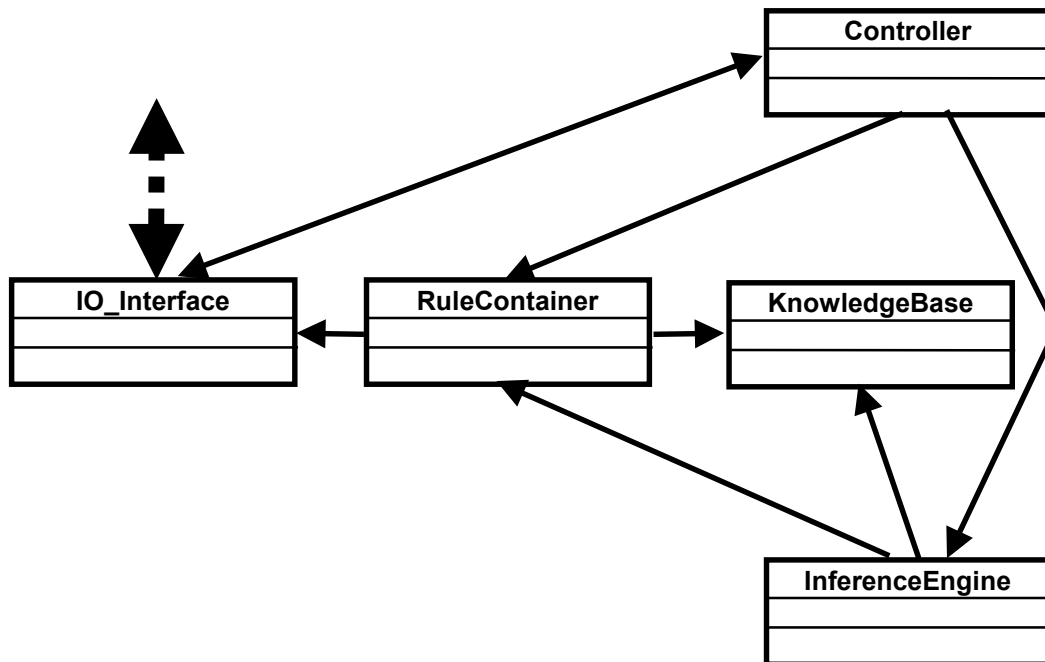


Figure 41 Knowledge Based Architectural Style

Because the **InferenceEngine** may need access to both the rules and the knowledge base to solve a problem, one way inner connectors go from **InferenceEngine** to both **RuleContainer** and **KnowledgeBase**. Since rules may be defined that update the **KnowledgeBase**, a one way inner connector connects the **RuleContainer** to the **KnowledgeBase**. Because the user may want a message sent back if the update succeeded or failed, the **RuleContainer** also has a one-way connector to the **IO_Interface**. Using Figure 41 along with the initial requirements allowed for the construction of the object model seen in Figure 42.

Because all of the interfaces of the knowledge base architecture are modeled in the exact same manner as those of the reactive architecture of Figure 38, for simplicity they are collectively represented as the **IO_Interface** class. The **RuleContainer** and **Rule** classes are essentially the same as those described in Section 5.1.

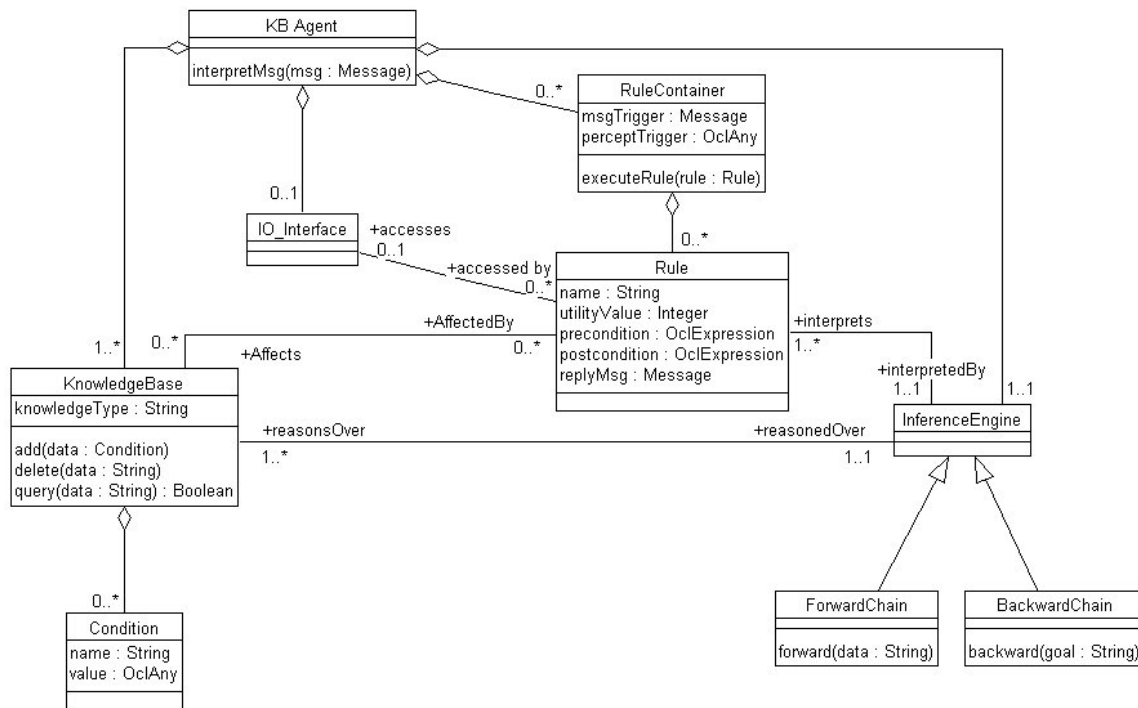


Figure 42 Knowledge Base Object Model

The `interpretMsg` operator of the `KB Agent` class directs the message based on the content and performative. For example, if the performative was `solve` and the content was `America has a king`, the message would be directed to the `InferenceEngine`. However, if the performative was `add` and the content was `sky=blue`, then the message would be directed to the `RuleContainer`. This is done because certain preconditions may need to be met before the knowledge base can be manipulated. The agent may only allow certain agents to modify its knowledge base, and may therefore have a rule that first checks the sender of any update message before it makes any changes. The `KnowledgeBase` class contains the `knowledgeType` attribute to delineate between multiple knowledge bases. This allows the user the ability to categorize knowledge in different repositories instead of using one centralized container for all information. The `add` method inserts a piece of data into the knowledge base while the `delete` method removes information from the knowledge base. The `query` method checks if a certain piece of

information is in the knowledge base and returns a true or false response as appropriate. The aggregate **Condition** class represents the actual data stored in the knowledge base. Because this data may be of varying types, **OclAny** is used to define the **value** attribute. The **value** attribute contains the actual data and can be identified by the **name** attribute. The **InferenceEngine** class provides the computational power needed to move through and manipulate the knowledge in order to effectively reason over new or existing data. Although many techniques exist for doing this, forward chaining and backward chaining are two of the most common, hence the definition of the **forward** and **backward** operators. The **forward** operator is passed the piece of information needing to be proven and then iterates through the rules and the knowledge base until it is proven or disproved. The **backward** operator is passed a goal and then attempts to find evidence for proving or disproving it. As previously stated, these two operators do not represent the only inference mechanisms in existence, therefore other inference classes may be added at the designers discretion. The transformation of the object diagram of Figure 42 into a component diagram is seen in Figure 43.

Like the reactive architectural style, a **Controller**, **IO_Interface**, and **RuleContainer** component are all defined. The **IO_Interface** is implemented in the same manner as Figure 39, while the substructure of the component is the same as that shown in Figure 40. The two new components are **KnowledgeBase** and **InferenceEngine**. An important aspect of the **KnowledgeBase** component is the **data** attribute. Because the **data** attribute of the knowledge base can be initialized by the user as well as modified by the agent, the attribute must be able to be both user defined and run time defined. This fact is captured by attaching '±' to the beginning of **data**. As with the reactive architecture, **Rule**, **Message**, and **Condition** data structures must be defined and associated with the appropriate components.

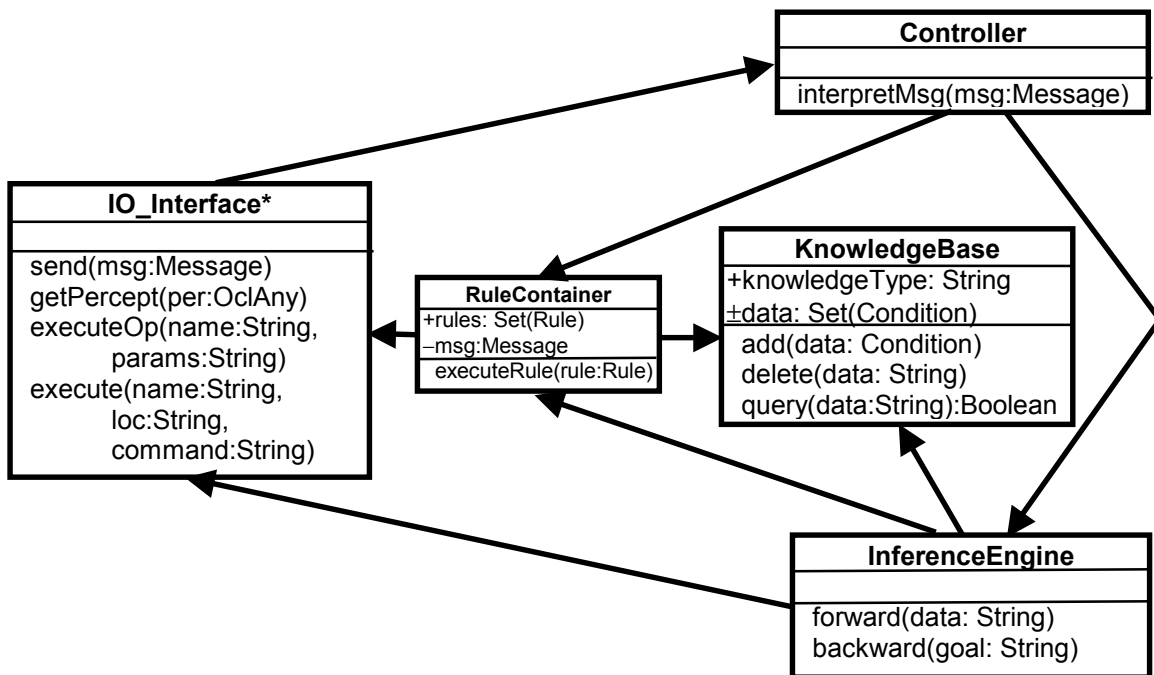


Figure 43 Knowledge-based Component Diagram

5.3 Planning Architectural Styles

As described in Section 2.3.6, there are two general types of planners used in most Artificial Intelligence systems today: dynamic planners and static planners. Dynamic planners take as input a goal, a set of operators, and the state of the agent and dynamically produce a plan to satisfy the goal. Static planners take as input a goal and the state of the agent, and use this information to choose an existing plan that best satisfies the goal. Although significantly different in how they execute, the overall architectural style is essentially the same. Both architectures contain a message interface used to receive messages from other agents. Messages contain the required information needed for the generation or selection of a plan. Because all required information is passed to the planner and because the only thing the planner is capable of doing is selecting or generating a plan, no other interfaces are required. In general, a plan consists of one or more steps, each of which contains a number of attributes. Each step contains

the name of the operator that must be executed, an ordering constraint, and a binding. The ordering constraint specifies which step or steps must be first executed before this step can execute. The binding is simply a list of parameters each operator will be passed. Plans generated by dynamic planners normally conform to this generic definition. Plans used by static planners are based on this plan definition, but add a number of attributes. Because the plans are already defined and must be chosen, a list of ‘goals satisfied’ is often associated with each plan. This list defines any goals that this plan may satisfy. Because multiple plans may satisfy the same goal, a utility value is also normally attached to each goal.

Based on this information the general architectural style for both types of planners is seen in Figure 44. The **MessageInterface** along with the two-way outer connector allows plan requests to be received, and plans to be sent back to requesting agents. Because the **Planner** is simply generating or selecting a plan, no access is needed to the **MessageInterface**.

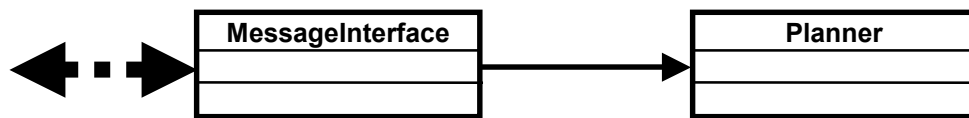


Figure 44 Planner Architectural Style

Therefore, a one-way inner connector connects the **MessageInterface** component to the **Planner** component. When the **MessageInterface** receives a request, the **Planner** will be called which will compute and return the plan to the **MessageInterface**, where it can then be sent to the requestor. The **Planner** contains some planning operator that will generate or select a plan based on the information passed to it. Each operator will have a return value that will be set equal to the plan generated or selected. Therefore, the **MessageInterface** is returned a plan once

the operator has completed execution, and is the reason a two-way inner connector is not required.

Although both planner types fall under one style, the object and component models differ significantly. For this reason, each planner type will be examined independently.

5.3.1 Dynamic Planner

Based on the information of Section 5.3 and Figure 44, the dynamic planner object model of Figure 45 was generated. The `DynamicPlanner` class acts as an interface to the planner being used. The `generatePlan` operator takes as parameters a goal, a list of available operators, and the current state of the agent. When implemented, the actual planner being used to generate plans will replace this component. The purpose of the operator is to capture the data the planner will require. This is not to say that a dynamic planner could not be specified from scratch using OCL, but for the purposes of this research, it is assumed a dynamic planner exists. The requests come in through the `MessageInterface`, which interprets the data and calls the `generatePlan` operator. The generated plan is returned to the `MessageInterface`, which in turn sends it to the requesting party. The `Plan` class represents the form the generated plan will be in. Each plan consists of one or more `PlanSteps`, which upon execution will satisfy the requested goal. The `op` attribute represents the operator that must be executed to complete the particular step.

The `Operator` type of this attribute is the same as that defined in Figure 38. The `ordering` attribute contains one or more operators that must have been evaluated before the current step can execute. If the `ordering` attribute is set to `NULL`, then that step represents the first step of the plan and can therefore be evaluated immediately. A key attribute to making this whole process work is `stepExecuted`. This boolean attribute keeps track of which steps have been executed.

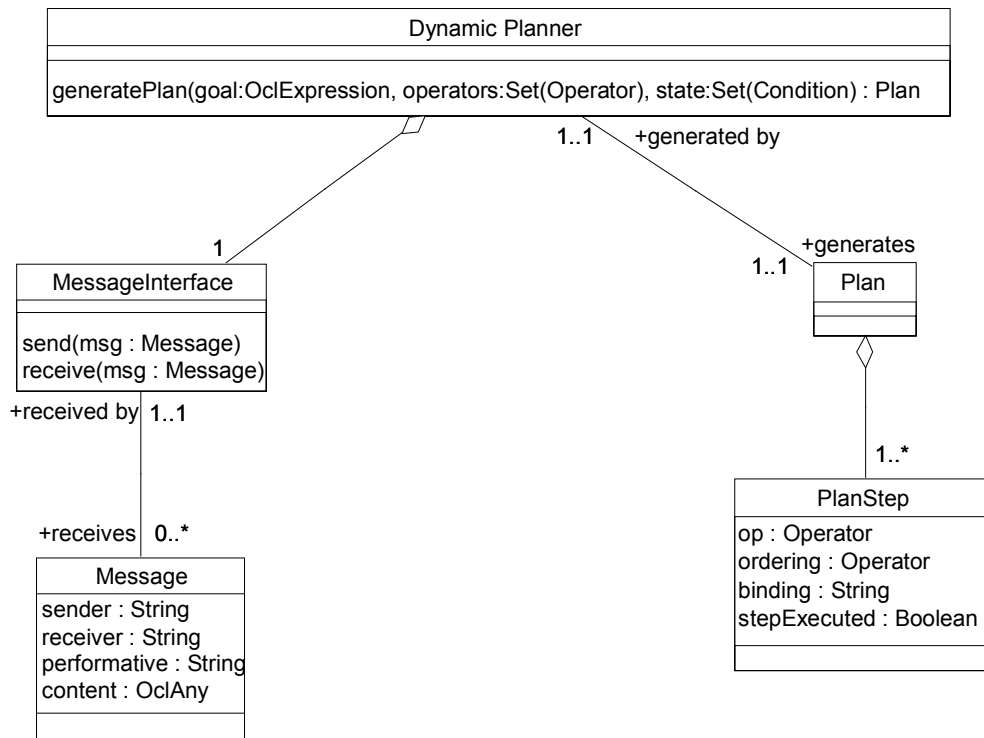


Figure 45 Dynamic Planner Object Model

For example, if a certain step is being examined for execution, and two operators are present in the ordering attribute, `stepExecuted` can be used to see if those operators executed. By taking the operator name, matching it to the `op` attribute in another plan step, the `stepExecuted` attribute will determine if the operator has been executed yet or not. This is not at all meant to imply that the planner controls the execution of a plan. This attribute is needed by the execution component that must exist in the agent that called the planner. Figure 46 depicts the component-based version of the object model. As seen from the figure, the only two components required are `MessageInterface` and `DynamicPlanner`. The conversion to the component diagram is rather straightforward with the only issue of importance being the definition of the data structures.

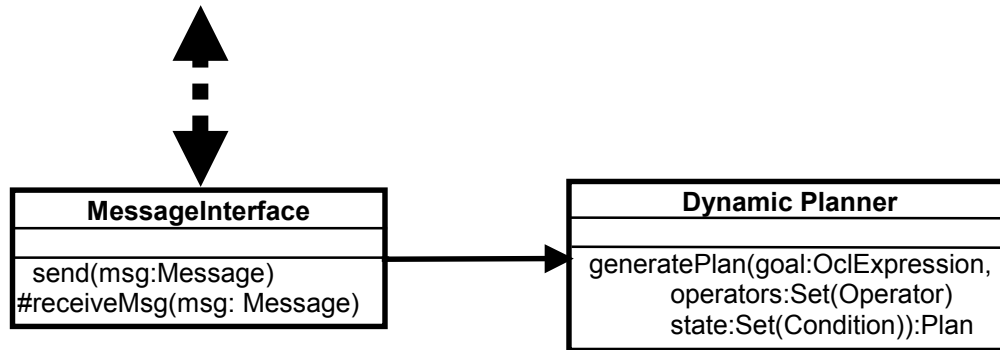


Figure 46 Dynamic Planner Component Diagram

Message, Operator, Condition, and Plan must all be defined and associated with the appropriate components.

5.3.2 Static Style

The major difference between the static planner and the dynamic planner is the fact that instead of generating a plan based on a set of conditions as in the dynamic planner, the static planner must choose from an existing set of plans based on a set of conditions. Figure 47 depicts the object model of the static planner. The **StaticPlanner** class contains the single operator **choosePlan**. This operator is passed as parameters a goal that must be achieved as well as the state of the environment. Based on this information, **choosePlan** will iterate through all plans it has access to and choose the most appropriate. As seen from the diagram, the plans used by this planning mechanism are an extension of those defined for the dynamic planner. The inherited class **StaticPlan** contains the additional attributes needed by the static planner. The **goals** attribute contains all information pertaining to what goals may be satisfied by the instantiation of the plan. The **utilityValue** attribute defines the applicability of the plan to the given situation. This attribute is used when more than one plan is found satisfying a particular goal.

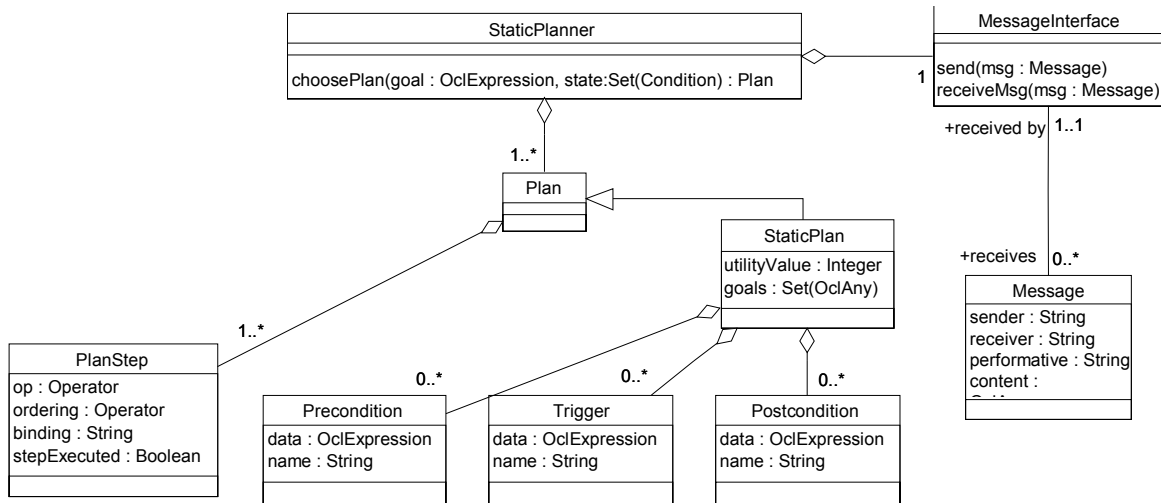


Figure 47 Static Planner Object Model

Because other conditions besides the overall goal must be taken into account when choosing a plan, the Precondition, Trigger, and Postcondition classes were required. The Trigger class is used to capture the fact that certain conditions may automatically trigger the selection of a particular plan. If certain conditions hold, the user may want a certain plan selected regardless of the overall goal. Precondition and Postcondition allow the user to add constraints to a particular plan. If certain conditions must be true before a plan can be selected, the Precondition class can store this information. If certain conditions must be made true once a plan has been executed, this information can be specified in the Postcondition class. Using the language defined in Chapter 4, Figure 48 depicts the component diagram.

5.4 BDI Architectural Style

As described in Section 2.3.2, the major attributes comprising a BDI architecture are beliefs, desires, intentions, and plans.

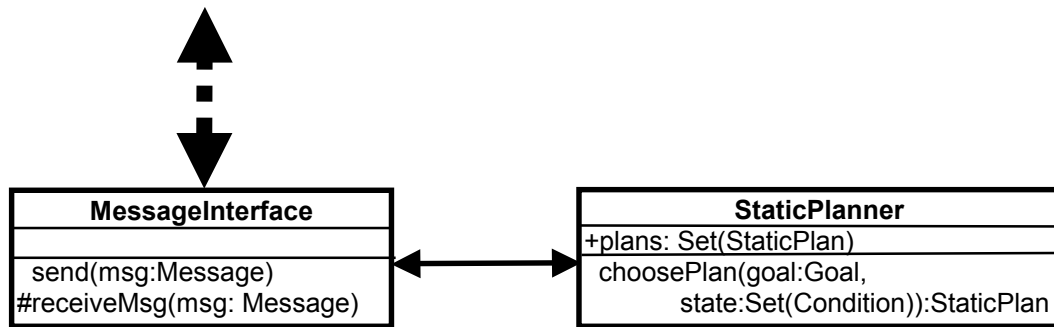


Figure 48 Static Planner Component Diagram

An agent may have multiple types of *beliefs*, such as beliefs about the environment or beliefs about other agents, but all this information is normally represented using declarative, logic-based statements. *Desires* correspond to the goals the agent must satisfy. The majority of architectures represent goals as a declarative statement or statements specifying the agent’s state once the goal is achieved. The user may define goals or the agent may adopt them. The *intentions* of the agent are simple to represent since they are just a list of plans that the agent will attempt to achieve. In order for an agent to achieve any of its desires, an existing plan must be chosen or a new plan generated that will satisfy the goal. Therefore, BDI agents can use static planners, dynamic planners, or both. The interfaces used in this type of architecture varied depending on the overall goal being achieved, but include sensors, effectors, message interfaces, and resource interfaces. A centralized controller is used to direct information as well as to control the execution of the agent’s intentions. A representation of this architectural style is seen in Figure 49.

As with the reactive architectural style, all interfaces are represented as a single `IO_Interface` component. Doing this shows how all interactions with other agents as well as with the environment are directed to the `Controller` component.

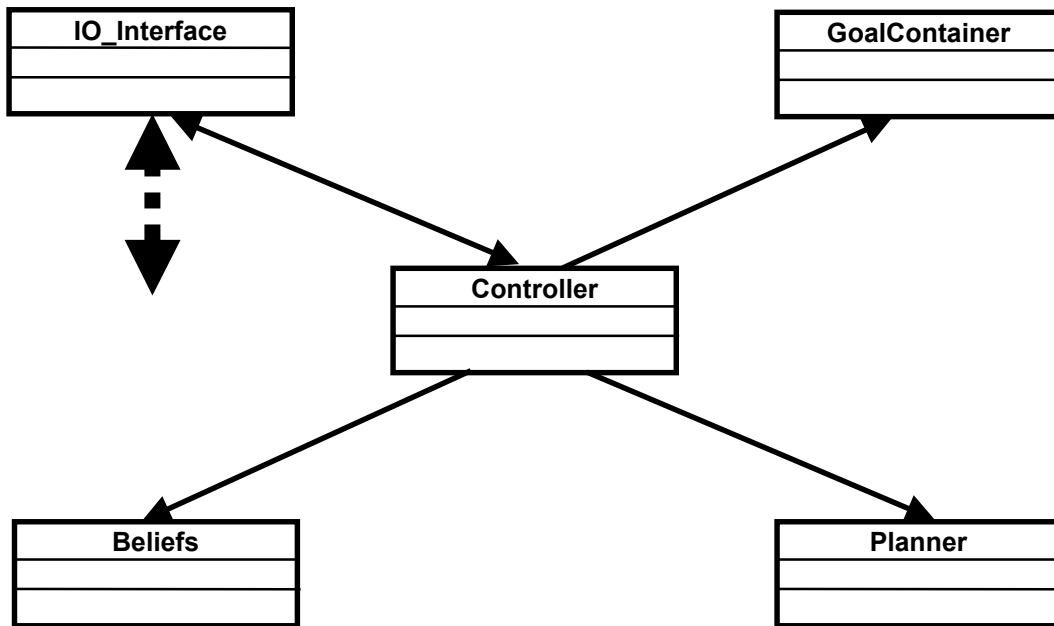


Figure 49 BDI Architectural Style

Because the overall execution is based in this centralized component, the **Controller** needs access to the resources of all components in system. Because the **Beliefs**, **GoalContainer**, and **Planner** are simply acting as resources, these components do not need access to any other components. It is for this reason that one way inner connectors connect the **Controller** to these components. Because intentions can be represented as a sequence of plans, they are stored within the **Controller** component and is the reason they do not show up in the diagram. The object model for this architectural style is seen in Figure 50. Because all of the interfaces of the BDI architecture are modeled in the exact same manner as those of the reactive architecture of Figure 38, they are omitted. The beliefs of the agent are captured in the **Beliefs** class of the figure. Because the type of information that may stored in an agents beliefs is the same type of information stored in a knowledge base, the **Belief** class is modeled in the exact same manner as the **KnowledgeBase** class of Figure 42.

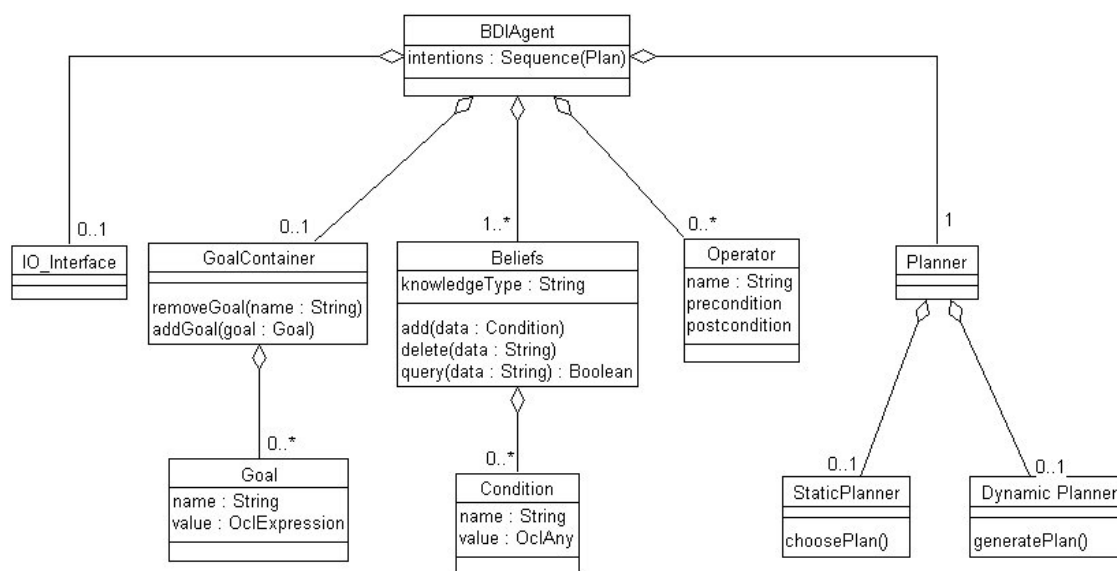


Figure 50 BDI Object Model

Also, because the planners used for this architecture are the same as those described in Section 5.3, the `StaticPlanner` and `DynamicPlanner` are modeled in the same manner as in Figures 46 and 48 (the operator signature is deleted from Figure 50 to save space). The only new classes not previously described are `GoalContainer` and `Goal`. The `GoalContainer` is a central repository for all goals defined for and generated by the agent. The `addGoal` and `removeGoal` operators are used for the addition and removal of goals from the repository. The `Goal` class consists of a `name` attribute for identification purposes and a `value` attribute to store the goal value. Figure 51 depicts the BDI component diagram. The `IO_Interface` component and its substructure is the same as that shown in Figures 39 and 40. Because either a static or dynamic planner can be used within this architecture, a generic `Planner` component was defined. This component contains the dynamic and static planner components of Figures 45 and 47 within its substructure. The substructure is shown in Figure 52. The operators `generatePlan` and `choosePlan` can then be used to call the operators of the respective components for the generation of or selection of a plan.

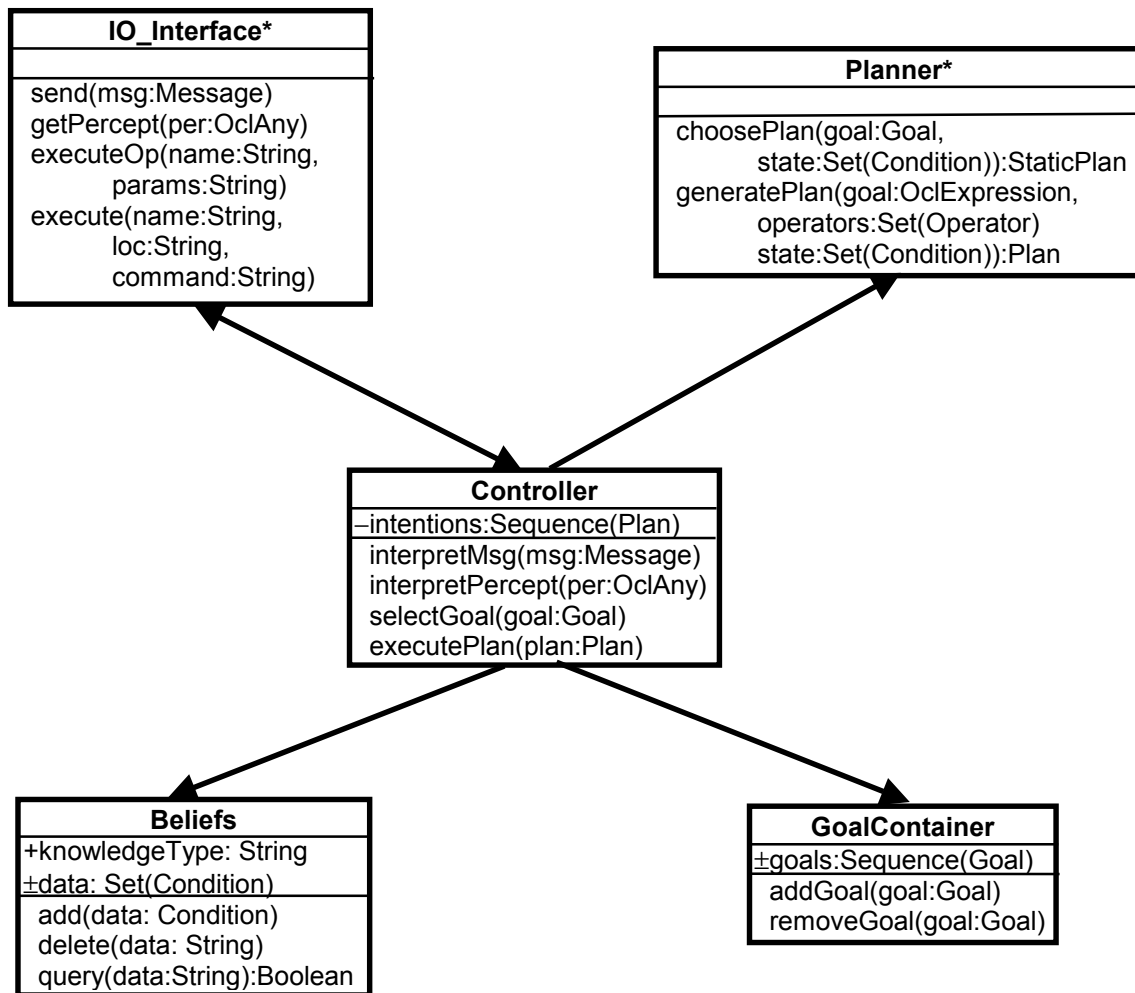


Figure 51 BDI Component Diagram

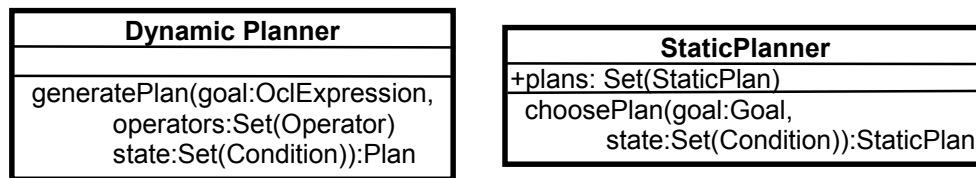


Figure 52 Planner Component Substructure

5.5 Generic Components

As a product of defining a set of architectural styles, a generic set of components was extracted that can be used to define many different agent types. Although each style represents

the ability to do drastically different things, it is readily apparent from the previous sections that there exist common components of which the majority of agents are a subset. Figure 53 represents the components extracted from the styles presented in Section 5.1 through 5.4. Note that because the **Beliefs** component of Section 5.4 was based on the same object model as the **KnowledgeBase** component of Section 5.2, both components are simply referenced as **KnowledgeBase** in the figure. The **IO_Interface** seen in many of the styles is represented in Figure 51 as the individual components that compose it.

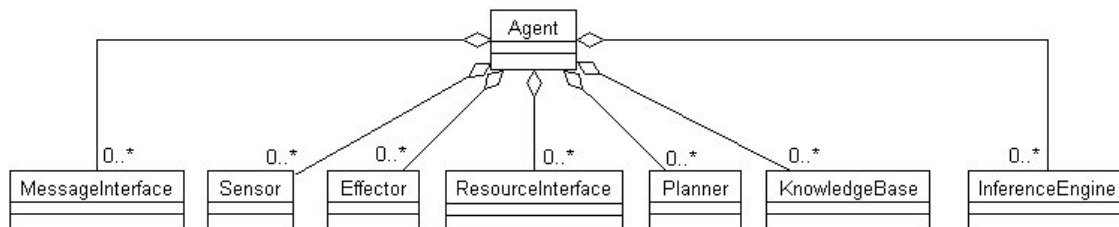


Figure 53 Component Baseline

The figure is not meant to imply that a working agent can be defined by simply connecting together a set of these components in an ad-hoc manner. Operators will need to be completely defined and additional attributes added to customize the components to the designers needs. The components selected are meant to act as templates from which large complex systems can be constructed.

5.6 Summary

This chapter defined a basic set of architectural styles that the majority of agent-based systems fall under. Each style was then modeled using the component language defined in Chapter 4. Doing this accomplished two things: it demonstrated the language was expressive enough to represent the most commonly used agent architectural styles, and it also built a set of templates a designer may use to more quickly and easily define an agent based-system. From

these templates, a basic set of components was extracted. These component templates provide the designer with building blocks that can be used to add to an existing style or to create a new style. As stated in Section 5.5, these templates are not all encompassing and should not limit the designer in any way. It may be necessary to define other styles and components in order to define a given system.

VI. Conclusions and Future Work

This thesis has presented the definition and implementation of a knowledge representation language that can be used to specify software systems. This chapter summarizes the conclusions of the previous chapters and presents ideas for future work that may be done with this research. Section 6.1 defines the conclusions of this thesis and Section 6.2 outlines possible future work.

6.1 Conclusions

The majority of this thesis has been concerned with the definition of a representation language to allow for the specification of software systems, specifically multi-agent software systems. The end result was a graphical and textual component-based language. This section outlines the key characteristics of the language and reviews the reasoning behind the approach used.

6.1.1 Use of Graphics and Text

Many software specification languages are strictly textual because of the ambiguity introduced by using graphics. The use of graphics within a specification does not allow any new information to be represented, but it does allow information to be presented in a way that is easier to comprehend. Using both graphics and text in a specification allow a problem to be viewed at two distinct levels of abstraction. The graphical view allows the overall structure of the system to be examined without being overwhelmed by specific details. The textual representation allows explicit details of every aspect of the system to be represented in a concise and unambiguous manner.

Object-oriented representation initially seemed like a logical choice for the required language. However although object-oriented techniques allowed for the correct level of detail, they provided the wrong level of abstraction. The premise of this research was that graphics would aid in understanding the specification by allowing the user to view the problem at a more manageable level of abstraction. Using object-oriented techniques was equivalent to taking a ten word sentence, putting an icon around each word, connecting the icons together and then saying the sentence is now easier to understand. The graphical representation was too detailed to allow for the desired level of abstraction. Component-based approaches allow for the desired graphical level of abstraction, but lacked text to provide the appropriate level of detail. To complement the component approach and to unambiguously specify and design software systems in a verifiable, efficient, and understandable manner, OCL was chosen.

6.1.2 Software Composition

It is important not to confuse the language defined in this thesis with a methodology for specifying software. This thesis does not propose an approach to specifying software systems, but does provide a means for doing so. Although the language was defined using many object-oriented techniques, this does not imply that these techniques must be used in defining the components used in this system. Any well-defined methodology for the specification of software can be followed when using the language.

6.2 Language Usage

The purpose of the language defined in this thesis is not to replace object-oriented techniques or ADLs, but rather to extend and enhance them. Object-oriented diagrams cannot capture architectural styles; however, using the approach described in Section 4.3, these diagrams can be transformed into component diagrams that may then be abstracted to an architectural style.

Traditional ADLs are not suited for specifying the internal behavior of components. However, using the language defined in this thesis, the internal behavior of components can be completely specified.

The overall approach to using the language is to examine the general characteristics of a problem and see if there is an existing architectural style to which the problem best conforms. If the problem can be described by an existing style, use that architectural style as a template for specifying and designing the system. The idea is to use the style to define a specific architecture by instantiating the style with particular values. Although components may need information added, deleted, or modified, the general structure of the style should aid in developing the system quickly and easily. Multiple styles can be combined, however, connectors may need to be added, deleted, and modified to allow for the two styles to be represented as one. Redundant and unused components may also have to be deleted. If multiple styles are used, styles can also be embedded in the substructure layer of components and connected together with inner-agent connectors.

If the problem cannot be categorized by an existing style, a new architecture may have to be defined. Using an object-oriented or other component based design methodology, the problem should first be broken down into basic components. Once these components have been identified, the designer should see if any predefined components exist matching the characteristics of the identified components. Components designed using the language should be stored in a common library that may be accessed by other designers to promote maximum reuse. In the *agentTool* environment, both architectural styles and components are stored in a central repository. Even if an existing component is not an exact match, the component can be modified to meet the requirement. If no existing component can be found, new components can be defined using the technique described in Section 4.3.

6.3 Possible Future Work

A number of areas in the research and implementation of this thesis still need more work. The purpose of this section is to identify and describe these areas.

6.3.1 Static Verification

The static representation of the system is the basic component diagram as shown in Figures 43. Verification of this portion of the system involves ensuring all information entered is of the appropriate type and is in the correct format. As mentioned throughout Section 4.5, the current implementation of the language lacks type and format checking in most areas. If an attribute is defined having a specific type, nothing is currently done to ensure any initial value assigned to that attribute is of that type. Besides type checking, the current implementation also does not do any format checking on most inputs. For example, when a user specifies an operator, one or more parameters may be defined that the operator can accept. These parameters should be entered in a certain format as described in Section 4.3. This format is not checked at this time. The final area needing work is the verification of OCL expressions. Like any language, OCL has a syntax and semantics that must be followed if the language is to be interpreted correctly. Currently no checking is done of any OCL expressions specified in the language. All of these issues become of critical concern if automatic code generation is ever to be achieved from the specification.

6.3.2 Variable Instantiation

Another area requiring work is component variable instantiation. The language is defined in such a way as to be used in two modes: specification and instantiation. The current implementation of the language only allows for specification. In the instantiation mode, the user

would be able to take the specification and instantiate the variables with specific values. For example, if the user specified a variable `instructors` that was of type `set(Instructor)`, in the instantiation mode the user could specify all the instructors belonging in this set. The object model defined in Figure 32 allows for the storage of any values associated with a variable, but currently the implementation does not. Type checking would also be required here to ensure the value associated with a given variable was of the same type as that specified.

6.3.3 Dynamic Verification

The overall dynamic representation of any system specified using the language defined in Chapter 4 is realized through the dynamic representation of each of its components. Because of this, verification must be done on the finite state machine of each component as well as the relationships between each component. Checking must be done to ensure there are no deadlocks or infinite loops within the system. Verification also needs to be done to ensure that all operators called from the state diagrams are currently implemented within the component diagram.

6.3.4 Code Generation

Before work in this area may begin, it is critical that the work of Section 6.2.1 through 6.2.3 be accomplished first. Until specific data values can be entered and until complete verification of the system can be done, automatic code generation should not be considered. This thesis provides the tools to allow for the automatic generation of code from specification, but currently does not provide the means.

6.4 Thesis Summary

The goal of this thesis, as outlined in Chapter 1, was to develop a knowledge representation language that can be used to unambiguously specify and design software systems

in a verifiable, efficient, and understandable manner. To ensure maximum understandability and ease of use, it was stated that the language should make use of both graphics and text to represent information. The language defined is a component-based specification language capable of representing all aspects of a software system in a formal and easy to understand manner. The graphical representation presents both the static and dynamic aspects of the system at an understandable level of abstraction, while the textual representation allows system details to be added in a precise and unambiguous manner. Completely defining the language object model allows for both extensibility and possible verification of a system specified using the language. Any specification composed according to the object model is now compatible with any past, present, or future specification defined according to the model. Because the object model formally defines the syntax and semantics of the overall language, the structure of the system specified can be verified for correctness. Because the dynamic representation is defined using finite state machines, verification of the behavior of the system can be done using techniques proposed by Lacey [LACEY00].

Although automatic code generation is not yet possible, the language contains the constructs that allow for this. The language can be used strictly for the specification of software systems or can be used to instantiate a specification with specific values. Although this portion of the language has not yet been implemented, the object model was defined in such a manner as to allow for this. Code generation issues such as variable and operator visibility and run time versus user-defined variable instantiation are embedded within the language.

The definition of architectural style and component templates do not make the language itself better, but do provide a designer with valuable examples of how to define specific components and styles. Designers may use the existing templates or may define their own templates to aid in the rapid development of system specifications.

Unlike many concept papers written on software specification languages, the vast majority of the language defined in this thesis has been implemented and integrated into the agentTool multi-agent development environment.

A number of languages have been defined for the specification of software systems. The language defined in this thesis has combined the best aspects of a number these specification languages to create a formal yet understandable language capable of specifying large software systems in an easy and precise manner.

REFERENCES

- [ACME97] Garlan, D., Monroe, R., & Wile, D., "Acme: An Architecture Description Interchange Language," *Proceedings of CASCON '97*, 1997.
- [AES95] Garlan, D., "An Introduction to the Aesop System," <http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps>, 1995.
- [BB98] Bigus, J.P. & Bigus, J., *Constructing Intelligent Agents with Java*, John Wiley & Sons, Inc., 1998.
- [BL85] Brachman, R. & Levesque, H., *Readings in Knowledge Representation*. Los Altos, Ca, Morgan Kaufmann, 1985.
- [BRK85] Brooks, R., "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol 2, pp. 14-23, 1985.
- [BOO94] Booch, G., *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings, 1994.
- [BH95] Bowen, J. & Hinchey, M., "Seven more myths of formal methods," *IEEE Software*, pp. 34-41, 1995.
- [CLIP98] CLIPS, What is CLIPS?, <http://www.ghg.net/clips/CLIPS.html>, 1999.
- [CM87] Cercone, N. & McCalla, G., *The Knowledge Frontier, Essays in the Representation of Knowledge*, Springer-Verlag New York Inc., 1987.
- [COOL95] Barbuceanu, M. & Fox, M., "COOL: A Language for Describing Coordination in Multi Agent System", in *Proceeding of the first International Conference on Multiagent Systems (ICMAS 95)*, 1995.
- [CS93] Coglianesi, L. & Szymanski, R., "DSSA-ADAGE: An Environment for Architecture-based Avionics Development," in *Proceedings of AGARD '93*, May 1993.
- [DMAR97] d'Inverno, M., Kinny, D., Luck, M., & Wooldridge, M., "A Formal Specification of dMARS," Tech. Rep. 72, Australian Artificial Intelligence Institute, Melbourne, Australia, November 1997.

- [DAF97] d'Inverno, M. & Luck, M., "Development and Application of a Formal Agent Framework," Proceedings of the First IEEE International Conference on Formal Engineering Methods, Hinchey and Shaoying (eds), 222-231, Hiroshima, Japan, 1997.
- [EP98] Eriksson, H. & Penkar, M., *UML Toolkit*, John Wiley & Sons, Inc., 1998.
- [FKV94] Fraser, M., Kumar, K., & Vaishnavi, V., "Strategies for incorporating formal specifications in software development." *Communications of the ACM*, vol. 37, pp. 74-86, 1994.
- [FMS97] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan and M. Wooldridge. "Formalisms for Multi-Agent Systems", *The Knowledge Engineering Review*, vol 12, 1997.
- [GK94] Genesereth, M. & Ketchpel, S., "Software agents," *Communications of the ACM*, vol 37, pp. 48-53, 1994.
- [GL87] Georgeff, M. & Lansky, A., *Reasoning About Actions & Plans – Proceedings of the 1986 Workshop*. Morgan Kaufmann Publishers, 1987.
- [GS96] Garlan, D. & Shaw, M., *Software Architecture Perspectives on an Emerging Discipline*, Prentice-Hall, Inc., 1996.
- [HODG91] Hodgson, J.P.E., *Knowledge Representation and Language In AI*. Ellis Horwood Limited, 1991.
- [JESS98] Friedman-Hill, E, "Jess, The Java Expert System Shell," Distributed Computing Systems, Sandia National Laboratories, ver 4.3, 1998.
- [JF99] Ferber, J., *Multi-Agent Systems An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman Inc., 1999.
- [KJ98] Knapik, M. & Johnson, J., *Developing Intelligent Agents for Distributed Systems*, McGraw-Hill Companies, Inc., 1998.
- [KM98] Kumar, D. & Meeden, L., "A Hybrid Connectionist and BDI Architecture for Modleing Embedded Rational Agents." Proceedings of 1998 AAAI Symposium on Cognitive Robotics, AAAI Press, 1998.
- [KWC98] Kleppe, A., Warmer, J., and Cook, S., "Informal formality? The Object Constraint Language and its application in the UML metamodel," *Proceedings of UML '98 International Workshop*, pages 127-136, 1998.
- [KS94] Kumar, D. & Shapiro, S., "The OK BDI Architecture", International Journal of Artificial Intelligence Tools, vol 3, number 3, pp 349-366, 1994.

- [LD95] Luck, M. & d’Inverno, M., “A Formal Framework for Agency and Autonomy,” *In Proceedings of the First International Conference on Multi-Agent Systems*, 254-260, AAAI Press/MIT Press, 1995.
- [LS98] Luger, G. & Stubblefield, W., *Artificial Intelligence Structures and Strategies for Complex Problem Solving*, Addison Wesley Longman, Inc., 1998.
- [MAE91] Maes, P., “The Agent Network Architecture,” *SIGART Bulletin*, 2(4):115-120, 1991.
- [MASE99] DeLoach, S., “Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems,” to be presented at Agent-Oriented Information Systems '99.
- [ML84] Mylopoulos, J. & Levesque, H., “An Overview of Knowledge Representation,” in Brodie et al, 1984.
- [MM97] Huhns, M. & Singh, M., *Readings In Agents*. Morgan Kaufmann, Inc., 1997.
- [MP92] Pollack, M., “The Use of Plans”, *Artificial Intelligence*, 43-68, 1992.
- [MYE78] Myers, G., *Composite Structured Design*, Van Nostrand Reinhold, 1978.
- [NASA95] “Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion”, NASA-GB-002-95, Release 1.0, 1995.
- [OCLW99] The Object Constraint Language OCL, the expression language for the UML, http://www.software.ibm.com/ad/standards/ocl.html/ocl_info.htm, 1999.
- [PB94] Popovic, D. & Bhatkar, V.P., *Methods and Tools for Applied Artificial Intelligence*, Marcel Dekker, Inc., 1994.
- [PDDL98] Ghallab, M., Howe, H., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., & Wilkins, D., PDDL – The Planning Domain Definition Language, Yale Center for Computational Vision and Control Tech Report, CVC TR-98-003/DCS TR-1165, 1998
- [PRES97] Pressman, R., *Software Engineering: A Practitioner’s Approach*, Fourth Edition, The McGraw-Hill Companies, Inc., 1997.
- [PROD92] “Prodigy 4.0: The Manual and Tutorial,” Prodigy Research Group, School of Computer Science, Carnegie Mellon University, 1992.
- [PRS99] “Procedural Reasoning System User’s Guide,” Artificial Intelligence Center, SRI International, 1999.

- [RAM88] Ramsey, A., *Formal Methods in Artificial Intelligence*, Cambridge University Press, 1988.
- [REICH91] Reichgelt, H., *Knowledge Representation An AI Perspective*. Ablex Publishing Co., 1991.
- [RG99] Gogolla, M., "A Metamodel for OCL," Second International Conference on the Unified Modeling Language: UML'99, 1999.
- [RN95] Russell, S. & Norvig, P., *Artificial Intelligence A Modern Approach*, Prentice Hall, Inc., 1995.
- [RUM91] Rumbaugh, J., Blaha, M., Premelani, W., Eddy, F., & Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [SHAP99] Shapiro, S., "SNePS 2.5 User's Manual", Department of Computer Science, State University of New York, 1999.
- [SWA94] Garlan, D. & Shaw, M., "An Introduction to Software Architecture," in V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1-39, Singapore, 1993. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.
- [SYC96] Sycara, K., Decker, K., Ananddeep, P., Williamson, M., & Zeng, W., "Distributed Intelligent Agents", *IEEE Expert/Intelligent Systems & Their Applications*, Vol. 11, No. 6, December 1996.
- [SZS95] Luck, M. & d'Inverno, M., "Structuring a Z Specification to Provide a Formal Framework for Autonomous Agent Systems," in *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users*, Jonathan Bowen and Mike Hinchey (eds.), Lecture Notes in Computer Science, 967, 47-62, Springer-Verlag, Heidelberg.
- [SZY98] Szyperski, C., *Component Software Beyond Object-Oriented Programming*, Addison Wesley Longman Limited, 1998.
- [UNI95] Shaw, M., DeLine, R., Klein, V., Ross, T., Young, D., & Zelesnik, G., "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314-335, April 1995.
- [WIN75] Winograd, T. "Frame Representations and The Declarative/Procedural Controversy", In *Representation and Understanding: Studies in Cognitive Science*. Edited by D. G. Bobrow and A. M. Collins. New York: Academic Press, 1975, 185-210.

- [WJ94] Wooldridge, M. & Jennings, N., "Intelligent Agents Theory and Practice," Available by FTP, 1994. Submitted to The Knowledge Engineering Review, 1995.
- [WK99] Warmer, J. & Kleppe, A., *The Object Constraint Language Precise Modeling with UML*, Addison Wesley Longman, Inc., 1999.
- [WM94] Wilkins, D. & Myers, M. "A Common Knowledge Representation for Plan Generation and Reactive Execution", SRI International Artificial Intelligence Center, 1994
- [WM97] Wilkins, D. & Myers, M. "The Act Formalism", SRI International Artificial Intelligence Center, 1997

APPENDIX A OCL GRAMMAR [WK99]

This appendix describes the grammar for OCL expressions. The grammar description uses the EBNF syntax, where "|" means a choice, "?" optionality and "*" means zero or more times [OCLW99].

```
expression :=  
    logicalExpression
```

```
ifExpression :=  
    "if" expression  
    "then" expression  
    "else" expression  
    "endif"
```

```
logicalExpression :=  
    relationalExpression  
    ( relationalOperator relationalExpression )*
```

```
relationalExpression :=  
    additiveExpression  
    ( relationalOperator additiveExpression )?
```

```
additiveExpression :=  
    multiplicativeExpression  
    ( addOperator multiplicativeExpression )*
```

```
multiplicativeExpression :=  
    unaryExpression  
    ( multiplyOperator unaryExpression )*
```

```
unaryExpression :=  
    ( unaryOperator postfixExpression )  
    | postfixExpression
```

```
postfixExpression :=  
    primaryExpression  
    ( ( "." | "->" ) featureCall )*
```

```

primaryExpression :=
    literalCollection
    | literal
    | pathName timeExpression? qualifier?
      featureCallParameters?
    | "(" expression ")"
    | ifExpression

featureCallParameters :=
    "(" ( declarator )? ( actualParameterList )? ")"

literal :=
    <STRING> | <number> | "#" <name>

enumerationType :=
    "enum" "{" "#" <name> ( "," "#" <name> ) * "}"

simpleTypeSpecifier :=
    pathTypeName
    | enumerationType

literalCollection :=
    collectionKind "{" expressionListOrRange? "}"

expressionListOrRange :=
    expression
    ( ( "," expression )+
    | ( ".." expression ) )?

featureCall :=
    pathName timeExpression? qualifiers?
    featureCallParameters?

qualifiers :=
    "[" actualParameterList "]"

declarator :=
    <name> ( "," <name> ) * ( ":" simpleTypeSpecifier )? "]"

pathTypeName :=
    <typeName> ( "::" <typeName> ) *
pathName :=
    ( <typeName> | <name> )
    ( "::" ( <typeName> | <name> ) ) *

timeExpression :=
    "@" <name>

```

```

actualParameterList :=
    expression ( "," expression ) *

logicalOperator :=
    "and" | "or" | "xor" | "implies"

collectionKind :=
    "Set" | "Bag" | "Sequence" | "Collection"

relationalOperator :=
    "=" | ">" | "<" | ">=" | "<=" | "<>"

addOperator :=
    "+" | "-"

multiplyOperator :=
    "*" | "/"

unaryOperator :=
    "-" | "not"

typeName :=
    "A"-"Z" ( "a"-"z" | "0"-"9" | "A"-"Z" | "_" ) *

name :=
    "a"-"z" ( "a"-"z" | "0"-"9" | "A"-"Z" | "_" ) *

number :=
    "0"-"9" ( "0"-"9" ) *

string :=
    "" ( ( ~["'", "\\", "\n", "\r"]
        | ( "\\" ( ["n", "t", "b", "r", "f", "\\", "'", "\"]
            | ["0"-"7"] ( ["0"-"7"] ) ?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
        )
    ) *
    ""

```

APPENDIX B FIGURE 19 OPERATOR DEFINITION

The general definition of an OCL operator is as follows:

Type1::operation(arg : Type2) : ReturnType
pre: --precondition expression goes here
post: --postcondition expression goes here with optional result variable

The expression specified after **post** is only evaluated if the expression specified after **pre** evaluates to true. Postconditions may contain two optional keywords to represent time: **result** and **@pre**. The **@pre** keyword is used to indicate the value of an attribute or association at the beginning of the execution of the operation. The keyword is postfixed to the name of the item it is associated with. The **result** keyword is used to indicate the return value for the operation [WK99]. Because a user may want to have a number of things to happen when the postcondition is evaluated, OCL expressions can be conjuncted together. Although OCL does not specify any order for the evaluation of conjuncted expressions, for the purposes of this thesis, it is assumed that expressions are evaluated in a top to bottom, left to right fashion. For example, the postcondition of the `executeMsgValid` operator of `RuleContainer` (specified below) contains two OCL expressions conjuncted together. The `msgTrigger=msg` expression will be evaluated first, followed by the `rule->forAll(x:Rule | x.execute)` expression. Listed below are the operator signatures for Figure 19.

MessageInterface::receiveMsg(msg: message)
pre: --none
post: ReactiveAgent.CheckRules(msg)

ReactiveAgent::CheckRules(msg: message)
pre: --none
post: RuleContainer.executeMsgValid(msg)

RuleContainer::executeMsgValid(msg: message)
pre: --none
post: msgTrigger=msg
and rule->forAll(x:Rule | x.precondition=>x.postcondition)

APPENDIX C LANGUAGE IMPLEMENTATION

This appendix shows how the language described in this thesis was implemented in the multi-agent system *agentTool*. What follows is a step by step description of how components needed for the reactive architecture described in Section 4.3 are created and saved in the agentTool multi-agent development environment.

It should be noted that I designed the language and coded the basic components (Figures 56 through 62). The coding of the connectors and component state diagrams (Figures 63 through 70) was accomplished by Jennifer Mifflin, an undergraduate research assistant.

Before any components can be specified, an agent must first be created. From the main agentTool dialog, the user must select the **Add Agent** button to add an agent to the screen. Once done, an additional tab will be created containing the agent name. Figure 54 shows how this is represented in agentTool. To change the agent name, the user must use the mouse to select within the **Currently Selected** text box. Once there, the user can change the name of the agent by replacing **Agent1** with the desired name. For the purposes of this example, the name **Reactive** will be used. Although multiple agents can be created and connected, only one agent will be examined for this example. Once the agent is created, the user can then select the tab containing the agents name (in this case **Reactive**). Once this tab is selected, the user will be presented with the inner agent screen seen in Figure 34. The internal representation of the agent is defined within this tab. The user can now begin specifying the components making up the particular agent type. Adding a component is done by selecting the **Add Component** button. A generic component will automatically be added to the window. By selecting a component and pushing the right mouse button, the component menu of Figure 56 is presented to the user.

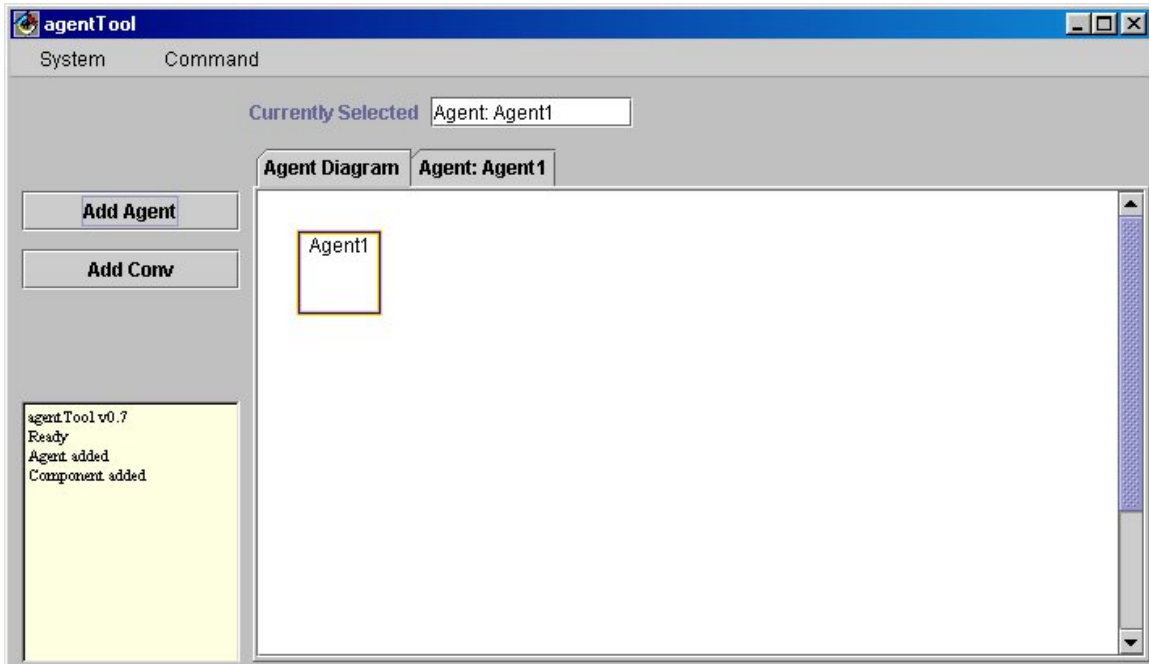


Figure 54 Creating an Agent

This menu allows the user to define all the static characteristics of the given component to include name, attributes, and operators. Selecting **Properties** will bring up the dialog seen in Figure 57 thus allowing the user to change the component name and designate whether it contains a substructure. Selecting the **Substructure** check box indicates that the component will contain a substructure. The default is not selected. Defining substructures within agentTool is not available in the current version of the software. An asterisk is appended to the name of any component designated as containing a substructure.

Selecting **Add Attribute** from the component menu allows the user to define component attributes one at a time. When the option is selected, the user is presented with the dialog box shown in Figure 58. The **Attribute Name** field is used to specify the name of the attribute being defined.

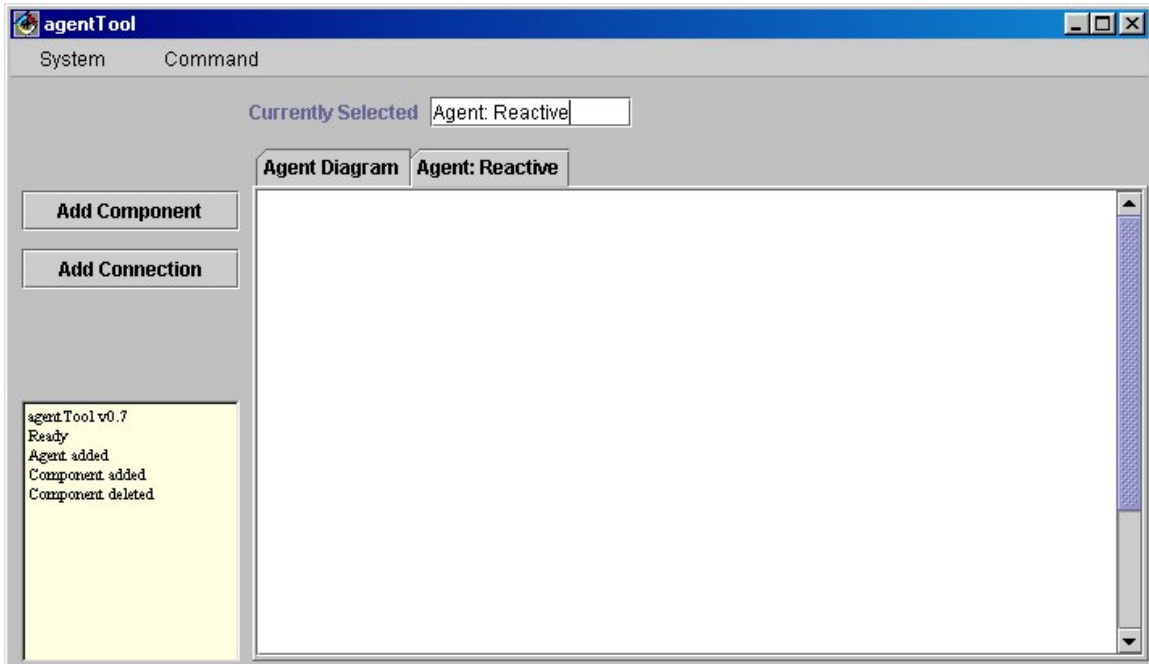


Figure 55 Internal Agent Panel

The **Attribute Type** field can contain any of the predefined OCL types as well as any user-defined types. The current version of agentTool does not allow the specification of user-defined types. The **Initial Value** field is optional, but can be used to set the attribute to a default value. Currently, no type checking is done within agentTool to ensure the type of **Initial Value** matches **Attribute Type** or to ensure **Attribute Type** is valid. If the attribute is a collection of objects, the user can specify it as a **Set**, **Sequence**, or **Bag** by selecting the appropriate radio button. The software enforces the constraint that only one of these types be selected at any given time. The attribute of Figure 58 will be represented as `rule:Set(Rule)`. The user can then choose whether the attribute is user defined, run time defined, or both. These selections will be used for the code generation portion of agentTool (not yet implemented). Once completed, the user selects the **Add** button. Figure 59 displays the **RuleContainer** once all attributes have been added.

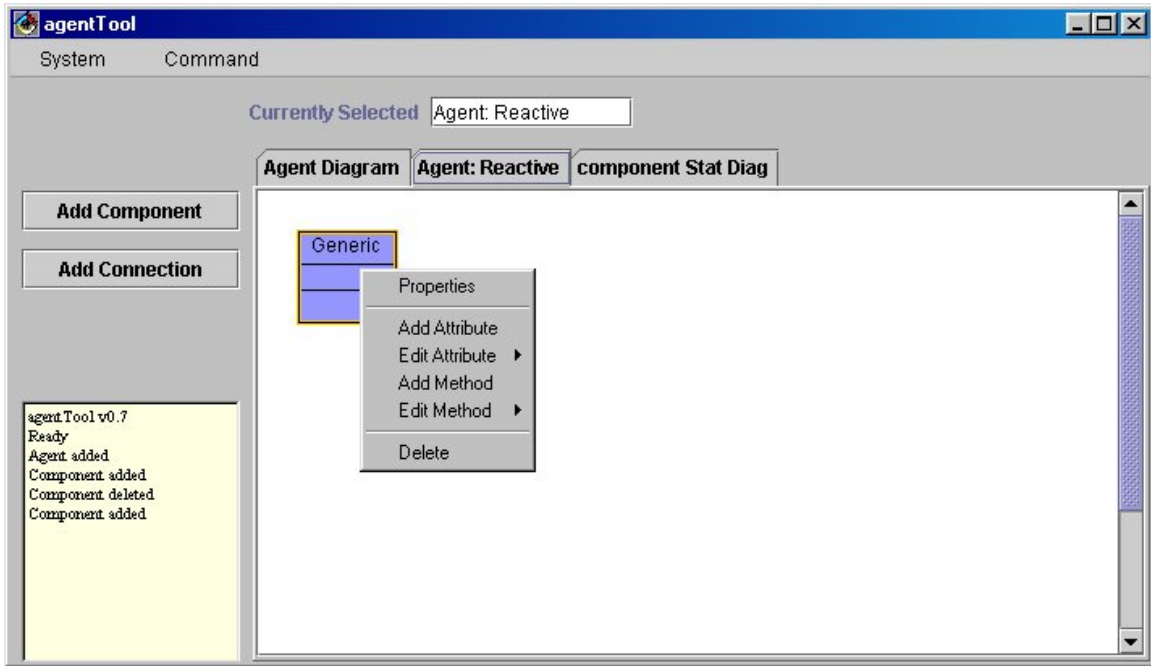


Figure 56 Component Menu



Figure 57 Component Properties

The '+' symbol indicates an attribute is user defined, while the '-' symbol indicates an attribute is run time defined. Although not shown in the figure, the '±' indicates an attribute that is both run time and user defined. If the user later wants to edit any of the attributes previously defined, they can go to the Edit Attribute option of the component menu. Once there, the user will be presented with a drop down menu containing the names of all attributes defined at that time.

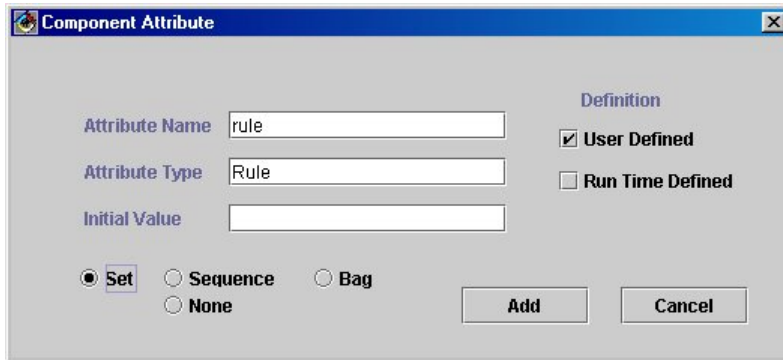


Figure 58 Component Attribute

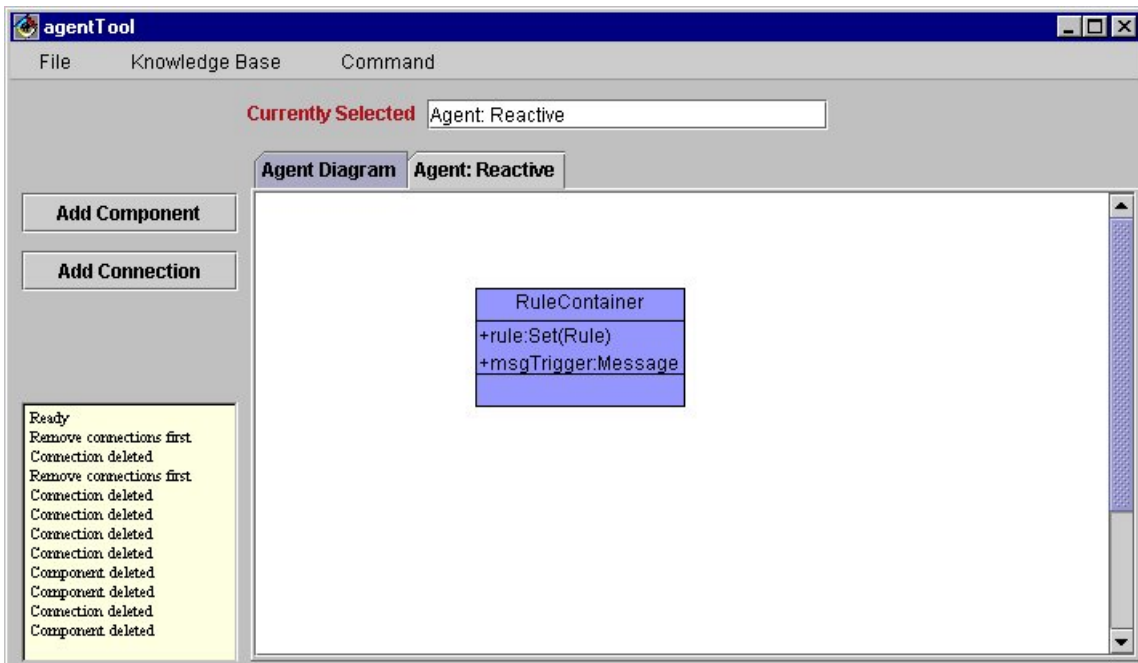
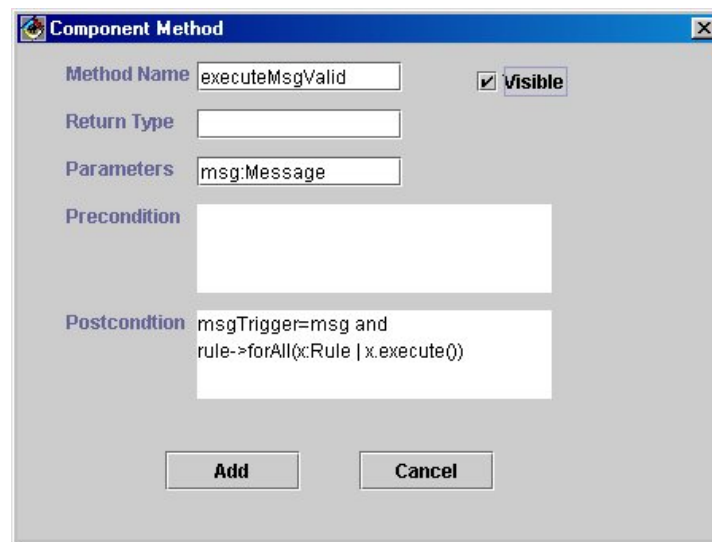


Figure 59 RuleContainer Attributes

The user can then select the attribute they wish to edit and will be presented with a dialog box similar to that of Figure 58 except with all values filled in and the **Add** and **Cancel** buttons replaced by **Delete**, **Change**, and **Cancel** buttons. The user can then make any desired changes to the attribute or delete the attribute.

Once all attributes have been defined, the user can then specify any operators needed for the component. Selecting **Add Method** from the component menu brings up the dialog seen in Figure 60. The **Method Name** is used to designate the particular operator being defined. Although not currently checked for, methods within the same component should not have the same name. The **Return Type** field is used to specify the type of data (if any) being returned by the method. Like **Attribute Type**, this field should contain a valid OCL or user defined type. The **Parameters** field is used to specify data to be passed to the method.



The screenshot shows a dialog box titled "Component Method". It contains the following fields and controls:

- Method Name:** A text box containing "executeMsgValid".
- Return Type:** An empty text box.
- Parameters:** A text box containing "msg:Message".
- Precondition:** An empty text box.
- Postcondition:** A text box containing the OCL expression: "msgTrigger=msg and rule->forAll(x:Rule | x.execute())".
- Visible:** A checked checkbox.
- Buttons:** "Add" and "Cancel" buttons at the bottom.

Figure 60 Rule Container Method

The format for entering data in this field is **parameter name:parameter type**. Commas should separate multiple parameters. No type checking is currently done on the format of this field or on whether the parameter types are valid. The **Precondition** and **Postcondition** fields are for entering OCL expressions describing the pre- and postconditions of the method. The **Visible** checkbox is used to specify the operator's visibility to other components. If the box is checked then the operator may be accessed by other components. Operators not visible are preceded by

the '#' symbol. Figure 61 depicts the RuleContainer once all attributes and methods have been defined.

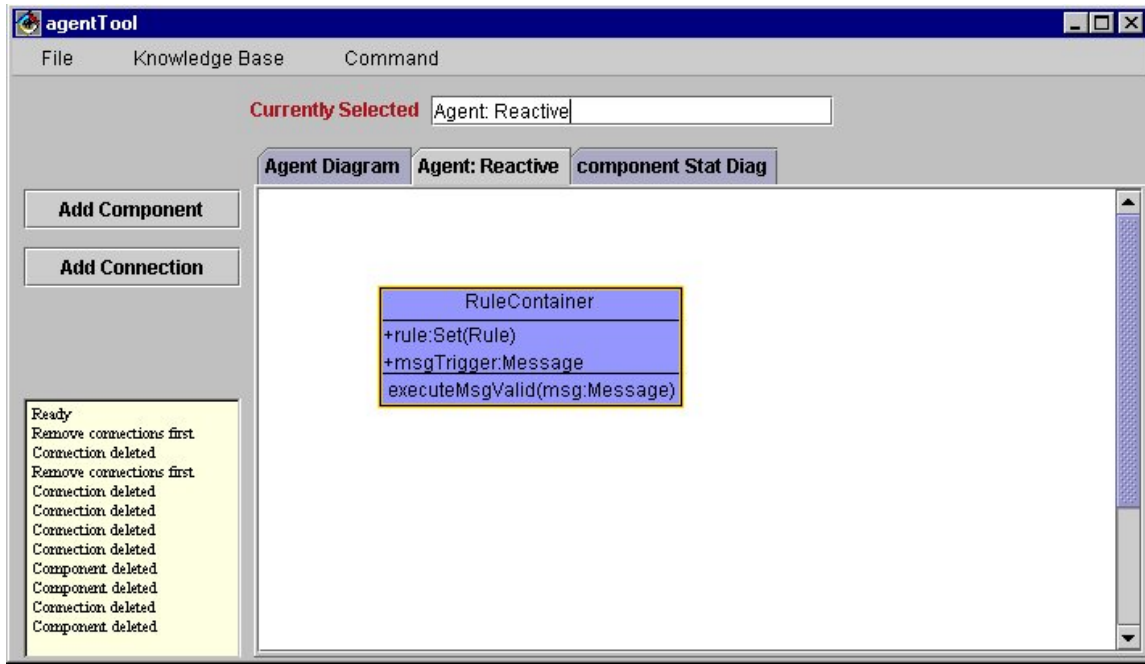


Figure 61 RuleContainer Component

The Edit Method option of the component menu works in a similar manner to that of the attribute, allowing the user the ability to change or delete any methods. All remaining components can then be defined following the same procedure. Figure 62 shows the Reactive agent panel once all components have been defined. Once one or more components have been defined, the user may wish to link a component to the agents environment or to another component. To do this the user simply selects the Add Connection button. To define an outer agent connector, the user must then simply use the left mouse button and double click on the desired component. Doing so will automatically add a one way outer agent connector to the component.

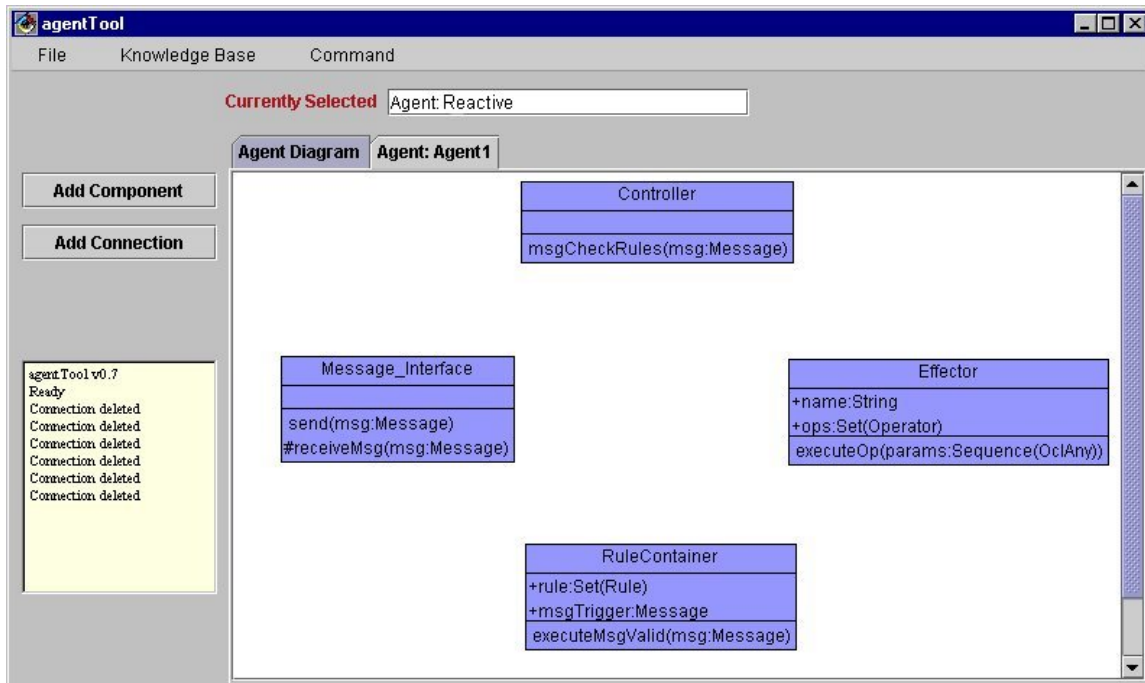


Figure 62 Reactive Agent Components

By right clicking on the connector, a drop down box will appear with four options (as seen in Figure 63). Selecting the first option, **Message_Interface**, would reverse the direction of the connector so that it was pointing at the **Message_Interface** component. Choosing the second option, **Framework**, points the connector away from the component, towards the environment framework. Selecting the **BothArrows** option defines a two-way outer agent connector. The **Delete** option simply removes the selected connector.

The process for adding an inner agent connector is very similar, except that once the **Add Connection** button is selected, the user must choose the two components that are to be connected. Once the second component is chosen, a one way inner agent connector will be drawn from the component selected first to the component selected second.

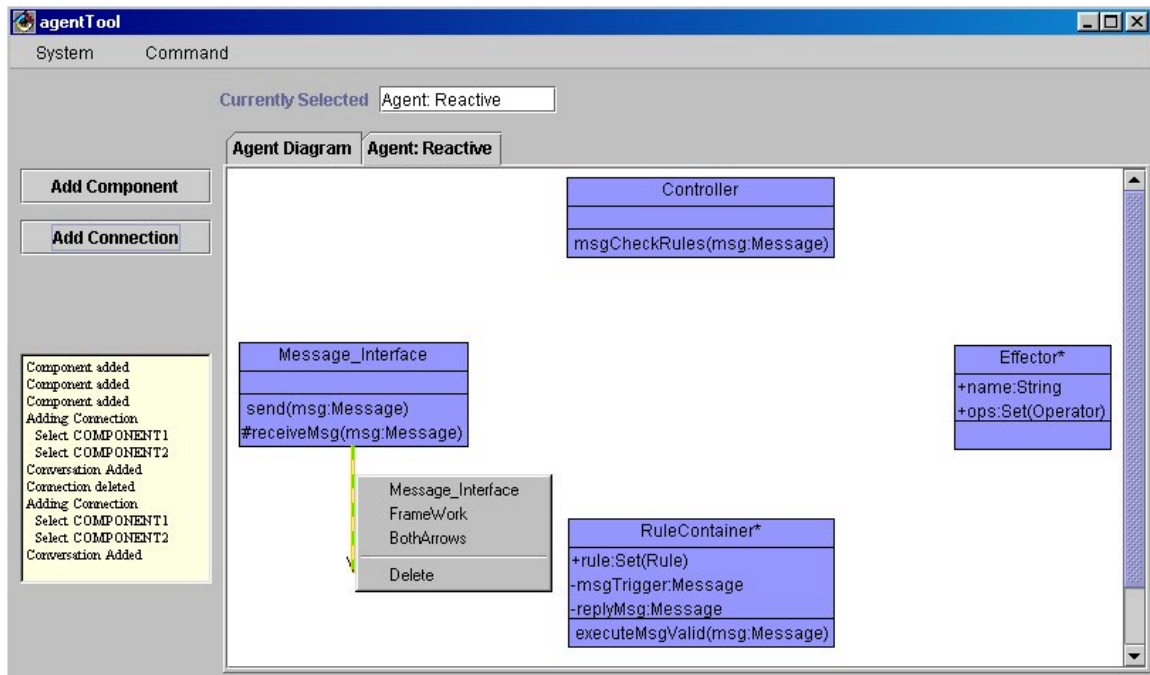


Figure 63 Outer Agent Connector

To change the orientation of the connector, the user right clicks on the connector and is presented with a set of options similar to those of the outer agent connector. The options will include the first components name, the second components name, **BothArrows**, and **Delete**. Selecting either component name will make the connector point to that particular component. **BothArrows** and **Delete** work in the same manner as outer agent connectors. Figure 64 depicts the reactive agent components completely connected. The thicker, dashed arrows represent the outer agent connectors, while the thin, solid arrows represent the inner agent connectors. The static representation of the agent is now complete.

To define the dynamic representation of the agent, state diagrams must be specified for each component. To specify the state diagram for a specific component, the user must select a component by left clicking on it. Once accomplished, another tab will appear with the name component Stat Diag. This tab will represent the state diagram for the selected component.

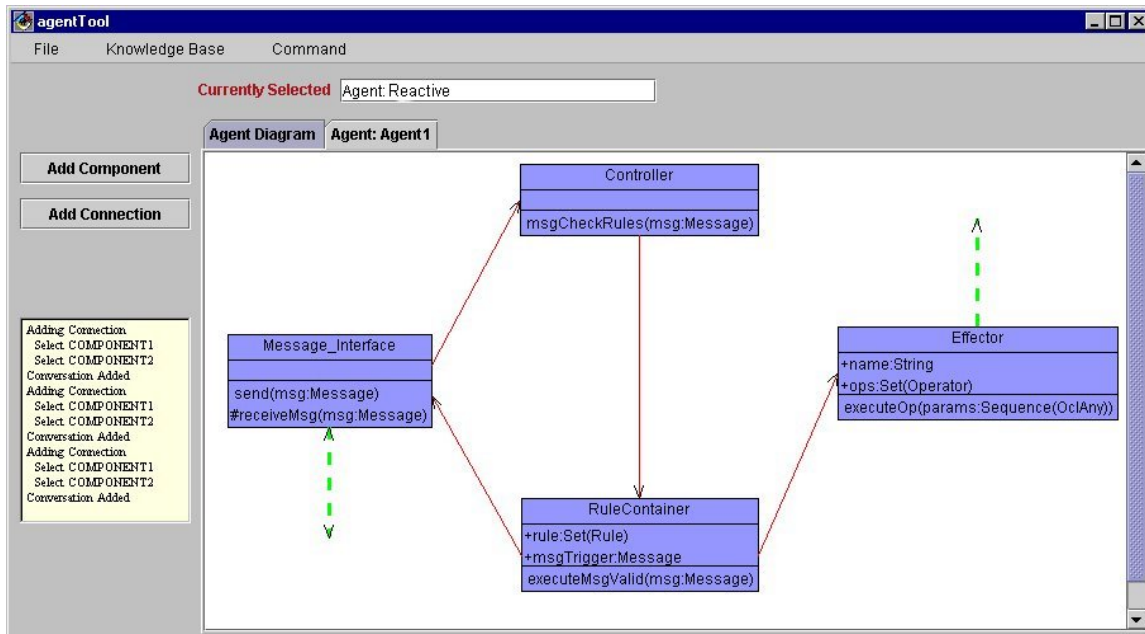


Figure 64 Component & Connectors

Selecting the tab will display the state diagram panel as seen in Figure 65, where the solid circle represents the start state, and the circle within a circle represents the end state. Adding a state is simply done by selecting the **Add State** button. Once defined, a state can be customized by right clicking on the state and selecting **Properties** from the pull down menu. The user can then change the name of the state as well as specify the action associated with the state. Figure 66 shows Figure 65 with a **Wait** state added. Once a state has been created, transitions can be created in one of two ways. If the transition goes between two states then the user must select the **Add Trans** button and then select the two states that the transition will go between. If the transition loops back on itself, the user must select the **Add Trans** button and then double click on the state having the loop back. For example, to add a transition from the start state to the **Wait** state, the user must select **Add Trans** and then select the start state followed by selecting the **Wait** state. To add a transition from the **Wait** state back to the **Wait** state, the user selects **Add Trans** and then double clicks on the **Wait** state.

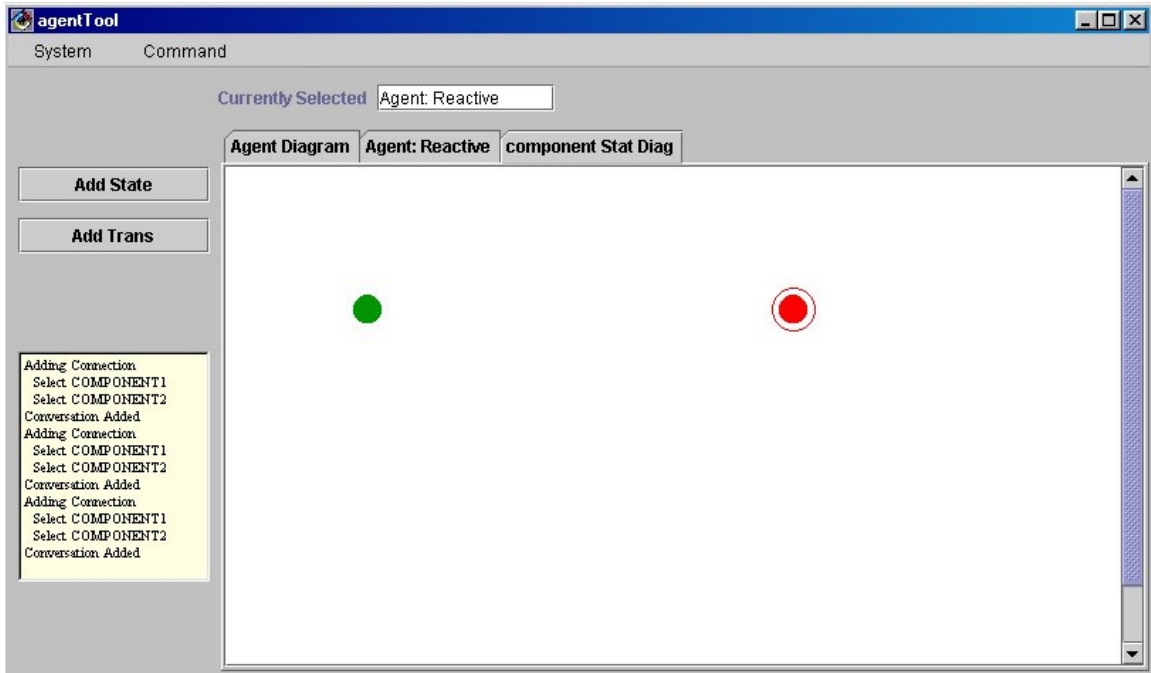


Figure 65 Component State Diagram

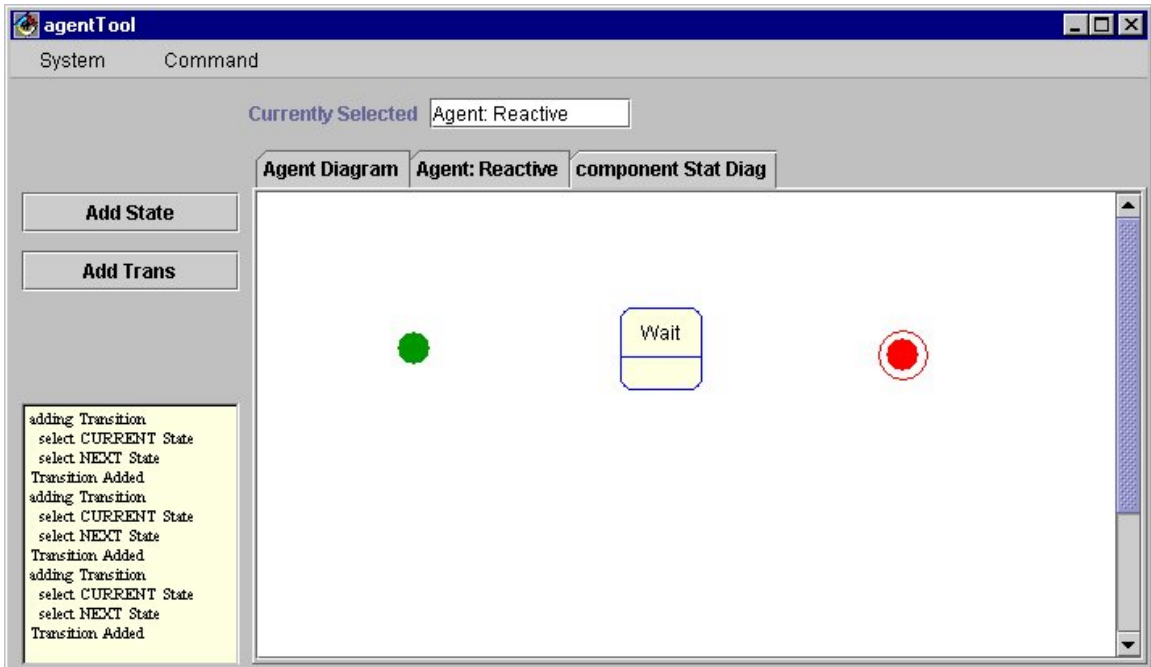


Figure 66 Wait State Added

Figure 67 depicts the diagram after this has been accomplished. To correctly model the Message_Interface components state diagram depicted in Figure 24 of Section 4.3.2.2, two transitions are needed from the Wait state to the Wait state. Currently agentTool does not offer a robust way to handle this situation. To model this second transition, a “dummy” state must be added for the state to transition to. A depiction of this workaround is shown in Figure 68.

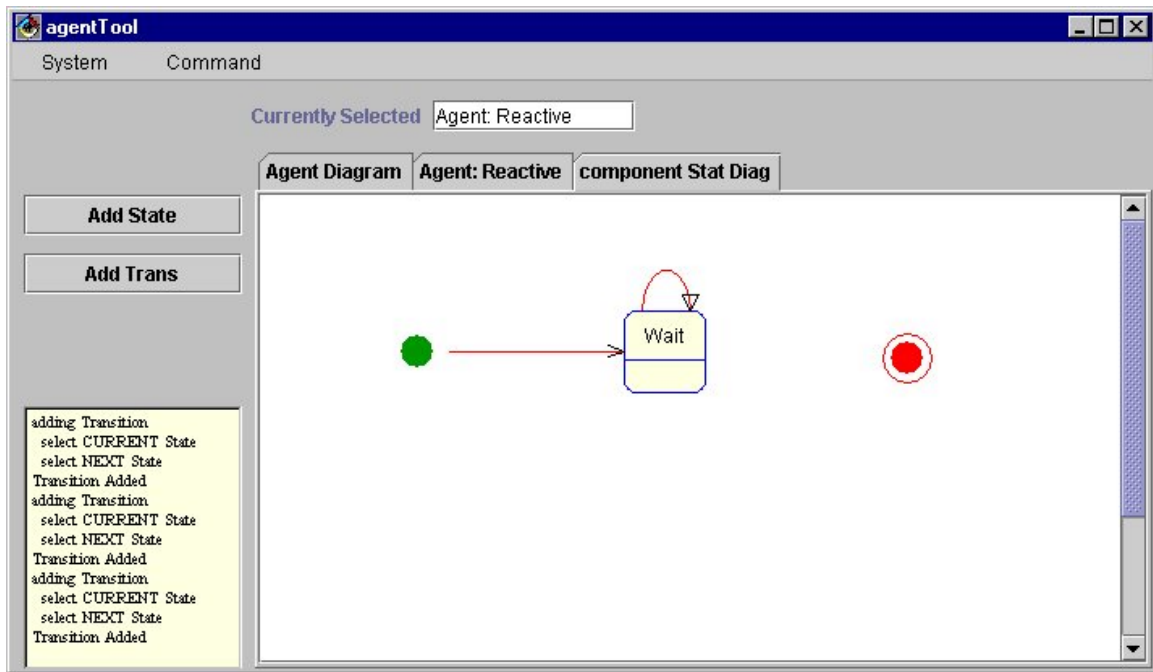


Figure 67 Wait State with Transitions

Once all states and transitions have been defined, the last thing that must be done is to define the transition characteristics. By selecting the transition and right clicking on it, the user is presented with the transition menu seen in Figure 69.

Selecting the **Properties** option allows the user to specify the received message, guard conditions, send message, and action for each transition. The **Reverse** selection simply reverses the direction of the transition and the **Delete** option removes the transition. Remaining options are not used in this research.

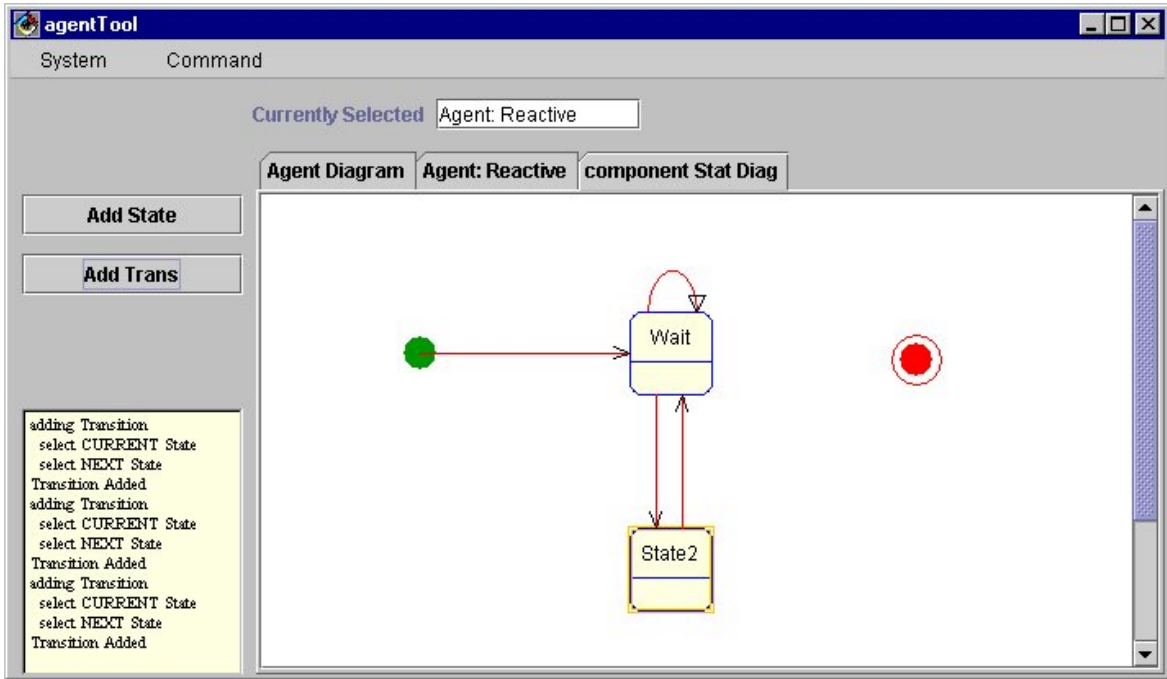


Figure 68 Dummy State

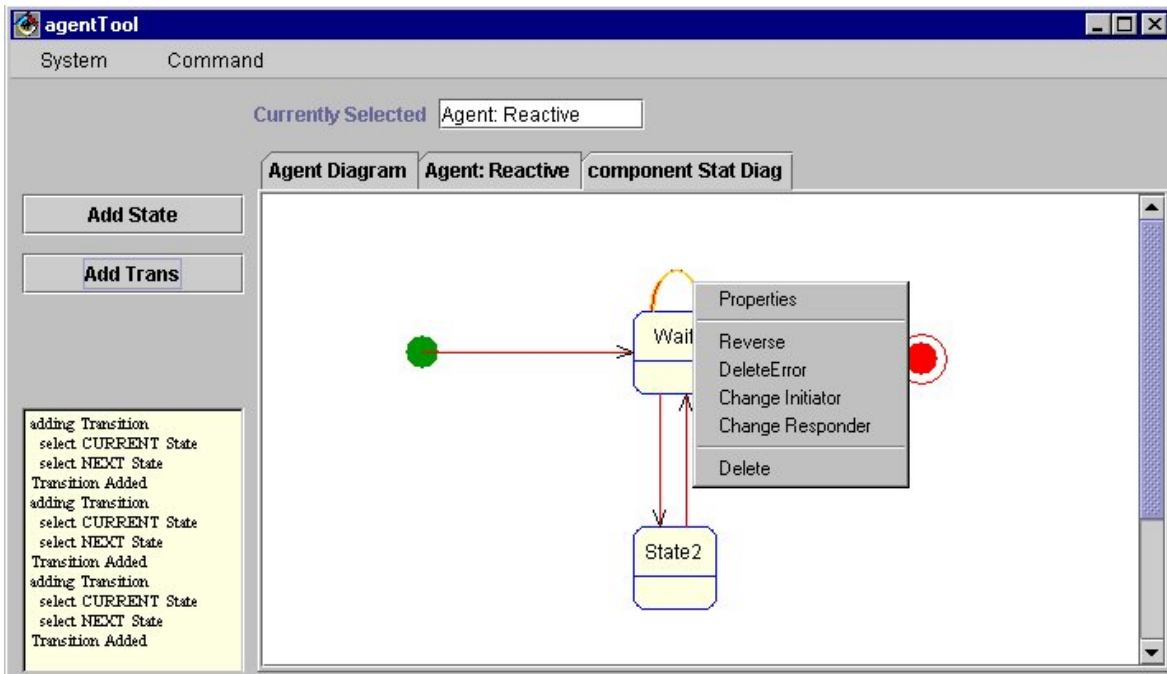


Figure 69 Transition Menu

Figure 70 depicts the completed state diagram. It should be noted that the `send(message)` event triggers the transition from the Wait state to State2. It is an automatic transition from State2 to the Wait state.

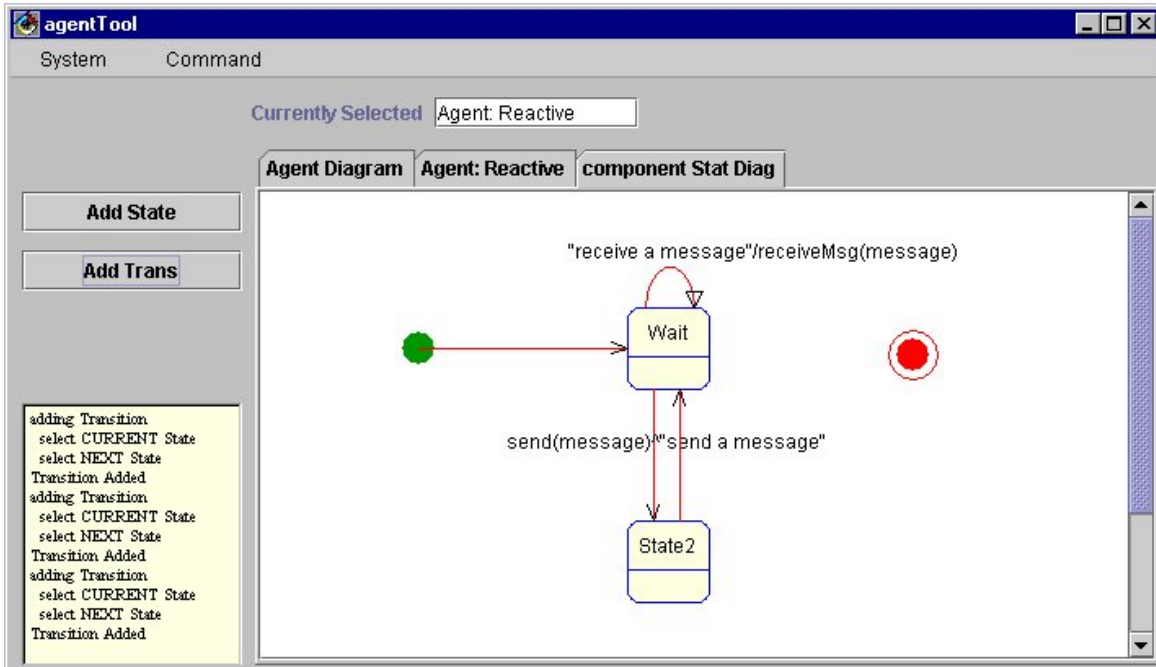


Figure 70 Completed Dynamic Model

APPENDIX D REACTIVE STYLE OPERATOR DEFINITION

The operator definition defined in Appendix B is followed throughout this appendix.

IO Interface::send(msg: message)

pre: --none

post: MessageInterface.send(msg)

IO Interface::getPercept()

pre: --none

post: Sensor.getPercept()

IO Interface::executeOp(name:String, params:String)

pre: --none

post: Effector.executeOp(name, params)

IO Interface::execute(name:String, loc:String, command:String)

pre: --none

post: ResourceInterface.execute(name:String, loc:String, command:String)

MessageInterface::receiveMsg(msg: message)

pre: --none

post: IO_Interface.Controller.msgCheckRules(msg)

Sensor::receivePercept(per: OclAny)

pre: --none

post: IO_Interface.Controller.perceptCheckRules(per)

Effector::executeOp(opname:String, params:String)

--verifies the precondition of the operator is true

pre: (ops.name).precondition

post: (ops.name).postcondition

Controller::msgCheckRules(msg: message)

pre: rules->exists(x:Rule | x.precondition=true)

post: RuleContainer.executeRule(rules->select(x:Rule | x.precondition=true))

RuleContainer::executeRule(rule: Rule)

pre: --none

post: rule.postcondition

VITA

First Lieutenant David J. Robinson was born on 30 October 1969 in Lancaster, New Hampshire. He graduated from White Mountain Regional High School in Whitefield, New Hampshire in June 1988. He served in the Air Force's enlisted force from October 1988 to August 1992. He completed his undergraduate studies at the University of Connecticut with a Bachelor of Science degree in Computer Science and Engineering. He was commissioned through the Detachment 115 Air Force Reserve Officer Training Corps (AFROTC) in August 1996.

Lieutenant Robinson's first assignment was at Cannon AFB, New Mexico as a Communications and Computer Systems Operator in January 1989. As an officer, his first assignment was at HQ AIA, Kelly AFB where he served as a C4I Systems Engineer in September 1996. In July 1998, he entered the Graduate School of Engineering's Computer Systems Engineering program, Air Force Institute of Technology. Upon graduation, he will be assigned to USSTRATCOM/J2 at Offutt AFB where he will be working in the Enterprise Architecture Services Branch.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A COMPONENT BASED APPROACH TO AGENT SPECIFICATION			5. FUNDING NUMBERS	
6. AUTHOR(S) David J. Robinson, First Lieutenant, USAF				
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/00M-22	
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			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES Maj Scott A. Deloach, ENG, DSN: 785-3636, ext 4622				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE	
ABSTRACT (Maximum 200 Words) With the size and complexity of software systems increasing, the overall design, specification, and verification of the structure of systems turns into a crucial concern for software engineers. The Air Force, as well as all of industry, is currently faced with the problem of having to produce larger and more complex software systems that run efficiently and reliably as well as being extensible and maintainable. The lack of a well-established notation upon which software engineers may agree has made solving this problem even more difficult. The goal of this research was to develop a knowledge representation language that can be used to unambiguously specify and design software systems in a verifiable, efficient, and understandable manner. To ensure maximum understandability and ease of use, the language was to make use of both graphics and text to represent information. The end result of this research was a component-based language that allows for the specification of software systems in a formal yet understandable manner. The language was implemented in the agentTool software development environment.				
14. SUBJECT TERMS Agents, agentTool, Components, Connectors, Language, UML, OCL, Multi-agent Environment, Formal, Specification, Representation, Architecture, Architectural Style			15. NUMBER OF PAGES 165	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102