DESIGN AND IMPLEMENTATION OF AUTONOMOUS AGENT ARCHITECTURE FOR SAGE

by

Amina Tariq
Amna Basharat

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelors of Computer Software Engineering

National University of Sciences and Technology,
Rawalpindi

May 2005

DEDICATION

In the name of Allah, the Most Beneficent, the Most Merciful

To our dear Families…especially to our Mothers

ACKNOWLEDGMENTS

After the grace of Almighty Allah and the prayers and support of our parents and family, we are deeply beholden to our supervisor Associate Professor Dr. H Farooq Ahmad for his continuous assistance, inspiration, patience and unconditional support. We are highly gratified to our Co-Supervisor, Professor Dr. Arshad Ali, Director General NIIT, for his continuous and valuable suggestions, guidance, and commitment towards provision of undue support throughout our thesis work. His ability of management and foresightedness helped us take our research beyond levels of excellence.

We would also like to show immense gratitude to Dr. Hiroki Suguri, Comtec, Japan for his valuable suggestions and guidance and for all his support during our Collaborative Research Visit to Japan as part of this project. We would also like to extend our gratitude to Prof. Kinji Mori, Tokyo Institute of Technology for extending his support, appreciation and encouragement for this research.

We wish to acknowledge the continuous support of the Commandant MCS, Major General Hamid Mehmood, Chief Instructor MCS, Brig. Dr. Muhammad Akbar, HOD CS Department Col. Raja Iqbal, Col. Naveed Khattak and the faculty and administration of MCS.

We are highly thankful to all of our professors whom had been guiding and supporting us through out our course and research work. Their knowledge, guidance and training enabled us to carry out this research work.

We are especially thankful to Mr Omair Shafiq and Mr Nauman Qureshi for working sincerely with us in development of the application and Mr Zaheer Abbas Khan for his valuable suggestions. We would like to offer our thanks to all of our colleagues in Comtec, Mr Ishikawa and Mr Lee who had valuable contribution over the course of our research.

We would like to offer our admiration to all our classmates, and our seniors who had been supporting, helping and encouraging us throughout our thesis project. We are also indebted to the MCS system administration for their help and support.

We would like to offer appreciation to our parents for their vision and commitment to make us learn, and other family members for their encouragement, support and prayers.

ABSTRACT

Multi-agent systems (MAS) advocate an agent-based approach to software engineering based on decomposing problems in terms of decentralized, autonomous agents that can engage in flexible, high-level interactions. Scalable, fault tolerant Agent Grooming Environment (SAGE), a second generation MAS offers a decentralized, fault tolerant agent framework unlike the first generation MASs. Currently the system framework of SAGE is in place, and a well defined agent architecture needs to be developed which will allow development of autonomous and intelligent agents on top of this decentralized system. The aim of the project was to design and develop a hybrid agent architecture in such a holistic and well defined manner that has not figured in the domain of FIPA-Compliant architectures so far. The existing architectures reside on top of centralized frameworks and primarily offer reactive behaviour support. The novelty of this Hybrid approach is that it allows agents to practically reason about their behaviour in addition to reactive and deliberative behaviour. This has not been previously done in the first generation MASs e.g. JADE, FIPA-OS. The highlight of the architecture is that it gives a generic design of sophisticated agent models such as BDI and reactive architectures with in well defined modules. Also presented is the design and implementation of a high level agent application which depicts the synergy of software agents, Grid Computing and the web services (Agent Web Gateway). The application not only validates the architecture but also illustrates the coordination of different entities by agents acting as owners in heterogeneous and dynamically changing environments.

TABLE OF CONTENTS

LIST OF FIGURES

Number Page

LIST OF ABBREVIATIONS

JADE	Java Agent Development Framework
FIPA	Foundation for Intelligent Physical Agents
UDDI	Universal Description and Discovery Integration
SOAP	Simple Object Access Protocol
CORBA	Common Object Resource Broker Architecture
Java RMI	Java Remote Method Invocation
IDL	Interface Definition Language
NASSL	Network Accessible Service Specification Language
WDS	Well-Defined Service
HTTP	Hyper Text Transfer Protocol
HTTPS	Secure HTTP
LDAP	Lightweight Directory Access Protocol
XSD	XML Schema Definition
ACL	Agent Communication Language
MAS	Multi-Agent System
KQML	Knowledge Query Manipulation Language
SL	Semantic Language
B2B	Business to Business
WWW	World Wide Web
IT	Information Technology
PCs	Personal Computers
RDF	Resource Description Framework
P2P	Peer to Peer
QoS	Quality of Service
TCP/IP	Transmission Control Protocol/ Internet Protocol
SAX	Simple API for XML
DOM	Document Object Model
JAXP	Java API for XML Processing
SMTP	Simple Mail Transfer Protocol

FTP	File Transfer Protocol
SAAJ	SOAP with attachment API for JAVA
API	Application Programming Interface
MIME	Multi-purpose Internet Mail Extensions
OS	Operating System
GUI	Graphical User Interface
IIOP	Internet Inter-ORB Protocol
ORB	Object Resource Broker
MTS	Message Transport Service
DF	Directory Facilitator
AMS	Agent Management System
SSL	Secure Socket Layer
XSL	XML Stylesheet Language
WSDL	Web service Description Language
GASS	Globus Access to Secondary Storage
GSDL	Grid service Description Language

Chapter 1

INTRODUCTION

1 INTRODUCTION

Multi-agent systems (MAS) are based on the idea that a cooperative working environment comprising synergistic software components can cope with problems which are hard to solve using the traditional centralized approach to computation [1]. Smaller software entities – software agents – with special capabilities (autonomous, reactive, pro-active and social) are used instead to interact in a flexible and dynamic way to solve problems more efficiently. Multi agent system is a distributed paradigm that contains a community of social agents, which can act on behalf of their owners. It is increasingly becoming a ubiquitous paradigm for the design and implementation of complex software applications as it can support distributed collaborative problem solving by agent collections that dynamically organize themselves. Multi-agent systems are based on the idea that a cooperative working environment comprising synergistic software components can cope with problems which are hard to solve using the traditional centralized approach to computation. The improvements of the use of multi-agents technology in automation and manufacturing systems are the fast adaptation to system reconfiguration (for example addition or removal of resources, different organizational structures, etc.), re-use of code for other control applications, increase of flexibility and adaptation of the control application and more optimized and modular software development

Intelligent agents have become the most vibrant and fastest growing research area in both artificial intelligence and computer science. New agent-based products, applications and services arise on an almost daily basis. Since the term "agents" has been very popular and used with a number of different definitions, this report reviews a broad range of toolkits and products, and provides an overview of the most important issues of agent-based technology.

The agent-based approach to software systems development views these autonomous software agents as components of a much larger business function. The main benefit of viewing them from this perspective is that the software components can be integrated into a coherent and consistent software system in which they work together to better meet the needs of the entire application (utilizing autonomy, responsiveness, pro-activeness and social ability). Typical agent applications are e-commerce, network management, information retrieval for further processing, digital tourism, supply chain management, support systems, web services, medical and Grid etc.

1.1 BACKGROUND

Multi Agent System (MAS) is a distributed paradigm that contains a community of social agents which can act on behalf of their owners. Foundation for Intelligent Physical Agents (FIPA) is one of the standard governing bodies, which provide an abstract architecture, which act as the guidelines to be followed by the multi-agent system developers [2]. FIPA has standardized autonomous multi-agent system abstract architecture, based on components like AMS, DF, ACL, MTS etc, which themselves are the agents in their own right.

First generation FIPA compliant multi-agent systems lack techniques to manage challenges faced by distributed systems. The MAS is moving into a new era, what is popularly known as the second generation, where the problems faced in the era of the first-generation are being tackled with and some of them are even getting resolved. Scalable and fault tolerant Agent Grooming Environment (SAGE), a second generation FIPA-compliant MAS is being developed at the NUST-COMTEC lab. The SAGE framework is currently providing an environment for creating distributed and intelligent and autonomous entities that are encapsulated as agents [3, 4]. The SAGE architecture provides tools for runtime agent management, directory facilitation monitoring and editing, message exchange debugging and agent life cycle control. However, the SAGE currently does not provide any built in mechanism to program the agent behavior and their capabilities.

The aim is to design an Agent Construction Model that allows description and development of a range of Agent types and agent-based applications. The challenge is to provide an agent API that is equipped with fundamental capabilities that an autonomous agent must possess to participate in the default society chosen by the MAS developers.

1.2 PROBLEM STATEMENT

Scalable and fault tolerant Agent Grooming Environment (SAGE), a second generation FIPA-compliant MAS is being developed at the NUST-COMTEC lab. For this MAS agent architecture is required which should allow the incorporation of agent properties like communication and coordination, planning, decision making, belief update and their integration.

The aim of this project is to design and develop a Generic Agent Construction Model that allows description and development of a range of Agent types possessing various behaviors, and roles.

1.3 SCOPE

There are two ways in which the scope of our project can be defined first in the Distributed Computing Paradigm and secondly with in the Agent oriented paradigm.

For defining the scope in the distributed computing domain of our project it is necessary to look at the main sub categories of the Distributed Computing, which include Grid Computing, Agent based technologies and Web Services. Figure 1.1 clearly depicts the place of our project in the distributed paradigm.

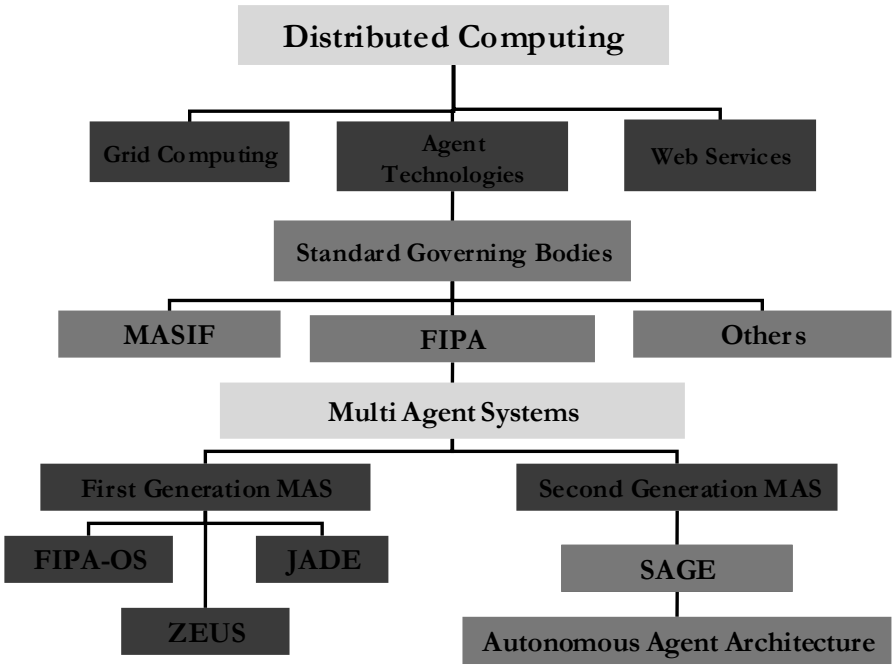

Figure 1.1: Scope in Distributed Computing

In the category of Agent based technologies the work is mainly related to those Multi-Agent Systems which are FIPA compliant. FIPA has specified the basic architecture of a Multi Agent Systems. Based on FIPA specifications First generation FIPA compliant Multi Agent Systems were developed, which had centralized system architecture. SAGE is the first Second Generation

Multi-Agent System which has a decentralized agent building framework. The scope of our project is basically to design and develop an autonomous agent architecture for the System Framework of SAGE.

Visualizing the scope of our project in the Agent oriented Paradigm; it can be seen that agent environment is a result of an approach that encompasses various phases in a layered manner. The current position of SAGE in the agent oriented paradigm abstraction layers is that the underlying language, application Framework and the Agent Infrastructure is in place as shown in Figure 1.2. Now on top of that frame work aim is at defining the agent role by developing autonomous agent architecture. The combination of the architecture and the framework would allow us to develop various agents, which will interact together to merge into agent society. This will lead certainly to the development of complete agent environment. Therefore the scope of our project lies in taking SAGE one step higher in the agent oriented paradigm.

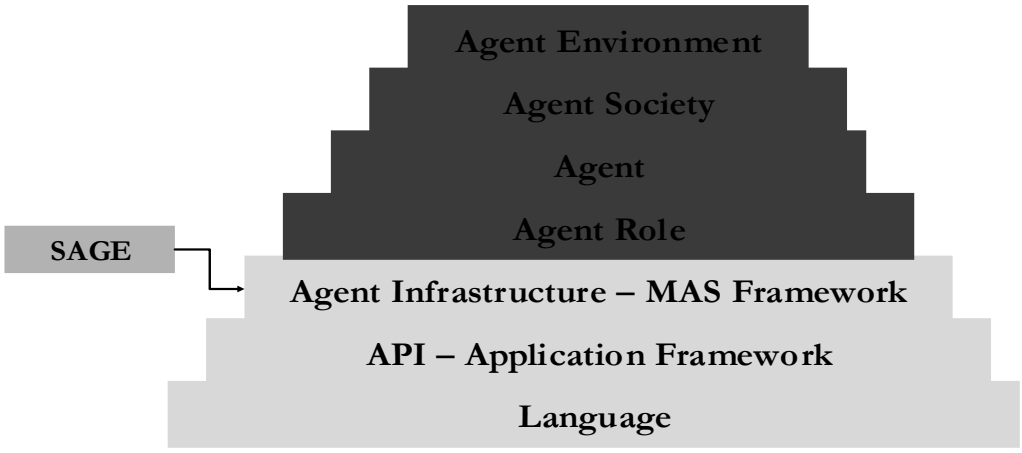

Figure 1.2: Scope in Agent Oriented Paradigm

1.4 PROJECT VISION

The SAGE supports a modular and extensible approach to design complex information systems, which require services of multiple autonomous agents having diverse capabilities and needs. These capabilities of agents are to be defined in agent's internal architecture.

The SAGE framework is currently providing an environment for creating distributed and intelligent and autonomous entities that are encapsulated as agents. The SAGE architecture provides tools for runtime agent management, directory facilitation monitoring and editing,

message exchange debugging and agent life cycle control as shown in Figure 1.3. However, the SAGE currently does not provide any built in mechanism to program the agent behavior and their capabilities. Our project aims at providing a strong middleware support to program agent behavior.

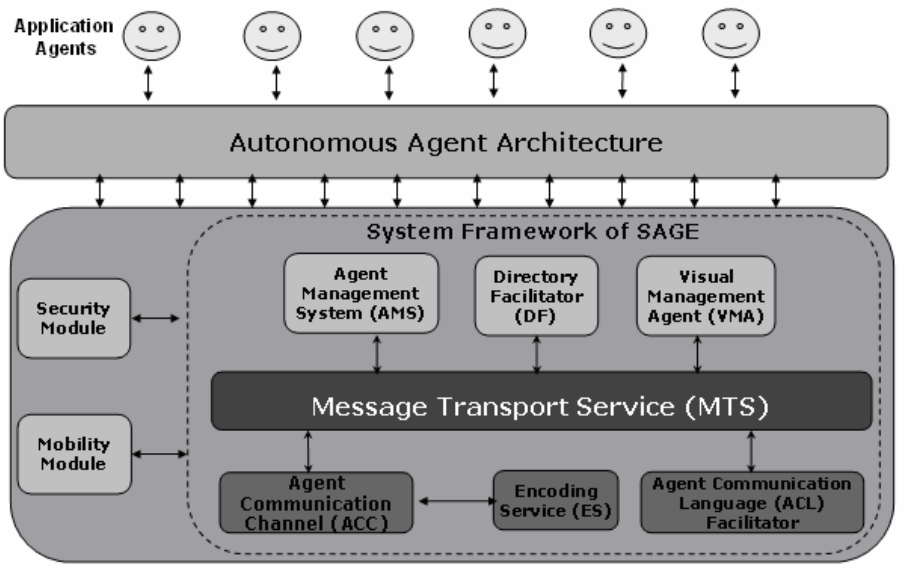

Figure 1.3: Project Vision

The aims and objectives that are associated with the achievement of the project:: (i) To provide an agent architecture which is a specification that describes how an agent derives rational actions to respond the events perceived from the environment to meet the goal., (ii) To design an Agent Construction Model that allows description and development of a range of Agent types and agent-based applications, (iii) To provide an agent API that is equipped with fundamental capabilities that an autonomous agent must possess to participate in the default society chosen by the MAS developers, and (iv) To provide actions or utilities that are commonly needed by any agent, interacting with in a collaborative environment for achieving mutual cooperation with other agent.

Chapter 2

LITERATURE OVERVIEW

2 LITERATURE OVERVIEW

2.1 FIPA

The Foundation for Intelligent Physical Agents (FIPA) is a nonprofit organization aimed at producing standards for the interoperation of heterogeneous software agents [2]. The core mission of the FIPA software agent standards consortium is to facilitate the interoperation and inter-working between agents across multiple, heterogeneous agent systems. To this purpose, FIPA has been working on specifications that range from agent platform architectures to support communicating agents, semantic communication languages and content languages for expressing messages and interaction protocols that expand the scope from single messages to complete transactions. The core message of FIPA is that through a combination of speech acts, predicate logic and public ontologies, standard ways of interpreting communication between agents can be offered that respect the intended meaning of the communication.

The Foundation of Intelligent Physical Agents (FIPA) is an international agent standardization body, which aims mainly for the establishment of specifications enabling the interoperability of agent systems. Therefore the prime focus is on the definition of a generic Agent Communication Language (ACL), enabling interactions between different vendors intelligent agent systems.

2.2 PROBLEMS OF THE FIRST GENERATION MAS

The first-generation FIPA-compliant MAS include the JADE [5], the FIPA-OS [6], the Zeus[7]. These Multi-agent systems research community has found it very difficult to convince the world to use agents and MASs for their daily tasks because: (i)Existing MAS lack fault tolerance, (ii) FIPA has recently standardized security issues of MAS and existing architecture do not cater security issues, (iii) Existing MAS are not light weight. JADE with the integration of LEAP, becomes light weight but it provides limited and restricted services and both the components become dependent on each other consequently. This way fault management also becomes difficult, (iv) Existing MAS are not scalable, and (v) Existing MAS are less dynamic for high performance. Dynamic invocation of services i.e. services on demand activation will cause MAS to provide high performance.

2.3 CLASSIFICATION OF AGENTS AND AGENT ARCHITECTURES

Agents may be classified along several ideal and primary attributes which agents *should* exhibit [8]. Based on Our Analysis and literature a minimal list of three has been identified:

2.3.1 Autonomy

This property enables the agents to operate without the direct intervention of human or others, and have some kind of control over their actions and internal state.

2.3.2 Learning

One of the distinguishing properties of agent is Self-Learning which actually allows the agent to make decisions on the basis of past experiences.

2.3.3 Social Ability

Agents are able to interact with other agent (and possible humans). This interaction is possible when the agent posses the property of social ability.

These are the three main properties, which are possessed by the agents of various kinds. An intelligent software agent is not required to posses them all but at least two. The classification of agents is done on the basis of which of these properties an agent exhibits. For defining a suitable type of agent for SAGE we carried out a detailed study of various agent types in the light of these properties.

2.4 TYPES OF AGENTS

Various types of agents that have been identified in the light of the properties described previously as shown in Figure 2.1.

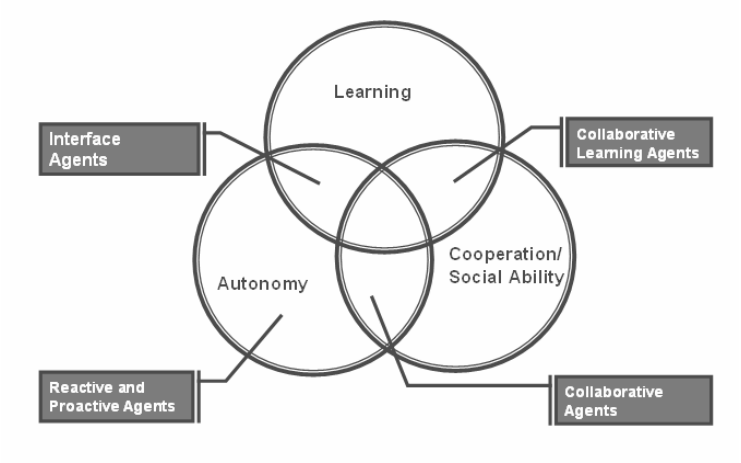

Figure 2.1: Agent Classification

Collaborative agents is that type of agents which emphasises autonomy and cooperation (with other agents) in order to perform tasks for their owners. But usually these types of agents can not perform Complex learning. Interface agents fall in that category of agents that emphasise autonomy and learning in order to perform tasks for their owners. Usually these types of agents perform their tasks without interacting with other agents so they do not posses the capability of social ability.A reactive system is one that maintains an ongoing interaction with its environment, and responds to changes that occur in it (in time for the response to be useful).Reactive agents act/respond in a stimulus-response manner to the present state of the environment in which they are embedded. Deliberative agents contain an explicitly represented, symbolic model of the world. These agents have the capability to make decisions (for example about what actions to perform) via symbolic reasoning. These agents are autonomous as well as posses the capability of self learning. These agents build internal models of the world and then use them to formulate plans in order to achieve goals.Hybrid agents refer to those whose constitution is a combination of two or more agent *philosophies* within a singular agent. These philosophies include a Reactive Agent philosophy, an interface agent philosophy, collaborative agent philosophy.

2.5 CLASSIFICATION OF AGENT ARCHITECTURES

Agent architectures are the fundamental engines underlying the autonomous components that support effective behavior in real-world, dynamic and open environments. They specify how the agents can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide

an answer to the question of how the sensor data and the current internal state of the agent determine the actions and future internal state of the agent. Architecture encompasses techniques and algorithms that support this methodology [9].

2.5.1 Logic Based Agent Architecture

In logic based agent architectures decision making is realized through logical deduction. In this case agents are viewed as particular type of knowledge based system. As the name symbolizes they contain an explicitly represented symbolic model of the world. The decisions are taken via symbolic reasoning [10].

2.5.2 Reactive Agent Architecture

The reactive architecture does not rely on symbol manipulation. In these types of architectures intelligent behavior can be generated without explicit representations proposed by symbolic AI. Intelligent behavior can be generated without explicit abstract reasoning. These architectures work on the principle that Intelligence is an emergent property of certain complex systems.

2.5.3 BDI Agent Architecture

BDI is one of the most prospective Architecture for Practical Reasoning. The agents having the BDI architectures (as shown in Figure 2.2) have roots in understanding practical reasoning. BDI agents usually perform two main processes: (i) Deliberation: deciding what goals we want to achieve and (ii) Means-ends reasoning: deciding how we are going to achieve these goals [11].

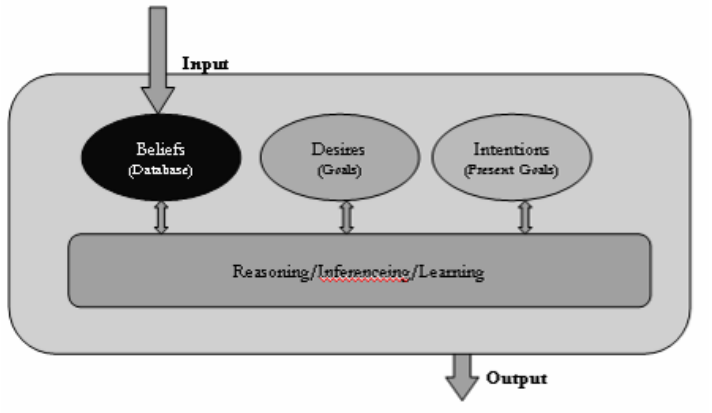

Figure 2.2: BDI Model for Agents

2.5.4 Layered (Hybrid) Agent Architecture

This type of architecture_decomposes the system into different layers to deal with different types of behaviors. Typically at least two layers – to deal with reactive and proactive behaviors. Broadly, two types of control flow within layered architectures: (i) Horizontal layering and (ii) Vertical layering. Figure 2.3 depicts these forms of layered architectures.

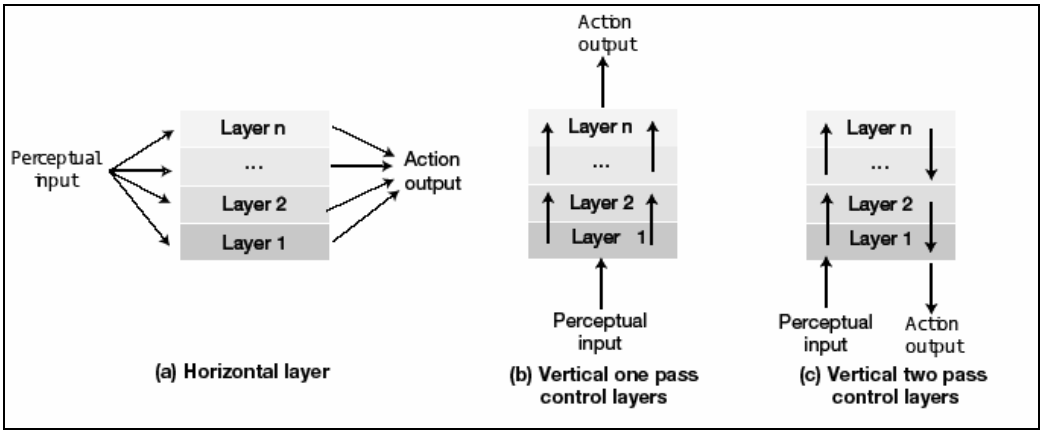

Figure 2.3: Layered Agent Architectures

2.6 AGENT ARCHITECTURES OF FIRST GENERATION FIPA-COMPLIANT MAS

MAS frameworks attempt to provide programmer with reusable agent-oriented classes which share useful relationships. FIPAOS, JADE, and Zeus are all open source Java based First generation MAS frameworks implementing to varying degrees. The features these frameworks provide are similar as they share a common goal of providing a code base for developing intelligent, distributed, and autonomous software using agents as the unit of encapsulation. What varies is their high level architecture. FIPAOS, JADE, and Zeus all have a core behavior subsystem that includes an execution process, ACL message interface agent behavior engine, and corresponding primitive processing objects [12].

2.6.1 Agent Architecture of FIPA-OS

Nortel Networks developed the FIPAOS framework with the intent of providing a platform whose architecture emphasized ease of extension, modularity, and therefore openness. FIPAOS, in accord with numerous FIPA specifications, provides support for agent management, ACL

message transmission and reception, and protocol adherence [6]. FIPAOS defines its agent architecture in form of an agent shell that serves as the foundation for building customized agents; it includes a task manager for constructing agents from primitive work units, a conversation manager that ensures protocol compliance while also providing conversation utilities.

2.6.2 Agent Architecture of JADE

Just as FIPAOS JADE MAS framework has implemented all of the mandatory components of the FIPA specifications [5]. Developed at CSLET, the primary objective of this framework is to make it easier to program multi-agent societies whose agents interact in compliance with FIPA. Starting from the FIPA assumption that only the external behavior of system components should be specified, while leaving the implementation details and internal architectures to agent developers, a very general agent model has been implemented by JADE, which is very primitive in nature. JADE uses the Behavior abstraction to model the tasks that an agent is able to perform and agents instantiate their behaviors according to the needs and capabilities.

2.6.3 Agent Architecture of ZEUS

Zeus was developed at the British Telecom Labs [7]. The MAS framework provides a comprehensive suite of monitoring tools at the agent and society level. The agent architecture of Zeus adopts layered approach to agents. They have the ability to plan sequences of steps needed to accomplish goals. The Zeus state machine support and monitoring tools are also well developed.

Chapter 3

REQUIREMENT ANALYSIS

3 REQUIREMENT ANALYSIS

For requirement analysis of Sage Agent Architecture it is required to first understand the architecture of Sage.

3.1 SAGE- THE SECOND GENERATION MAS

The MAS is moving into a new era, what is popularly known as the second generation, where the problems faced in the era of the first-generation are being tackled with and some of them are even getting resolved. Currently, developers are emphasizing to develop high performance multi-agent systems that are fault tolerant, scalable, secure, autonomous, intelligent, dynamic and light-weight. Scalable and fault tolerant Agent Grooming Environment (SAGE), a second generation FIPA-compliant MAS is being developed at the NUST-COMTEC lab. The SAGE implements all the basic FIPA specifications that provide the normative framework within which FIPA agents can exist, operate, and communicate. JAVA is being used as the programming language.

SAGE, a second generation FIPA-compliant MAS is being developed at the NUST-Comtec labs. SAGE implements all those basic FIPA specifications that provide the normative framework within which FIPA agents can exist, operate, and communicate. Currently, The SAGE architecture provides tools for decentralized runtime agent management, directory facilitation monitoring and editing, message exchange debugging and agent life cycle control. It overcomes the problems inherent in First Generation MAS by providing support for a decentralized architecture.

3.2 MAIN COMPONENTS OF SAGE

SAGE implements all the modules specified in FIPA abstract architecture. SAGE provides the physical infrastructure in which agents can be deployed. The Agent Platform (AP) consists of the machine(s), operating system, agent support software, FIPA agent management components (DF, AMS and MTS) and agents as shown in Figure 3.1.

An **Agent Management System (AMS)** is a mandatory component of the AP. The AMS exerts supervisory control over access to and use of the AP. Only one AMS will exist in a single AP. The AMS maintains a directory of AIDs which contain transport addresses (amongst other things) for agents registered with the AP. The AMS offers white pages services to other agents.

Each agent must register with an AMS in order to get a valid AID. The AMS is a reification of the Agent Directory Service.

A **Directory Facilitator (DF)** is an optional component of the AP, but if it is present, it must be implemented as a DF service. The DF provides yellow pages services to other agents. Agents may register their services with the DF or query the DF to find out what services are offered by other agents. Multiple DFs may exist within an AP and may be federated.

Where as **Message Transport Service (MTS)** is the default communication method between agents on different APs. The main highlight of SAGE is its decentralized Architecture.

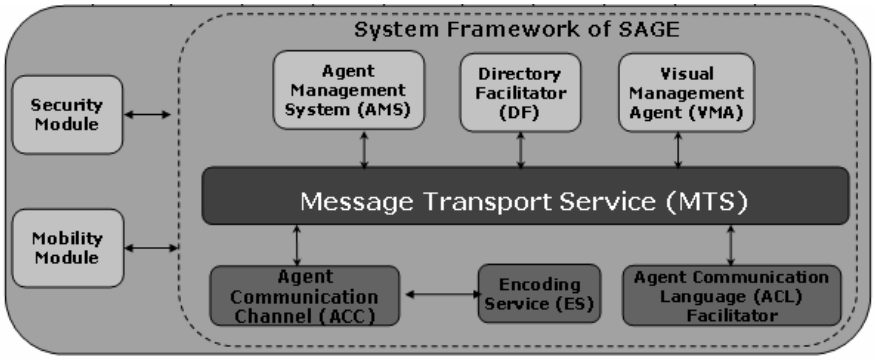

Figure 3.1: System Framework of SAGE

3.3 SYSTEM LEVEL AUTONOMIC CHARACTERISTICS OF SAGE

In Multi Agent Systems each component exhibits its own autonomic behavior. Particularly at the level of individual components of an MAS, well-established techniques, many of which fall under the rubric of fault tolerance, have led to the development of elements that rarely fail, which is one important aspect of being autonomic. Motivated by this very concept SAGE has been developed as a, fault-tolerant, decentralized MAS framework.

SAGE achieves the aim of a fault tolerant Agent Platform by offering a decentralized architecture based on the notion of Virtual Agent Cluster, which provides fault tolerance capability by using separate communication layers among different machines. The Virtual Agent Cluster works autonomously, regardless of the external environment events, providing a self healing, proactive abstraction on top of all instances of MASs. Also the architecture ensures high assurance

using peer to peer architecture which brings scalability, fault tolerance and load balancing among distributed peers.

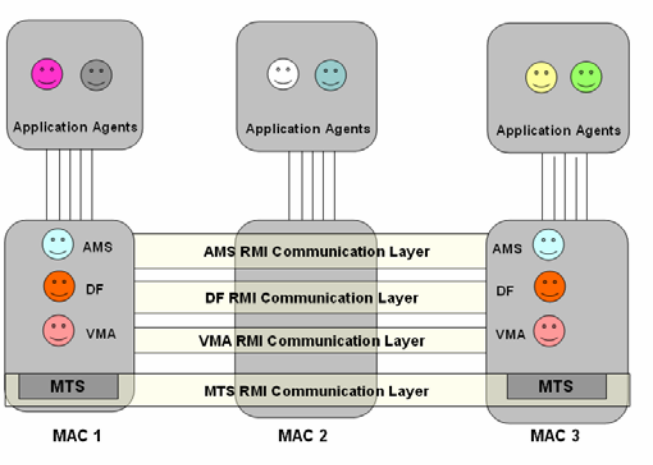

Figure 3.2: Decentralized, Fault-Tolerant Architecture of SAGE

The decentralized architecture of SAGE as shown in Figure 3.2, also embeds the capability of self-monitoring at the system level by allowing the agents to internally monitor themselves as well as externally monitor other agents. The external monitoring capability has been incorporated in SAGE by allowing all the instances within the Virtual Agent Cluster send heart beats (Hello messages) to each other to check the liveliness of peer instances of MAS.

Also one of the features that is highlight of SAGE's autonomic system is the capacity of the agents to be self-descriptive as each sage-agent keeps its own descriptive information as attributes, and makes it available to other through sage's system agents e.g. the DF. The system framework then makes the agents dynamically discover and interact with each other. Since part of FIPA-compliant MAS, SAGE agents communicate with each other using ACL Messages. An important principle of the system is that no other means of communication between the elements is permitted. This allows us to completely specify the interactions between SAGE agents in terms of the interfaces that they support, and the behaviors that they exhibit through these interfaces. The self-management of the system will arise not only from the myriad/numerous interactions among sage-agents but also from the internal self-management of the individual sage-agents—just as the social intelligence of an ant colony arises largely from the interactions among individual ants . The distributed, fault-tolerant service-oriented infrastructure of SAGE's system Framework is supportive of these agents and their interactions.

3.4 REQUIREMENT ANALYSIS FOR SAGE'S AGENT

For Agents in SAGE it was analyzed that the requirement is that agents should be reactive as well as they should be capable of reasoning about their behaviour i.e. Goal-directedness.

It is needed for the Sage Agents to be reactive, responding to changing conditions in an appropriate (timely) fashion. Also, it is needed for the agents to systematically work towards long-term goals. Thus an Overall Autonomous Behavior is needed for SAGE.

Further requirement analysis revealed that agents in SAGE ought to be social entities that should solve their problems by negotiating and cooperating with other agents. Thus an enhanced nature of intelligent social collaboration and cooperation is needed for SAGE.

In addition to the requirements defined above efficiency is also one of the prime requirements which are to be maintained. Agents should be bale to carry out multiple conversations concurrently.

Thus the Analysis of Agent Requirements for SAGE allows for the conclusion that each agent in SAGE should be able to support a combination of Collaborative, Reactive, Proactive and Self-Learning Agent Philosophies. Thus each agent in SAGE will necessarily be a HYBRID Agent

3.5 REQUIREMENT ANALYSIS FOR SAGE'S AGENT ARCHITECTURE

The Requirements for the Agent Architecture of SAGE has been categorized as functional requirements which are identified in the light of characteristics required for SAGE agents and Non-functional requirements which relate to the system level and efficient running of the over all system.

3.5.1 Functional Requirements

The functional requirements have been identified to include:

3.5.1.1 Minimal Behavior Support

To embed autonomy in each SAGE agent it was required that lower-Level Agent Actions for reactive agent behavior should be defined in the agent architecture. Also to facilitate the proactive behavior modeling support for state based modeling of the agent behavior was required.

3.5.1.2 Intelligent Communications

As the requirement is to equip each agent with the capability of social ability, for this reason a Negotiating Framework for the agent that facilitates intelligent communication and cooperation is required at the agent architecture level, providing SAGE the support for Intelligent High Level Inter-Agent communications.

3.5.1.3 Knowledge Representation and Reasoning

To facilitate Learning, Decision Making and Generation of Goals by the Agent through analysis of its own motivations and beliefs, a well defined knowledge representation an interpretation mechanism is required at the architectural level.

3.5.2 Non-Functional Requirements

The non-functional requirements have been identified primarily to include Efficiency**.** Efficient internal agent processing is one of the prime requirement to be considered while developing the agent architecture. As agents are social entities they require Intra-Agent Concurrency where an agent must be able to carry out several concurrent tasks in response to different external events. Requirement is that this level of concurrency should not be a burden on the system. Every module should be as light weight as possible.

Chapter 4

CONCEPTUAL AGENT MODEL FOR SAGE

4 CONCEPTUALIZATION OF AGENT ARCHITECTURE FOR SAGE

In the light of the above requirement analysis a hybrid, layered model for the agent architecture was proposed that encapsulates the autonomous properties of the agent in well defined modules. First a conceptual model was developed for the agent architecture inspired by the Bell's model of rational agent based upon which the design of the actual agent architecture for SAGE was given.

4.1 CONCEPTUAL MODEL FOR SAGE'S AGENT ARCHITECTURE

In this section, a reference architecture and methodology for building an autonomic agent capable of playing a role in a future autonomic computing infrastructure is outlined.

For the agents in SAGE the strong notion of agency suggested in [8, 10, 13] was adapted. It was analyzed that the agents must possess reactivity, proactivity, social ability, self-learning and adaptability. Thus a complete notion of a hybrid agent was conceived. To incorporate these skills in SAGE agents such an agent architecture was required that should be enable the agents to carry out their tasks in a concurrent manner efficiently. To enable the agent reactivity it was realized that certain minimal behavioral support is also required at the architecture level. Since agents are highly social entities, the support for high level agent conversations was also taken as a must to be provided by the Agent Architecture. Furthermore, in order to endow the SAGE Agents with the capability of reasoning and adaptation, it was conceived that a reasoning engine should be provided as an ingredient of the Agent Architecture. As a consequence, the Agent Architecture for SAGE was designed be hybrid architecture – mix of deliberative and reactive architectures. The high-level functional architecture on which the actual agent architecture for SAGE is based on is pictured in Figure 4.1, inspired by Bell's model of Rational Agent [14].

Each SAGE agent is conceptualized to be responsible for managing its own internal state and behavior and for managing its interactions with an environment that consists largely of signals and messages from other agents (FIPA-ACL messages) and the external world in the form of the multi-agent environment.

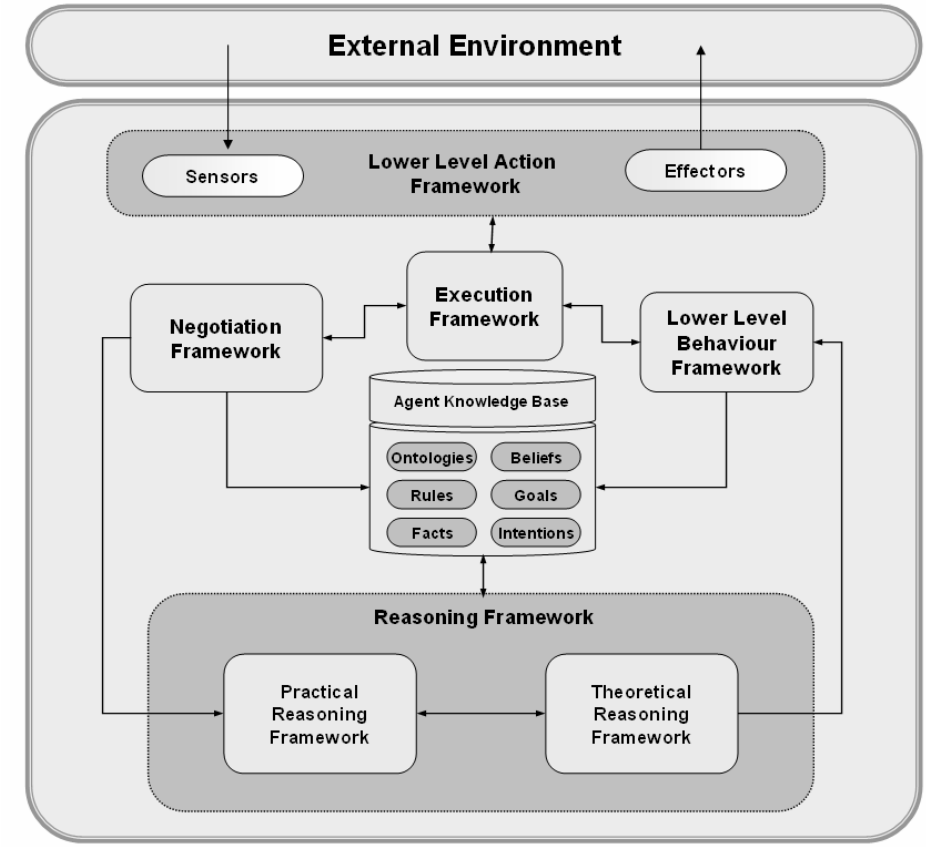

Figure 4.1: Conceptual Architecture for Autonomous Agents in SAGE

We have proposed the conceptual autonomous agent architecture shown in Figure 4.1 that features a lower level action framework consisting of sensors and effectors for interacting with the external environment. This lower level action framework works on top of a number of reactive, adaptive, and reasoning frameworks that dynamically model the SAGE agent itself and its environment.

In order for our intelligent autonomic agent to build and maintain a model of the external environment and of its own components, we conceived The Reasoning Framework which embeds the capability of self-adaptation with in the SAgent.

This Framework allows the agent to reason about its action by embedding the capability of reasoning on the basis of the knowledge agent possess i.e. the reasoning of the agent that is directed towards the agent beliefs. In addition this framework allows the agent to make decisions about the, what goals agent wants to achieve utilizing the capability of the practical reasoning framework, which also defines the course of actions to achieve the goals

Hence we can say SAgent's internal behavior and its relationships with other agents are driven by goals that are embedded in it, by other agents that have authority over it, or by subcontracts to peer agents with its tacit or explicit consent.

Each SAGE agent may require assistance from other agents to achieve its goals. If so, it will be responsible for obtaining necessary resources from other agents and for dealing with exception cases, such as the failure of a required resource. However, once an agent finds potential providers of an input service, it must negotiate with them to obtain that service. Thus SAgents need flexible ways to express multi-attribute needs and capabilities, and they need mechanisms for deriving these expressions from human input or from computation. They also need effective negotiation strategies and protocols that establish the rules of negotiation and govern the flow of messages among the negotiators. For embedding the capability of high level negotiations with in each SAGE agent we have proposed a Negotiation Framework which will provide built in support for high level agent conversations and will support the myriad interactions amongst the system entities.

In addition to the assistance required from other agents to achieve its goals, an agent may have to take certain lower level actions which will be taken in accordance with the plan generated by the Practical Reasoning Framework. We have conceptualized a Lower Level Behavior Framework that will be responsible for providing set of actions needed commonly by every agent in an agent society. This frame work also provides support for State-Based modeling of agent behavior to define the complex agent behavior. This stems from the fact described earlier that aim of agent architecture is to facilitate the programmer in defining the role of the Autonomic Agent.

Like other autonomic elements, SAgents are complex entities that may be carrying out several negotiations and tasks simultaneously. In addition they have to continually sense and respond to the environment in which they are situated. For controlling the execution of the multiple agent tasks concurrently in SAGE we have conceptualized an Execution Framework that is responsible for controlling the concurrent execution of various agent tasks.

4.2 REMINISCENCE WITH MAPE MODEL

The model we have developed is also reminiscent of the MAPE (Monitor, Analyze, Plan, and Execute) model [15, 16], as per the vision of autonomic computing [17, 18, 19] which breaks management architecture down into four common functions: (i) Collecting data (ii) Analyzing data,

(iii) Creating a plan of action, and (iv) Executing the plan. By Monitoring (M) behavior– through the Negotiation and the lower level behaviour framework, Analyzing (A) data and Planning (P) the actions that should be taken- through the Practical Reasoning Framework and Executing (E) them through the execution and the lower level action framework a kind of a control loop is created. This completes the picture of the autonomic entities in our system as per the vision of the autonomic computing.

The MAPE model assumes the existence of a common knowledge element that represents the knowledge about a problem space that is shared among the four components of the MAPE model. This shared knowledge is provided in our model in the form of Agent's Knowledge Base. It includes such aspects as information about what beliefs or goals the agent possesses what plans they execute, and what rules specify their behavior.

By constructing the base agent shell using this architecture, intelligent autonomic systems of widely varying complexity could be built. The system retains all of the advantages of the reactive behavior-based architecture while adding internal mental states, including models of the self and world, emotions, learned behaviors, planning, and meta-level decision-making. In the next section we present the design and component level detail of the actual model that has been designed for SAGE.

Chapter 5

AGENT ARCHITECTURE FOR SAGE

5 PROPOSED MODEL OF AGENT ARCHITECTURE FOR SAGE

5.1 PROPOSED MODEL FOR SAGE'S AGENT ARCHITECTURE

The Figure 5.1 shows the design model for the autonomic agent architecture of SAGE referenced on the Conceptual architecture outlined in the previous section. The model consists of three core sub-systems: (i) The Action Sub-Engine, (ii) The Behaviour Sub-Engine and (iii) The Reasoning Sub-Engine

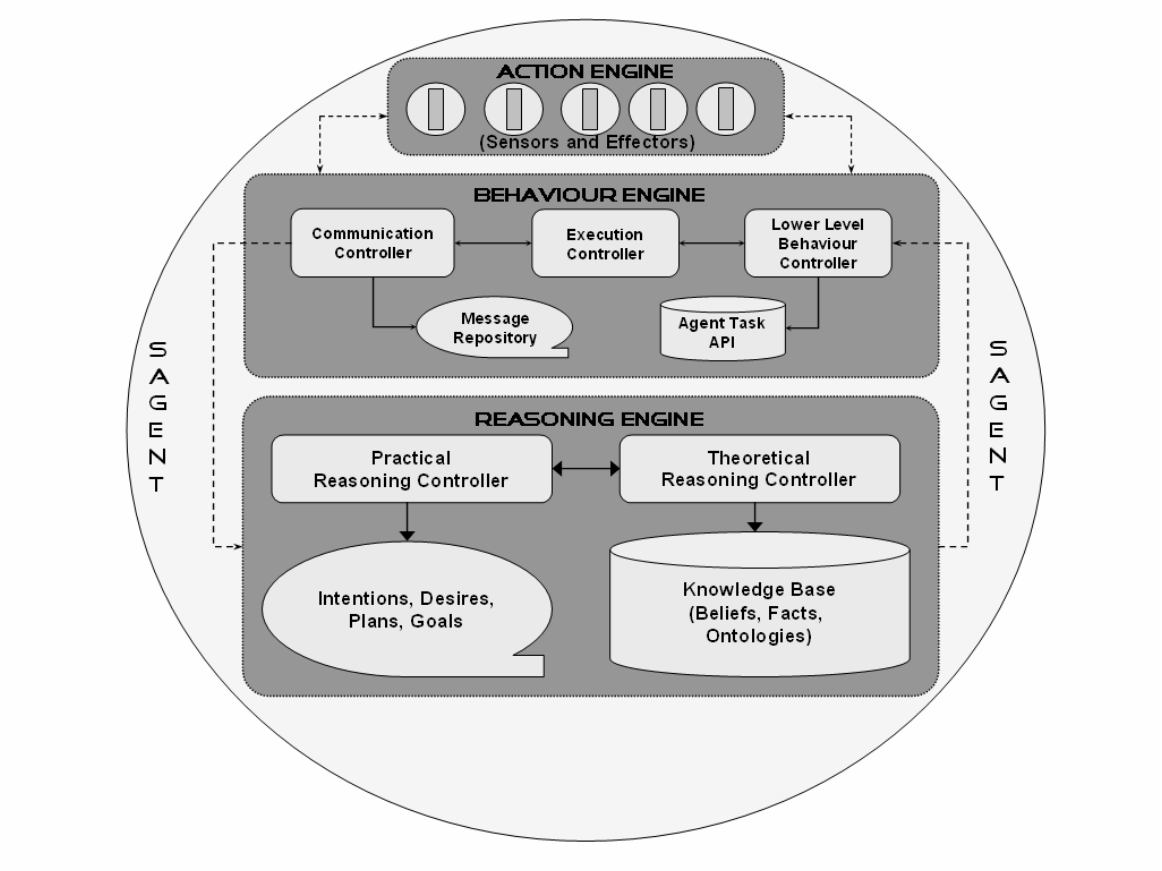

Figure 5.1: Proposed Autonomous Agent Architecture for SAGE

5.2 CORE SUBSYSTEMS OF THE AGENT ARCHITECTURE

The Behaviour Engine theoretically consists of Execution and communication controllers along with sensors and effectors. This system represents the agent's functional and non-functional capacities and skills ("know how"). The Reasoning Engine is composed of a module for

theoretical reasoning – Theoretical Reasoning Controller and a module for practical reasoning- The Practical Reasoning Controller, each with an associated database. Theoretical Reasoning Controller is responsible for agent reasoning that is based on its beliefs only. Where as The Practical Reasoning Controller represents the agent's reasoning about what it should do and consists of a high-level AI planning system.

The high-level interactions between the sub-engines are shown by the dotted. The Reasoning Sub-Engine receives input from the communication controller (which in turn receives its input from the lower level action subsystem) in the form of an ACLMessage and based on the reasoning passes the decided action to be taken to the Lower Level Behaviour Controller. The action is then executed within the Action Sub-Engine with the help of the execution controller. The interactions have been kept to a minimum so as to address the key factors of self-protection and complexity.

One of the most essential and mandatory component SAGE agent's architecture is the agent's knowledge base which contains the representation of the agent, agent behaviour, and the environment perceived by the system. Conceptually conceived as a central repository in the conceptual architecture, the agent's knowledge base now takes the shape of separate repositories and data bases with the constituent sub-engines where necessary. For example a message repository comes with the communication controller for storing messages. Similarly both the Practical and theoretical reasoning controllers have their associated Databases for storing the mentalistic and reasoning concepts of the SAgents.

Chapter 6

BEHAVIOUR ENGINE

6 THE ACTION AND THE BEHAVIOUR ENGINE

6.1 THE INTRA-AGENT CONCURRENCY MODEL FOR SAGE AGENTS - (THE EXECUTION CONTROLLER)

The definition of tasks is critical to completely define the behavior of agents in Multi-agent systems. Tasks not only define the types of internal processing an agent must do, but also how interactions with other agents relate to those internal processes. As it is in the nature of agents to operate independently and to execute in parallel with other agents, we define the agent behavior to be defined as a set of a number of concurrent tasks. Each task specifies a single thread of control that defines the behavior of an agent and integrates inter-agent as well as intra-agent interactions by providing the agent with the ability to have inter-agent negotiations, with each negotiation proceeding at its own pace.

6.1.1 Levels of Concurrency for Agent Architecture

In order to develop truly autonomous and socializing agents, it was essential to provide for two levels of concurrency: firstly at the Inter-Agent level and secondly at the Intra-Agent level. Inter-Agent Concurrency model refers to the system level concurrency between agents in SAGE. In SAGE, this level of concurrency is provided to agents by spawning each agent as a separate JAVA thread. Intra Agent Concurrency model refers to the concurrency at the local level, being internally managed within the agent. We have devised an Execution Controller using a suitable technique to provide for this level of concurrency, without depreciating the efficiency of the multi-agent system.

6.1.2 Proposed Execution Models for Achieving Intra-Agent Concurrency

There are two prospective methodologies or approaches based on which the Execution Controller can be designed for achieving intra-agent concurrency: The Multi-threaded approach employs the standard JAVA threading model based on the notion that is already being employed within SAGE to achieve inter-agent concurrency and the Single Threaded approach- which will attempt to provide intra-agent concurrency using custom-built, split-phase and event-based tasks within a single thread.

6.1.2.1 Analysis of JAVA threading model for the Agent Architecture

The threads appear to be an attractive alternative since they allow the programmer to write a single sequence of operations and ignore the points at which the execution may be blocked. Unfortunately, the typical amount of memory required to support this technique prevents applications from scaling to a large number of threads [20]. The JAVA Threads were not designed for large-scale parallelism [21]. In the current Java releases, JVM (Java Virtual Machine) supports the preemptive round robin scheduling algorithm and maps each thread it initiates to the OS or the kernel, as shown in the Figure 6.1. Since there is one to one mapping of the java thread onto the OS threads both agents and their activities are mapped onto the OS resulting into greater overhead on the OS. Heavy context switching is imposed, since passing the control from one thread to another, is about 100 times slower than simply calling a method. The level of CPU consumption is also very high, considering that an idle agent thread sitting in a continuous loop consumes 100% CPU. We realized that if JVM supported a user level scheduler, then these shortcomings could easily be overcome [22].

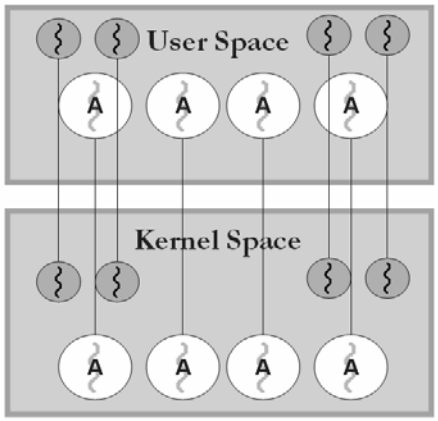

Figure 6.1: One to one mapping of User threads onto kernel Level Threads

6.1.2.2 Cooperative Multitasking using the event-driven approach (Single-Threaded Model for Execution Controller)

Our analysis of agent Task performance reveals that most tasks that an agent performs are dependent on the occurrence of a certain event. Agents perform different tasks based on their states. For instance, there will be some basic tasks, which will be initialized by default when the agent is created. Tasks involving message handlers will only be performed when an agent has to send or receive messages.

In this model we aimed to model agent tasks as user scalable threads or Tasks, which have minimized dependency on OS. These user-scalable Tasks are not mapped directly onto the kernel level; their management is undertaken solely at the application level, making it transparent to the OS kernel. As depicted in Figure 6.2, only the agents that run on the system are mapped on the Kernel space by JVM, whereas the management of the Agent's tasks is carried out at the user level using a Task Library. The model removes itself from the reliance on the JAVA thread library. Instead it adopts an Even-driven mechanism to achieve the concurrent execution of tasks.

Using the approach of Event-driven or event-based programs [22], we handle various agent task situations by registering handlers to respond to various events. This approach does not require threads or synchronization (on a single processor), and leads to efficient, user-schedulable agent task execution, with little support from the operating system.

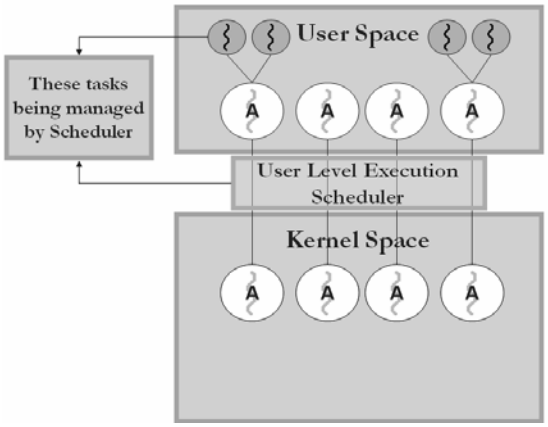

Figure 6.2: User Level Task Management

6.1.3 IMPLEMENTATION

In order to choose the most efficient model for the SAGE's agent architecture, a detailed performance comparison of the two approaches described, was carried out. A prototype implementation of the ideas presented here was written using Java. In the multi-threaded model, all the agent tasks are spawned as threads within the agent. In order to do this the generic Task class was made to extend the Java Thread class. The functionality of the task is incorporated in the run() method of the class. Multiple tasks are initiated by the agent, which also extends the Java thread (every agent is a java thread).

We designed and implemented a simple user-level cooperative multi-tasking package in order to provide intra-agent concurrency. There are two main components of the package: The Execution Controller and a Task library. In this package no threads are spawned for the agent activities. Instead, a generic Task API is created for modeling various types of agent tasks. These are simply modeled as Java Objects. In order to achieve interleaved execution of these tasks, a lightweight Execution Controller, as shown in Figure 6.3 was implemented, which controlled the life-cycle of the task. The five-state model shown in Figure 6.4 was chosen to simulate the life cycle of the tasks inspired from the process model of Operating systems [23].

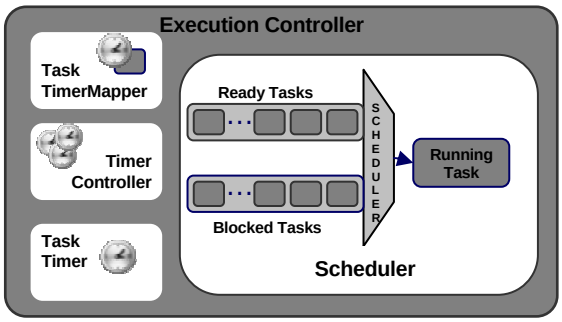

Figure 6.3: Single Threaded Execution Controller for SAGE

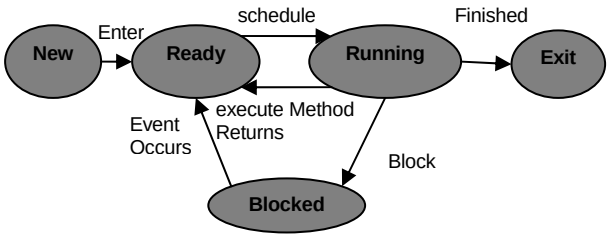

Figure 6.4: Life cycle of an Agent Task.

6.1.4 Mechanism of Execution Controller

The execute() method of the Task contains the code specifying the functionality of the task. There are methods for blocking and restarting Tasks. The task may be blocked while waiting to receive a message or similar events. Upon occurrence of the particular event, the Task is restarted. The agent programmer creates customized Tasks and delegates them to the Execution Controller. The scheduler, which is part of the Execution Controller, implements a Round-Robin scheduling algorithm for defining the interleaving pattern of execution for tasks. There can be only one currently running task. The rest of the active agent tasks are either ready or blocked, which is dynamically maintained by the scheduler using the internal data structures, queues in this case as shown in Figure 6. Scheduling is being done using the method schedule() of the Execution Controller class. The functionality specified in the Task is decomposed in such a manner that in one slice of execution only a certain portion of the task is executed and then the execute() method returns allowing another task to execute based on the cooperative scheduling policy as discussed

earlier. The decomposition of the tasks is dependent on the nature of the tasks and the application to be developed.

If a Task is blocked during the course of its execution for a specified time using the block() method provided, scheduler puts that task into the blocked queue. For each blocked Task a Task Timer is maintained. Mapping between each task and its respective timer is also maintained using the TaskTimerMapper class. This facilitates the TimerController to restart the corresponding Task whenever a Timer expires.

6.2 THE TASK API

6.2.1 Composition Of Agent Role

The preliminary modeling analysis for behavior engine for SAGE reveals the strong need for providing the support for lower level generic task management support. This stems from the fact described earlier that aim of agent architecture is to facilitate the programmer in defining the Agent role. We have analyzed that defining agent role requires agent behavior definition at various stages. Looking into the composition of the agent role it is uncovered that the Agent role can be described by a set of Complex tasks the agent can perform. Each of these Complex Tasks is accomplished when agent executes certain state based tasks which basically treat behavior execution in form of finite state machines and may be involved in the execution of more than one lower level Agent tasks. These are the Tasks which are simplest in nature. These cannot be decomposed further into simpler tasks and form the foundation of the agent behavior. To move up on the ladder of defining the agent role the first rung is to determine these lower level Tasks which are commonly needed by every agent in the society. This hierarchy is depicted as in the Figure 6.5.

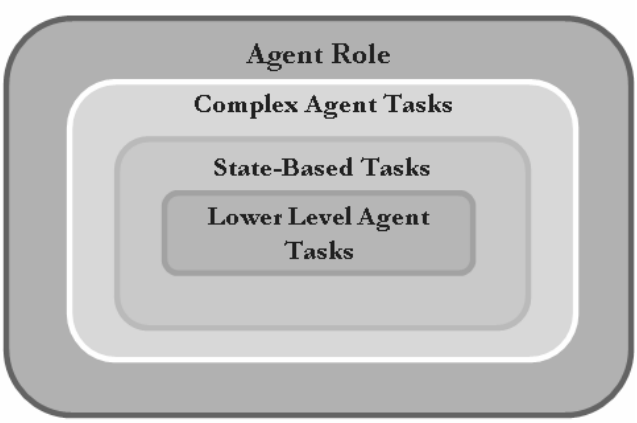

Figure 6.5: Decomposition of Agent Role

As the first step in defining the Agent role we have designed and implemented a Task API for SAGE Agents which includes the set of Tasks needed commonly by every agent in a society. The selection and design of the Tasks has been done, keeping in mind the fact that Agent architecture is a middleware ,thus the requirement is that each of the task units should be generic in nature and suitable to be extended to any domain or application.

6.2.2 The Notion of "Task Unit"

In SAGE, for providing the ability to an agent to perform certain lower level tasks we have introduced the concept of "Task Unit", where various Task Units basically encapsulate the actual role an agent has to perform .We aim at providing the programmer or the user to define the behavior of the Agent by customizing these Task Units to the domain as per required. A single Task Unit represents a task or action that an agent can carry out. To cater for complex actions we have provided the provision to define a Task Unit in terms of several subtask units .Basically what our aim was to provide the set of Task Units needed commonly by every agent in a society.

6.2.3 Classification of Task Units

The first step in the development of the generic Task API was to classify the Task units into various categories. The selection of criteria to classify various Task Units was one of the critical points due to factors: (i) Since a middleware was being developed it was important that classification should not be done keeping in mind a specific application or scenario. (ii) The classification should be easily grasped by the programmer and should not leave any generic action that can be performed by the agents unattended.

6.2.4 Criteria for Classification of Task Units

Keeping in mind the above factors, for identifying various types of the Task units , a criteria was defined based on which we carried out the classification of Task Units into different categories. Task units were categorized based on three general properties: (i)Lifetime: The total execution time of the Task Unit, (ii) Execution Pattern: The manner in which the Task executes e.g. a complex task unit may be executing concurrently or in a consecutive order and (iii)Composition: The composition factor i.e. whether the task is composed of one or more sub-units

The motivation for selecting these characteristics was the fact that no domain specific characteristic could be selected for providing a General purpose support to the programmer. Identified Types of Task Units. Based on the composition of the task units, they were categorized into two main types. These two main categories were further classified into sub-categories based on there different life times and Execution Pattern. The hierarchy is shown in the Figure 6.6.

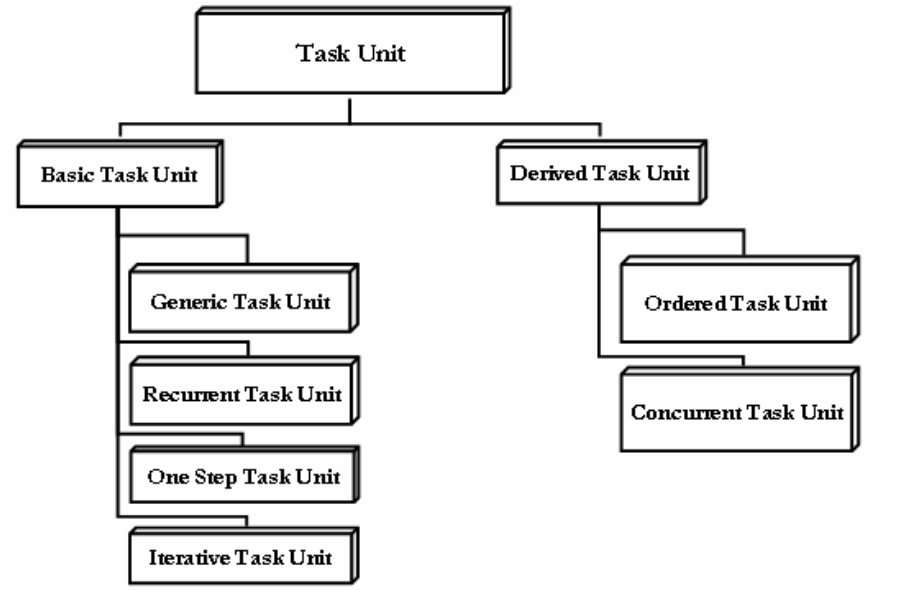

Figure 6.6: Classification of TaskUnits

6.2.5 Basic Task Units

They can not be decomposed further into sub task units as their name indicates they are primitive in nature. These types of task units may be ended as soon as their execution ends or may last for the life-time of the agent. They execute in one go, without interruption. Basic Task units

can be further categorized on the basis of their life time. Some general purpose types identified are: (i) Generic Task Unit: A general purpose type for providing the programmer to implement any task that is primitive to the application being created.(ii) Recurrent Task Unit: This Task unit stays active as long as its agent is alive and is repeatedly called after every event, (iii) One Step TaskUnit: This type of task unit is executed once after which it dies and (iv) Iterative Task Unit: This type of task unit is executed a fixed number of times as specified by the programmer.

6.2.6 Derived Task Units

They are composed of more than one basic task units specified as their sub task units /children task units. These types of task units may be ended as soon as their execution ends or may last for the life. Time of the agent also these task units may exist for a programmer specified duration .Also in some cases their existence may be dependent on their execution Pattern. As they are composed of more than one task units so their Execution Pattern depends on how the sub task units are executed. Derived Task Units can be further categorized on the basis of their composition. Some general purpose types identified are: (i) Concurrent Task Unit: Contains a set of subtask units that are executed in parallel and (ii) Ordered Task Unit: Contains a set of subtask units that are executed in a sequential or a consecutive manner

6.2.7 Design of the Task API

The Overall design of the Task Unit is well depicted in Figure 6.7. Each Task Unit is endowed with various handlers. The functionality of these handlers is specific to the type of the task being implemented. These handlers are responsible for controlling the overall functionality of the Task Unit.

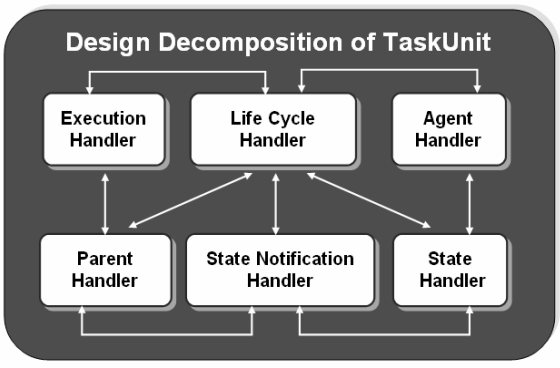

Figure 6.7: Design Decomposition of Task Unit

All types of the Task Units are provided with an Execution Handler which is responsible for defining the execution pattern and controlling the execution of the Task Unit. Also all types of the Task Units are provided with a couple of state handlers. State Change Handler is basically responsible for controlling any change in the state of the Task Unit and taking the required actions in response to that state change. The other handler is the State Change Notification Handler which is responsible for notifying the concerned Task Units about the state change occurred in the task unit owning the handler.

Design and execution pattern of Derived Task Unit is more complex due to its complex composition. For Derived Task Units a Parent Handler is provided which is responsible for the parent child relationship management. As in case of Primary Task Units it is not possible to have subTask Units so this handler is not required with them so the implementation design of the Basic Task Unit was kept simple and the methods implementing the parent handlers were not extended.

6.2.8 Algorithmic Design of Derived Task Units

For Derived Task Units the execution and state change notification algorithms were designed which are depicted in the flowcharts. For Derived Task Units these algorithms had to be designed explicitly because in case of these Task Units the execution of subtask Units is inter dependent. The general Execution Pattern of Derived Task Unit is depicted in the flow chart in Appendix A.

The state notification handling in case of Ordered and Concurrent Task Units is different from each other because of their different execution patterns. The design of State change handling in both Ordered and Concurrent Task Units is depicted in form of flow charts in Appendix A

6.2.9 State Based Modeling for SAGE Agents

State automata are a very often used modeling technique in control theory and reactive systems. As agents are ought to be reactive entities the idea of state-based modeling of an agent is to let the agent have several states (which can indicate rather complex assumptions) and the decision about the next action is then based on both, state and situation. State-based programming for agent behavior has been explored to model agent communication and behaviors, as well more general reactive systems. The State based modeling of the agent behavior results in achieving reactivity and proactivity for the agent at the minimal level. Also supports the achievement of enhanced social ability. And to model communication and Interactive agent behavior

6.2.9.1 Types of State Based Modeling

There are two main types of techniques used for state Based Modeling: (i) Finite State Machines, (ii) Hierarchical State Machines. Finite State machines have fixed states and transitions and there is no state modeling within states. On the other side the Hierarchical State machines have Dynamic Transitions between states and allow layers of state modeling Agent Behavior as a State Machine

At the agent level, we describe the behavior of each individual agent by a state machine. In that case each state in the State Machine for each agent is an *agent state*. Every agent consists of a set of agent states, but only one agent state can be active at a particular time.

6.2.9.2 Design of FSM Task Unit

At the initial level we have devised an FSM Task Unit SAGE Agents which allows the modeling of Agents behavior as an FSM. FSM maintains the transitions between states and selects the next state behavior to execute. States are registered, named and stored. Some Limitations of FSM modeling includes there are fixed transitions between the states. Once the start and finish states are registered, the state machine is ready for execution. After one state has executed it returns one and only fixed transition to be made. The transitions only serve to link states; they do not encapsulate agent behavior. The overall design of FSM is shown in Figure 6.8.

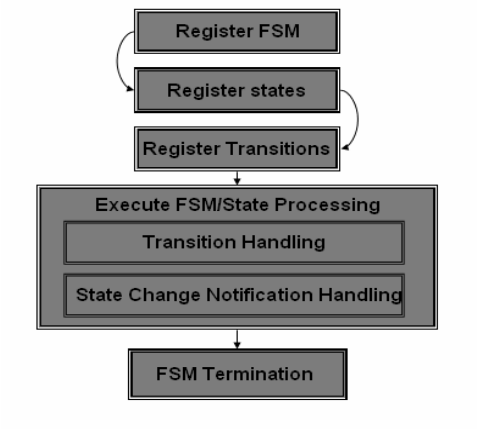

Figure 6.8: Design of FSM

6.2.10 Implementation of the Task API

The implementation of the Task API has been done as a hierarchy of classes. As seen from class diagram in Appendix A, the root class Task Unit lies at the top. This is an abstract class that implements the handlers common to all subclasses. Each handler is implemented as a set of

methods that define the functionality of that handler. The programmer can put the functionality of specific actions, it wants the agents to perform in the execute() method of the Task Unit. But the programmer is not responsible for controlling the underlying execution of the Task Unit. This is done automatically by the Execution Handler. Also incase of Derived Task Unit the programmer is only responsible for defining the relationships between Task Units but their management is kept transparent to the programmer by using the functionality of the associated parent handler. Each type of Task Unit is implemented as separate class extended from this generic root class.

6.3 THE COMMUNICATION CONTROLLER

A multi-agent system (MAS) is a system containing more than one agent in which agents can interact and hence influence each other's behavior. Groups of agents can do things that individuals cannot e.g. Routing over distributed domains, meeting schedule, etc.

6.3.1 Agent Communication – A Layered Model

All the communication in an MAS may be viewed as a layered model depicted in Figure 6.9. Agents require high level conversational support at the application layer on top of the underlying message transport infrastructure in order to enable high level inters agent interaction. At the application level they require a well defined communication and content language in which they can exchange messages. In addition to this they require well defined conversation patterns in form of interaction protocols in order to engage in intelligible conversations. This support was to be included in the communication controller of the agent architecture.

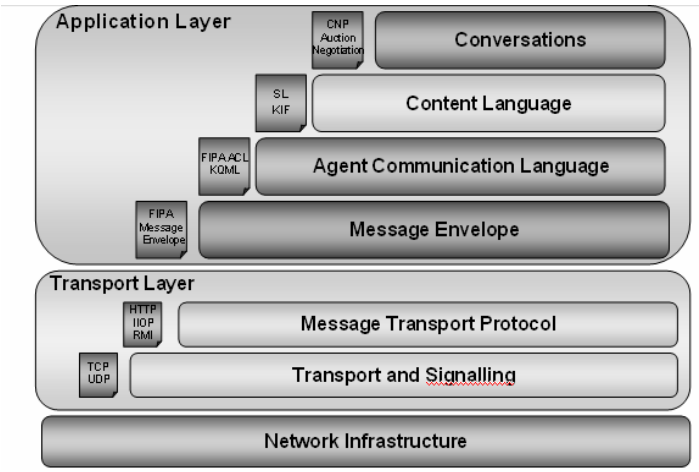

Figure 6.9: Layered Model of Communication in MAS

6.3.2 Agent Conversations and Interaction Protocols

By their nature, agents can engage in multiple dialogues, perhaps with different agents, simultaneously. The term *conversation* is used to denote any particular instance of such a dialogue. Ongoing conversations between agents often fall into typical patterns. In such cases, certain message sequences are expected, and, at any point in the conversation, other messages are expected to follow. These typical patterns of message exchange are called *interaction protocols*. Thus, the agent may be concurrently engaged in multiple conversations, with different agents, within different IPs.

Conversations are composed by one or more messages. They are ruled by agent interaction protocols (AIP). Given a message (and its specific performative), the AIP defines which is the set of performatives that could be associated to the following messages. If a message does not comply with this rule, the agent could not understand it.

6.3.3 Design of communication Controller

An important contribution for independent and autonomic behaviour of agents is the ability to communicate with other agents and software components. To facilitate this ability a communication controller was designed as shown in Figure 6.10. FIPA-ACL has been implemented as part of the system framework of SAGE [3, 4]. The ACLMessage Interface has been designed to provide a dynamic interface to the programmer to utilize the features of ACL Module. The ACLMessage interface provides reusable TaskUnits for sending and receiving messages to alleviate programmers from writing tedious and redundant code.

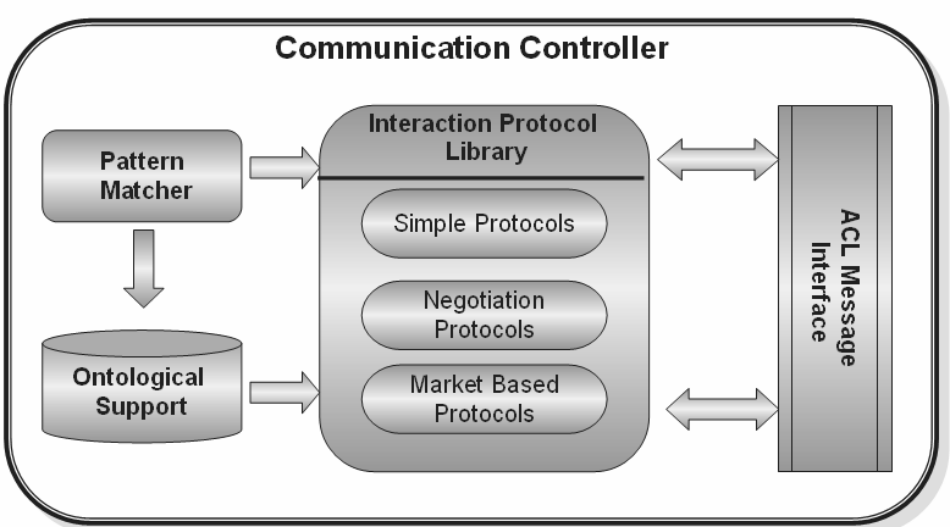

Figure 6.10: Design of Communication Protocol

The prime purpose of the Communication Controller is to provide high-level conversation management support to the *SAGE Agents*. A library of Interaction Protocols has been provided allowing *SAGE Agents* to communicate in a one-to-one or one-to-many mode based on FIPA-Interaction Protocol library specifications [27]. This alleviates existing restrictions of agent communication mechanisms, which are designed to enable communication between agents only of the same platform. Having one common means of communication, in the form of a shared middleware, also reduces the complexity of the environment and simplifies security barriers. The protocols range from simple query and request protocols, to more complex ones, such as the well-known negotiation protocols e.g. contract net negotiation protocol or the market based English and Dutch auctions. The protocols are modeled based on the notion of State-Based Modeling for Conversation Protocols.

Complex knowledge management domain may lead to complex interactions between *SAGE Agents*; in order to support this complexity it is necessary to have a good support for content language and ontology. General support for ontologies based on a model of the FIPA-SL content language has also been made to facilitate programmer and also allow for intelligible and more meaningful interactions. The importance of Ontologies for interaction protocols has been highlighted in [24].

The FIPA-specified support for content and ontologies and interactions allow for content based nature of inter-agent communication. The content-based nature of the communication

decouples the sender of a message from its receivers, and promotes a context aware and autonomic behaviour of *SAGE Agents* in the system.

An important component of the Communication Controller is the pattern matcher. The messages received through the Execution Controller are recognized and analyzed through the pattern matcher before being passed onto the Practical Reasoning Sub-Engine for reasoning purposes. The Pattern Matcher also allows for Message Template building and customized message patterns for utilization in Interaction Protocols and for their semantic interpretation

Along with the Communication Controller each SAGE Agent has been provided with a central message repository which serves as a useful abstraction mechanism for storing messages and forwarding when needed or requested. The Communication Controller interacts with this message repository allowing messages to be sent and received when Interaction Protocols are used. The processing of the messages remains the responsibility of the owner and dependent on the domain.

The design of Communication Controller is being enhanced so that *SAGE Agents* are also given the ability to dynamically change the type and nature of messages they would like to receive as well as produce different notifications based on their current context.

6.3.4 Detailed AUML Design of Agent Interaction Protocols for SAGE

Agent Interaction Protocols (AIPs) provide us with reusable solutions that can be applied to various kinds of message sequencing encountered between agents. AUML suggests some extensions to the standard UML for the specification of Agent Interaction Protocols [28]. AUML basically adopts a layered approach to protocols:

6.3.4.1 Level 1 - Represent the overall protocol (packages, templates)

Level 1 of the protocol basically defines the overall protocol in form of packages and templates. Packages are responsible for aggregating the modeling elements into conceptual wholes. Protocols can be codified as patterns of agent interaction. AUML packages can group sequence diagrams (to model protocol patterns).The template representation of the protocol allows for the depiction of customized protocol representation. As the parameterized packaged protocol can be treated as a pattern that can be customized to any problem domain. A template is a parameterized model element whose parameters are bound at model time.

6.3.4.2 Level 2- Represent interactions among agents (sequence, collaboration, activity, state diagrams)

Level 2 design of the AUML design for the interaction protocols all emphasizes interactions among Agents. In level2 there is much choice available for describing the interactions among the agents. Interactions can be depicted in many forms like Extended Sequence Diagrams (concurrent threads of interaction), Collaboration Diagrams, Activity Diagrams and State Charts. However, all of these representations actually contain the same information about the agents.

6.3.4.3 Level 3- Represent internal Agent Processing (activity and state diagrams)

This level depicts the lowest level of AIP specification. It illustrates the internal processing of the agent that is not aggregated. This level fulfills the requirement of spelling out detailed processing with in an agent. State Charts and Activity Diagrams are commonly used at this level too. These diagrams are suitable for depicting the internal processing going with in the agent.

6.3.5 Implementation of Communication Controller

The implementation of the interaction protocols was aided by the FSMTaskUnit of the Task API. Both the initiator and the responder roles were extended from this very class to model the behaviour of the Interaction protocol in the form of a state machine as per the Level 3 Design of the IPs. The overall implementation model is depicted by the class diagram in Figure 6.11.

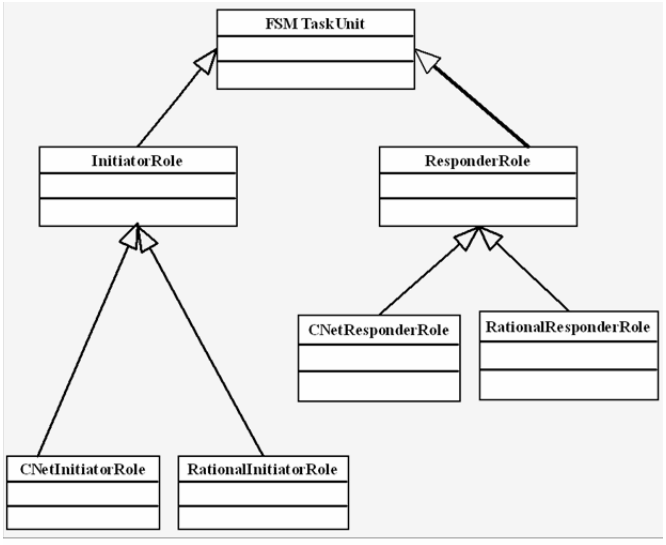

Figure 6.11: Class Diagram for Interaction Protocols

Chapter 7

REASONING ENGINE

7 THE REASONING ENGINE

The Reasoning Engine is composed of a module for theoretical reasoning – Theoretical Reasoning Controller and a module for practical reasoning- The Practical Reasoning Controller, each with an associated database. Theoretical Reasoning Controller is responsible for agent reasoning that is based on its beliefs only. Where as The Practical Reasoning Controller represents the agent's reasoning about what it should do and consists of a high-level AI planning system.

7.1 THEORETICAL REASONING CONTROLLER FOR SAGE

Each problem that I solved became a rule which served

afterwards to solve other problems.

- *Rene Descartes*

In the world of the agents, where one agent is capable of reacting to and reasoning about events which occur in its environment, execute actions and plans in order to achieve goals in his environment, and communicate with other agents. Some of them can be quite simple, but it is normal that when the system provides certain intelligent behavior (for example, capacity to plan, reasoning and so on), it means that some agent with a powerful cognitive model exists. To program those behaviors it is necessary to use something more than classic procedural languages.

We are aiming to make SAGE agents intelligent. First of all we can see that an intelligent agent is an agent that has capability to make inferences based upon the knowledge it has as depicted in the Figure 7.1.

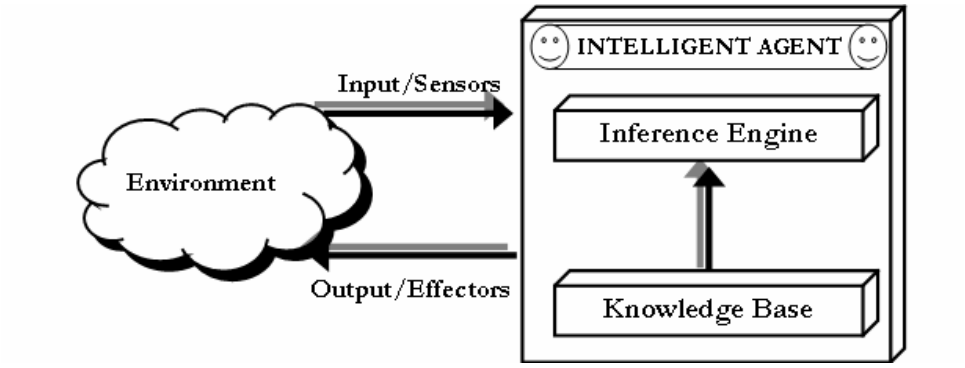

Figure 7.1: Intelligent Agent Model

For making the SAGE agents intelligent they need a powerful cognitive model at the agent architecture level. The first step towards making the agents intelligent is to endow agents with the capability of theoretical reasoning. Theoretical reasoning is the reasoning of the agents that is directed towards their beliefs. Agents with the capability of theoretical reasoning of are often reactive, having the added capability of making inferences. Capability of theoretical Reasoning can be easily embedded with in the agents by making them to follow a chain of rules [25]. Rules are needed for expressing participant agent‘s individual decision making as well as the contracts that bind it to other agents .Nailing down the desired decisions is nontrivial and often involves incrementally augmenting a given specification. Likewise, contracts are frequently partial, especially early in the process of being designed .This is the reason that rules are highly suited to specifying such decision making and contracts in the agent domain. Realizing the importance of Theoretical Reasoning with in the agents, a Theoretical Reasoning Controller was conceived for SAGE's Agent Architecture.

7.1.1 Synopsis of Integration of MAS and Expert Systems

The design of Theoretical Reasoning Controller was based on the concept of integrating the Agent Architecture with a rule based expert system. Where expert systems are actually computing systems designed to imitate higher level cognitive processing. Expert systems simulate human reasoning in some domain. An expert system performs a set of activities traditionally associated with highly skilled or knowledgeable humans activities like medical diagnosis and stock market analysis. Admittedly, we don't want our agents to be skilled in these fields; however we do want them to be competent entities in the environments in which they live.

There reasoning is done by heuristic or approximate methods. Rule based Expert Systems are that type of Expert Systems which Reason" using IF…THEN…ELSE rules. They have the capability to reason deductively (forward-chaining) or inductively (backward-chaining). Rule-based expert systems combine well with agents for two reasons: First of all rules make a compact definition of behavior possible. Where in its simplest form, a behavior is a set of actions and the conditions under which those actions should happen. Secondly from a purely visual perspective, once you get the hang of the notation for rules, it's much easier to understand system activity by examining a set of rules than by unwinding the equivalent nested if/then/else code. Summarizing all we can say that Behavior is the basis of autonomy. And rules are a concise definition of behavior. And expert systems are very good and very efficient at applying rules.

Although to construct our agents we used procedural languages, need to integrate other mechanisms as systems based on rules for example JESS etc. We aim to blend two technologies - join a platform for developing multi agent system (MAS) with an Expert System (ES) e.g.(Java Expert System Shell) JESS [29]. Using JESS, we can build applications that have the capacity to "reason" using knowledge supplied in the form of declarative rules. JESS, which is implemented in Java, is increasingly being used in agent toolkits such as FIPA-OS, JADE and JATLite, and to support complex reasoning.

The main idea was to encapsulate expert system rules within the agents. Rule based Expert systems are well suited for the purpose of theoretical reasoning in SAGE agents because Rules make a compact definition of behavior possible as described above. Also Nowadays, is more or less easy to join a platform for developing multi-agent system (MAS) with an Expert System (ES) which allows building of such agents that have the capacity to "reason" using knowledge supplied in the form of declarative rules.

7.1.2 Detailed Design for Theoretical Reasoning Controller for SAGE

As described above the expert system to be integrated with SAGE was decided to be JESS.JESS provides a fully developed Java API for creating rule-based expert systems. SAGE has been completely implemented in Java. Its capabilities can only be fully exploited by using the Java programming language.

JESS is a rule-based expert system which in its simplest terms, means that Jess's purpose is it to continuously apply a set of if-then statements (*rules*) to a set of data (the *knowledge base*). Two important constructs make up a JESS knowledge base: facts and rules. A *fact* is a construct that defines a piece of information that is known to be true. A *rule* is nothing more than an if/then statement that defines the set of facts that must be true (the *if* part) before a set of actions (the *then* part) can be executed. JESS is a powerful expert system because actions themselves can assert new facts. When this happens additional rules apply and their actions are executed. JESS is one of the efficient expert Systems as it uses a very efficient method for inference known as the Rete algorithm, which alleviates the inefficiency generally associated with expert systems by remembering past test results across iterations of the rule loop. The working model of Jess is shown in Figure 7.2.

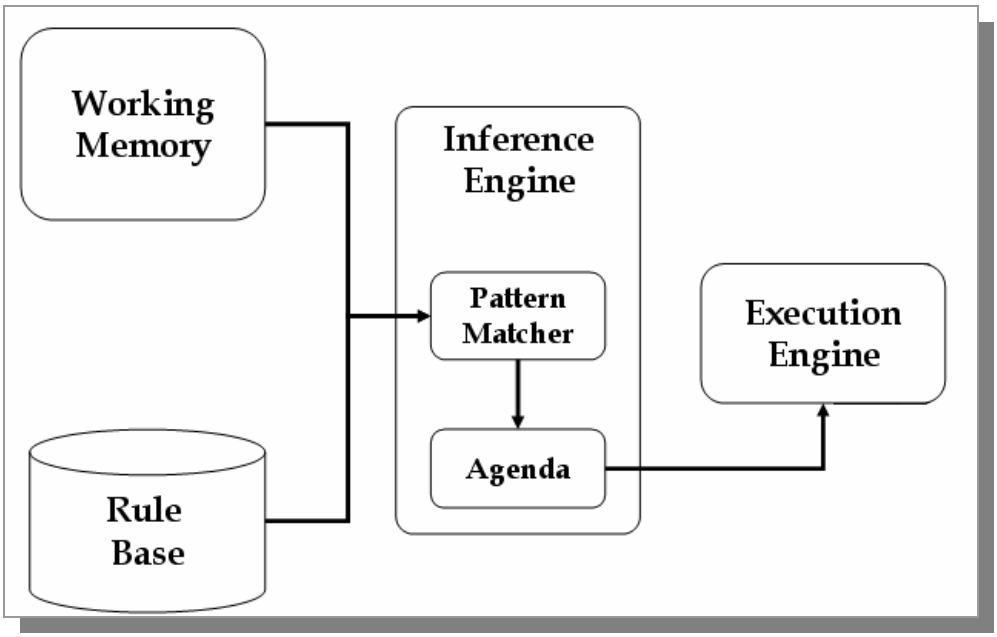

Figure 7.2: Working Model of JESS

7.1.3 Execution Model for JESS

SAGE Theoretical Reasoning Controller is designed to work in a manner such that for each received message, it asserts a fact in the JESS engine that describes the message. Then the inference engine of JESS comes into play to fire the rules that match the fact. This allows a JESS program to control sending or receiving messages and creating or destroying Task Units in response to the message received. An implementation practice that we have found useful is the usage of JESS to control the activation and deactivation of the SAGE TaskUnits by implementing, as a consequence, a mixed reactive-deliberative agent architecture (where JESS plays the deliberative role and the SAGE TaskUnits play the reactive role). The working model is shown in Figure 7.3.

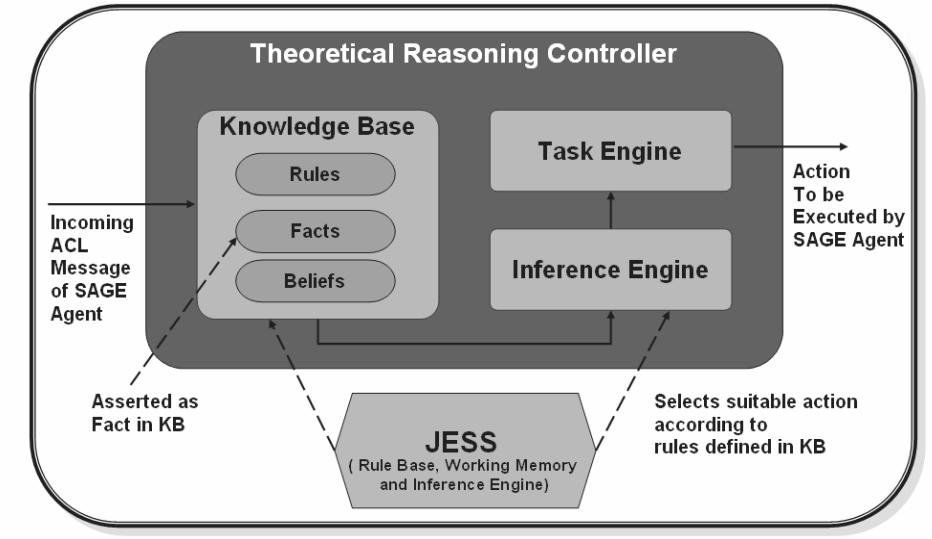

Figure 7.3: Execution Model of Theoretical Reasoning Controller

7.1.4 Implementation model of Theoretical Reasoning Controller

The implementation of the jess Model was done in a form of a complete package. There were three main requirements for the implementation: Conversion between Jess fact to ACLMessage and vice versa. It was also required to have a Jess controller for controlling commands of the Jess engine directly from SAGE. Plus it was also required to implement user functions for allowing SAGE commands to be executed from within JESS. All these requirements were catered for in discrete and well defined modules, namely Jess ACL Converter, Jess Engine Controller and Jess Message Porter.

7.1.4.1 Jess ACL Converter

It contains methods for conversion between ACL Message to Jess Fact (String) and vice versa

7.1.4.2 Jess Engine Controller

It provides methods for asserting facts and executing jess commands, Parses the rule base file and provides methods for running Jess Engine

7.1.4.3 Jess Message Porter

It implements the Jess user function for sending messages directly from Jess Engine.

7.2 THE PRACTICAL REASONING ENGINE

Moving ahead in the journey towards development of the fully autonomic *SAGE Agents*, we analyzed that autonomy of agents is directly related to their capacity to make decisions without intervention of the human users [10]. We aim to endow the SAGE users with the need only to make relatively less frequent predominantly higher level decisions, which the system will carry out automatically via more numerous ,lower level decisions and actions. To attain this aim we need *SAGE Agents* to be self adaptive so that they can find ways to best interact with neighboring agents and to describe themselves to other agents. Practical Reasoning is seen as the most prospective means of achieving self-adaptation for *SAGE Agents* which involves Planning, Deliberation and Goal-Directed behaviour.

7.2.1 Means for Practical Reasoning

The Belief-Desire-Intention (BDI) Model [30, 31] is seen as a preliminary means to provide a base for practical reasoning and thus self-adaptation. The relevance of the BDI model can be explained in terms of: i) Its philosophical grounds on intentionality and practical reasoning [32]; ii) Its elegant abstract logical semantics and different implementations, e.g., IRMA, and the PRS-like systems, including PRS, dMARS, and iii) Successful applications, e.g., diagnosis for space shuttle, factory process control, business process management as suggested in [32].

The BDI model for *SAGE Agents* has been designed to support the event-based reactive behaviour as well as pro-active behaviour. This BDI incorporation within the *SAGE Agents* can further be extended with learning competencies for MAS situations.

7.2.2 Design of Practical Reasoning Controller

We have designed the BDI based Practical Reasoning Engine as composed of two sub-engines. BDI-Deliberation sub-engine includes a deliberation process (as shown in Figure 7.4) contains the deliberation process responsible for the analysis and processing of agents goals and beliefs. The Deliberation process has been designed keeping in view all the factors and considerations mentioned in [33]. Incoming messages, as well as internal events and new goals serve as input to the Deliberation process. Based on the results of the deliberation process these events are dispatched to already running plans, or to new plans instantiated from the plan library.

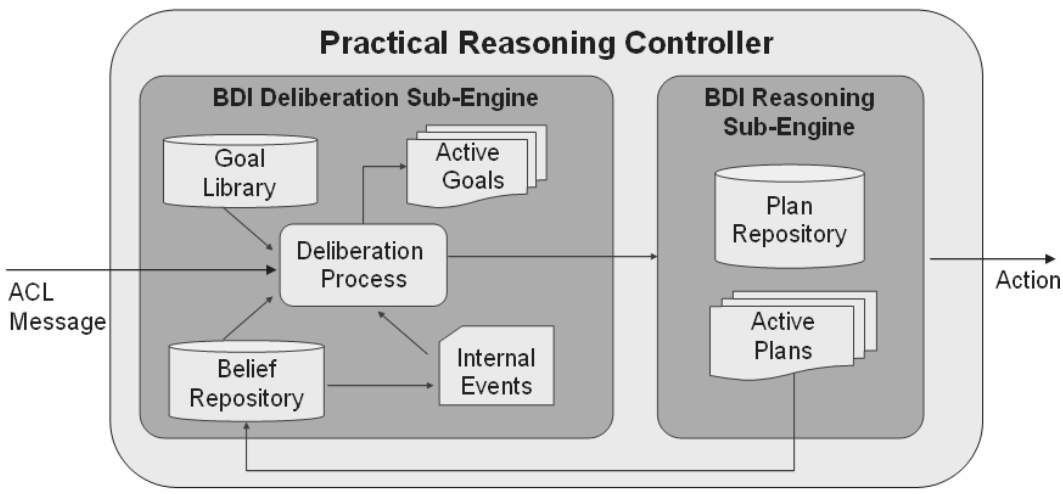

Figure 7.4: Design of Practical Reasoning Controller

The BDI Reasoning Engine is responsible for determining the deliberative attitudes, defined as plans. The Reasoning Engine determines these plans on the basis of the goals selected by the BDI Deliberative Engine. Running plans may access and modify the belief base, send messages to other agents, create new top-level or sub-goals, and cause internal events. SAGE Agents use the plan-library approach to represent the plans of an agent, instead of performing adhoc planning.

7.2.3 BDI Execution Model

The execution model for Practical Reasoning Sub-Engine is event-based. Everything happening inside a SAGE agent is represented as event. Message events denote the reception of an ACL message. Goal events announce the emergence and the achievement of goals, and internal events (called stimuli) report e.g., changes of beliefs, timeouts, or that conditions are satisfied.

Chapter 8

UNIT TESTING

8 UNIT TESTING OF AGENT ARCHITECTURE

Unit testing of each module was carried out before integration with SAGE. Various test scenarios were devised and implemented for each module to verify the accuracy of the functional and non-functional aspects of the module.

8.1 TESTING AND EVALUATION OF EXECUTION CONTROLLER

To evaluate the performance overhead imposed by the threaded tasks in the Multi-threaded model, the implementation of our user-level event-driven Execution Controller package was compared with the prototype implementation of the multi-threaded tasking model as described above.

8.1.1 Performance Criteria

The metrics used to evaluate the performance of each, were turnaround time and throughput. These parameters were chosen, since they are quantitative in nature and can be easily measured online [23]. These parameters provide an accurate measure of the efficiency of the system. Turnaround time may formally be defined as the time interval between the submission of a task and its completion whereas Throughput is the number of tasks completed per unit time.

8.1.2 Test scenario

A simple task was created and the nature of the task was kept exactly the same for both models. The number of tasks was gradually increased and throughput and turnaround time for both models were calculated.

Some difficulty was faced while creating the same test scenario for both the models, since the parameters being observed were highly dependent on the number of context switches for the threads and the number of method calls for the event-driven tasks. In order to provide a level playing field, the maximum level of switching in case of the single treaded model was created. It was observed that on average a thread switch occurs after the execution of one instruction. Therefore the tasks were also made to switch after executing one instruction effectively, in the case of the single-threaded model. In addition the tasks created using the Java threading model were kept to the same priority level since non-preemptive, round-robin cooperative scheduling policy is

being followed in the single threaded execution model. This gave an accurate reflection of the amount of overhead involved, when threads switch as compared to a method call or return.

However the constraints still remain in the testing environment as eventually the control lies with JVM and then the OS. So the results are bound to vary given the number of applications running and the OS being tested on. For the tests conducted same environment was provided to both models.

8.1.3 Preliminary Prototype Implementation

It is clearly evident from Figure 8.1that the single-threaded cooperative multitasking model gives relatively better results than the multithreaded model in terms of total execution time for the tasks. Initially, when the number of tasks is few, the difference in execution time is minimal - the difference being 20 seconds when the number of tasks is 5. Gradually as the number of tasks is increased, the difference in execution time also increases, and reaches 100 seconds when the number of tasks increase to 25. The difference in throughput, which is almost constant, shown in Figure 8.2 reflects that the throughput obtained for single-threaded model is considerably better than the Multi-threaded model. It is also reflected that the average turnaround time for each task is comparatively greater for the multi-threaded model as compared to the single threaded one.

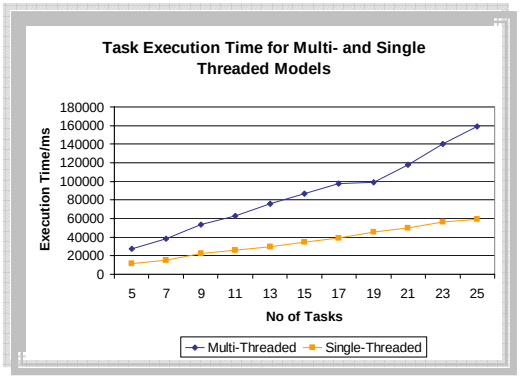

Figure 8.1: Total Execution Time vs. the number of tasks

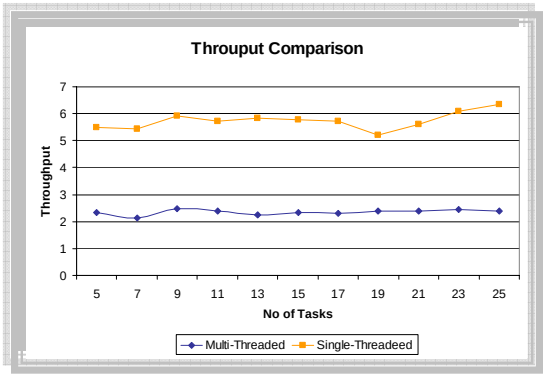

Figure 8.2: Throughput vs. Number of tasks

8.1.4 Results for Prototype Implementation

The Prototype Implementation done using JDK1.4.0 alone was first carried out and results observed. It is clearly evident from Figure 8.3 that the single-threaded cooperative multitasking execution model gives considerably better results than the multithreaded model in terms of total

execution time for the tasks. Initially, when the number of tasks is few, the difference in execution time is minimal - the difference being 5 seconds when the number of tasks is 5. Gradually as the number of tasks is increased, the difference in execution time also increases, and reaches almost 180 seconds when the number of tasks increases to 100. The difference in throughput, which is almost constant, shown in Figure 8.4, reflects that the throughput obtained for single-threaded model is relatively better than the Multi-threaded model. It is also reflected that the average turnaround time for each task is comparatively greater for the multi-threaded model as compared to the single threaded one.

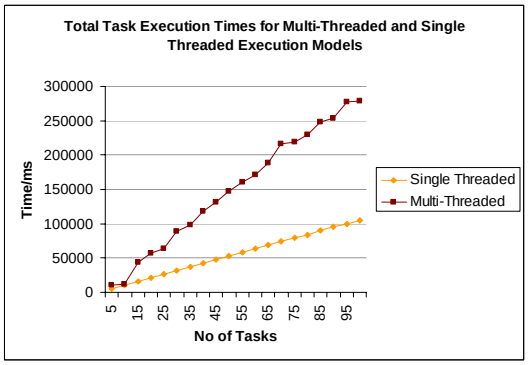

Figure 8.3: Total Execution Time vs. the number of tasks

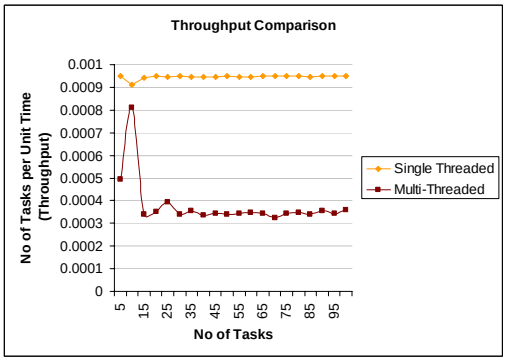

Figure 8.4: Throughput vs. Number of tasks

8.1.5 Results for Prototype Implementation with SAGE

The testing carried out for the prototype implementation clearly favored the single threaded execution model as described above. In order to provide for satisfactory results and testing, the prototype version of the two models proposed and implemented above were individually integrated with SAGE and tested again in a similar manner. The results obtained are shown in Figure 8.5 and Figure 8.6. The most simplified test scenario was created using a single SAGE Agent to generate tasks and their execution was controlled by the execution controller object that was assigned to each SAGE Agent.

It is not surprising that the results obtained with SAGE are almost identical to those obtained as a standalone implementation without SAGE. This owes to the fact that the Agents in SAGE are themselves implemented as JAVA threads. The only overhead incurred is due to the Service Agents, such as AMS, DF, and VMA. This is same for both the models. The general trend for the total execution times for tasks is linear, as was previously the case, with difference between the single threaded and the multi-threaded models increasing with increase in the number of tasks.

There are a few slight deviations from this linear trend which are considered allowable again owing to the fact that service agents are running concurrently with the active Agent under testing.

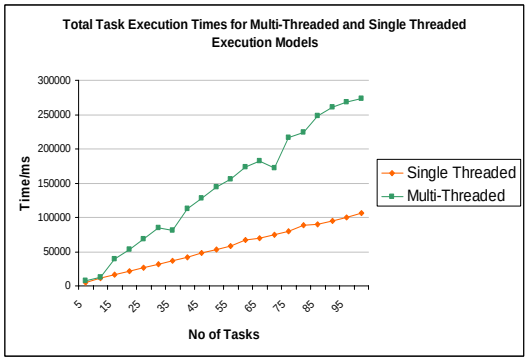

Figure 8.5: Total Execution Time vs. the number of tasks

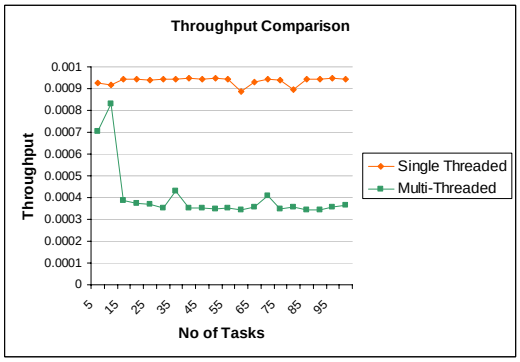

Figure 8.6:Throughput vs. Number of tasks

8.2 TESTING AND EVALUATION OF TASK API

The testing of the Task API required testing of both the functional and logical accuracy of the various types of tasks defined. For this purpose various test examples were implemented and executed. The formulation of examples was carefully done in such a manner that in addition to the functional validation, it should also check for the logical flow of the Task Units. The performance evaluation of the Task API was carried out in conjunction with the overall architectural evaluation which may be referred to in Chapter 9.

8.3 TESTING AND EVALUATION OF COMMUNICATION CONTROLLER

Similarly the same form of validation was carried out for the all the interaction protocols and the related modules implemented as the part of the communication controller. Combined Evaluation of the Task API and the Communication Controller was performed.

8.3.1 Test Scenario

Two different test scenarios were created. Both the protocols were initially tested on a single machine and then on distributed platform.

8.3.2 Results for FIPA Request

The results obtained for FIPA Request on Distributed Machines were as shown in Figure 8.7. As can be clearly seen that the results show a linear increase in total time execution with respect to the increase in the number of agents. This trend verifies the scalability of the architecture and its efficient and lightweight nature.

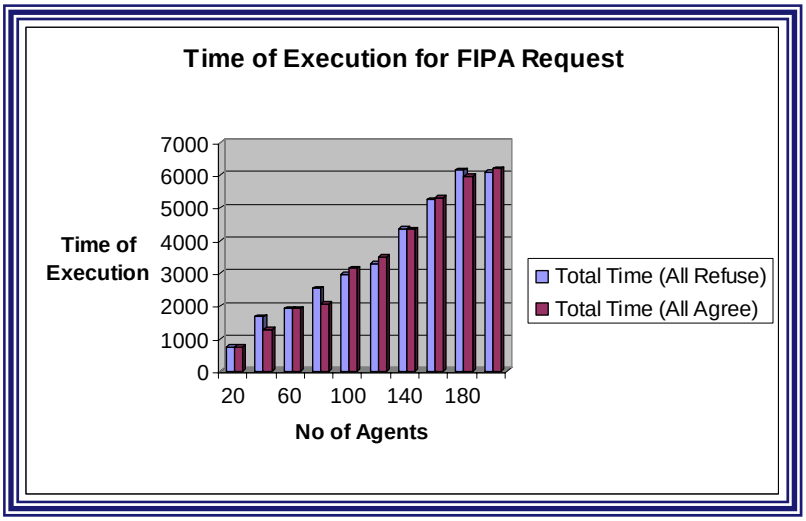

Figure 8.7: Total Execution Time vs. No of Agents for FIPA Request

8.3.3 Results for FIPA Contract Net

The results obtained for FIPA Contract Net on Distributed Machines were as shown in Figure 8.8 and Figure 8.9. As can be clearly seen that the results show a linear increase in total time execution with respect to the increase in the number of agents. This trend verifies the scalability of the architecture and its efficient and lightweight nature.

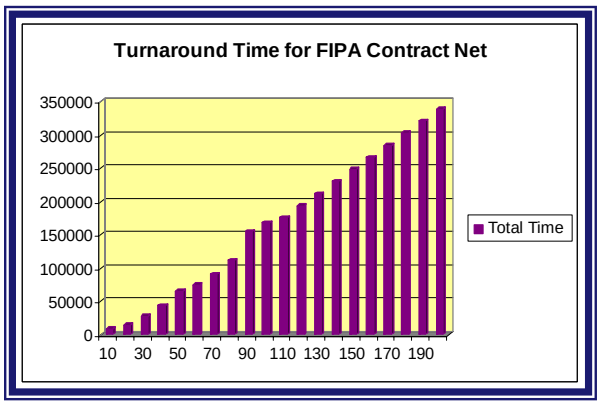

Figure 8.8: Total Execution Time vs. No of Agents for FIPA Contract Net

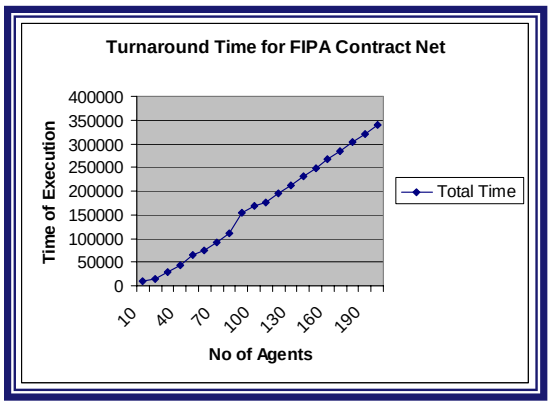

Figure 8.9: Total Execution Time vs. No of Agents for FIPA Contract Net

8.4 TESTING OF THEORETICAL REASONING CONTROLLER

For testing of the Theoretical Reasoning Controller for SAGE a small test application was created. The application invokes the Jess engine using the jess package and interacts with the Test Agent of the SAGE platform. The performance evaluation of this module was carried out in conjunction with the overall architectural evaluation which may be referred to in Chapter 9.

Chapter 9

INTEGRATION WITH SAGE AND OVERALL SYSTEM TESTING

9 INTEGRATION WITH SAGE AND OVERALL SYSTEM TESTING

9.1 INTEGRATION WITH SAGE

The integration with SAGE was carried out in a iterative manner. Each module was encapsulated in a well defined package. The entire distribution of the package was encapsulated as a whole in one main package namely the agent package.

9.2 OVERALL PACKAGE DISTRIBUTION

The Figure 9.1 shows the overall package diagram for the agent architecture for SAGE. The agent package encapsulates all the sub packages which formulate the agent architecture of SAGE.

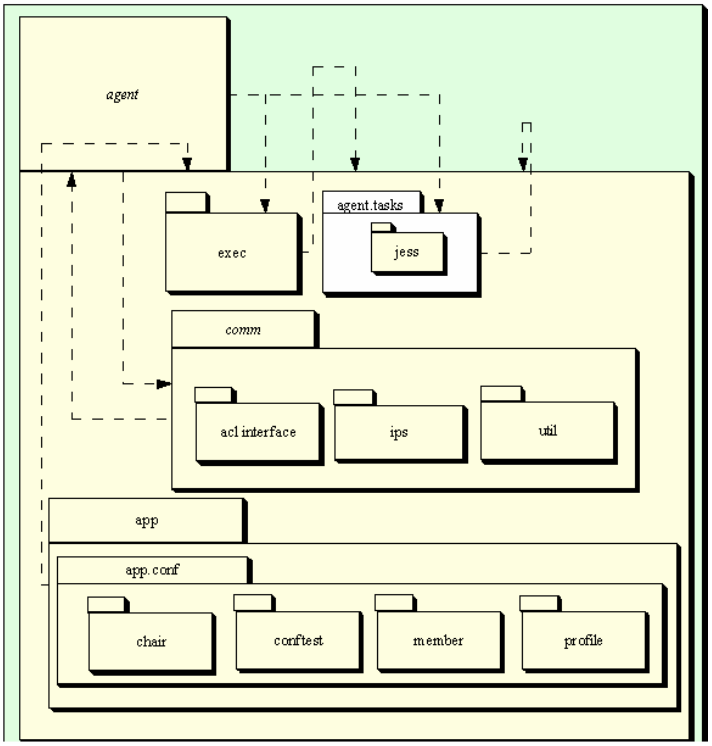

Figure 9.1: Overall Package Distribution of the Agent Architecture

The main sub packages of the Agent architecture are:

9.2.1 Exec

This package contains all the classes and interfaces that results in the execution controller of the agent.

9.2.2 Agent.tasks

This package contains all the classes and interfaces which define the Task API of the agent. This package contains a sub package jess that encapsulates the theoretical reasoning controller for SAGE agents. All the modules responsible for jess integration are implemented with in this sub package.

9.2.3 comm

This package contains all the sub packages that formulate the communication Controller for SAGE agents. The sub packages include the ips sub-package which contains the skeletons of the interaction protocols, the ACL interface sub package which allows the interfacing of agents with ACL module of the SAGE system Framework and the util package which provides general purpose utilities required o Agent conversation management for example the Sender and Receiver Task Units which are responsible for sending and receiving messages respectively.

9.3 OVERALL ARCHITECTURE ANALYSIS

No discrete parameters have been specified for evaluation of agent architectures [12]. For the validation and testing of the characteristics and features of the architecture a high level agent application was developed incorporating all the features of the architecture. The aim of this application was not only to test the different modules of the Agent architecture but also to depict the ease with which the architecture allows the agent to interface with the latest technologies like web services using the Agent-Web Gateway and the Grid [36, 37].

9.4 SYNERGY OF AGENTS, GRID AND WEB SERVICES

Grid computing has two aspects that make it differ from older meta-computing and distributing computing efforts. One is the scale of the data handled. Other is the use of computing sources and data sources that are not controlled by the user or his organization. To achieve both, a dynamic way to define and use a computer or data service is required. This is the goal of the

OGSA effort. Similarly, data and resources should be defined in a way that is understandable and usable by the target user community. This is the goal of "ontologies", part of the Semantic Web effort.

Our vision of the integration of agents with Web services and Grid computing is to lay foundation for a self-regulating system, for e-business realization as shown in Figure 9.2

In Autonomous Semantic Grid, Web services and Grid (OGSA) will provide an open system for dynamic resource sharing and agents will be the provider or consumer of resources while acting as proxy for humans (autonomy). Ontologies will bridge the gap between agents and Grid by bringing semantics into Grid. The whole system will build upon sheer trust among the entities i.e. (Service Providers and Service consumers).

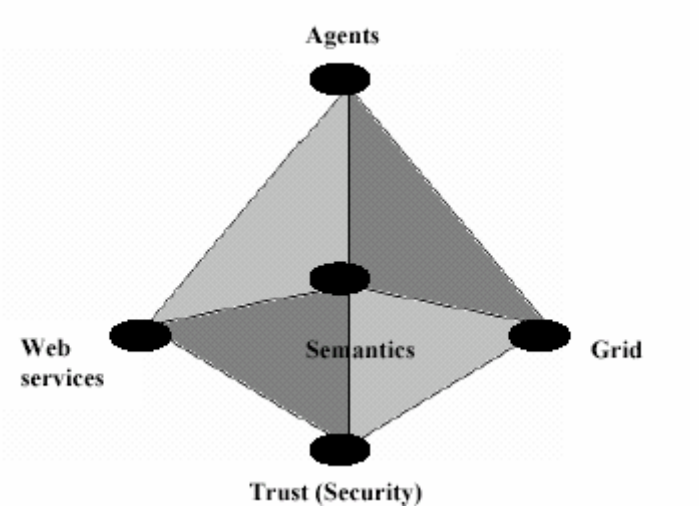

Figure 9.2: Synergy of Technologies

9.4.1 Conference Planner Application

Realizing the suitability of Agents for planning activities we have designed a Conference Planner application that facilitates the planning a conference .The aim is to deign such an application that should allow agents of various conference members to plan out a conference by

negotiating with each other. The roles of the Agents defined for the application are: (i)Conference Chair Agent and (ii) Member Chair Agents

9.4.2 Conference Chair Agent

Conference Chair Agent belongs to the Conference chair person, who wants to plan out a conference. To work out the topics this agent selects the members from the active member list (any number of members) and sends them a request to suggest topics for the conference. Resulting in a one to many negotiation scenario .Conference Chair Agent when receives reply from the members it compiles the results and generates the final list of topics. Along with the selection of the topics ,the agent plans out the time and place for the conference along with topics by searching a huge database containing data of the past conferences. This search is carried out by utilizing the Grid services.

9.4.3 Conference Member Agents

These Agents when receive the requests from the Conference Chair Agent ,select some random topics which take form of their preference list. These Agents then search the web through the web services, to get a look at related articles and papers. These agents posses the capability to filter out the related articles according to their preferences. These Agents then send their list of topics along with related articles titles to the Conference Chair Agent.

9.4.4 Negotiation Scenario

The theme of the application was to implement one to many negotiation scenarios amongst agents and to demonstrate the high level intelligent interaction amongst agents. A conference planner system was created based on the negotiation scenario depicted in Figure 9.3. Two main Agent Roles were described – One agent in the Conference planner is delegated as the Conference Chair and one (or more) as the Conference Member Agent as described previously.

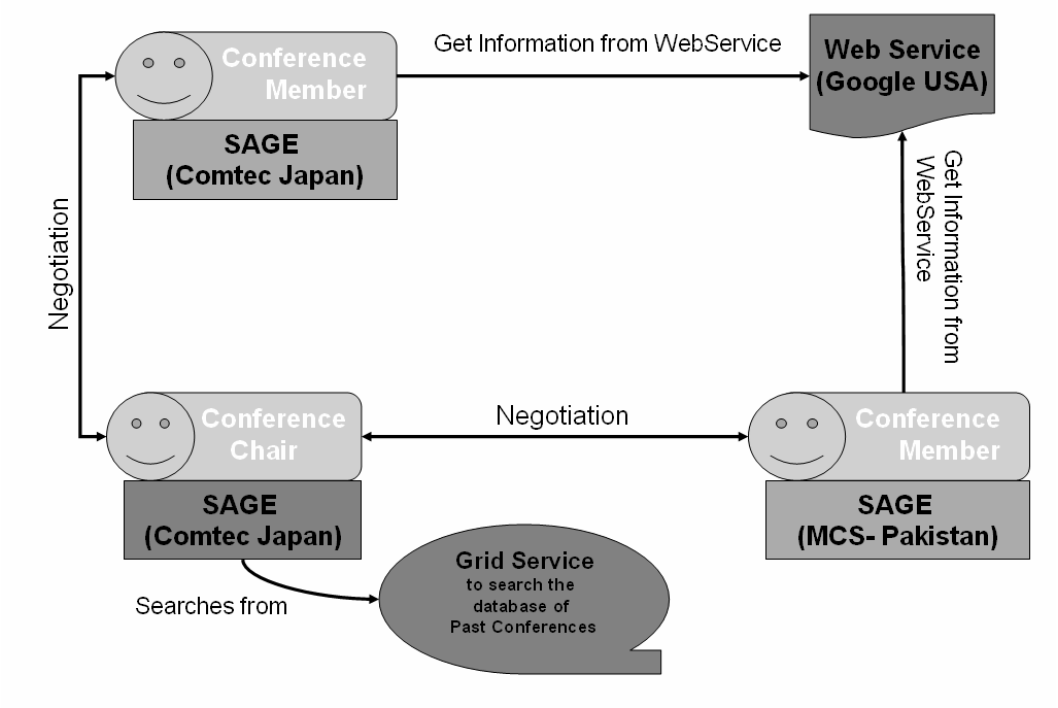

Figure 9.3: Overall Agent Collaboration Scenario

Initially the application was implemented with FIPA Request as the underlying protocol as shown in Figure 9.4. With this Protocol as the underlying protocol the negotiation scenario was limited as the Initiator Agent, the Conference Chair agent could only send the message once. For this reason the first step of the negotiation scenario was that agent sends request of participation to any number of member agents .The member agents interpret the request. Then decide to agree or refuse. If agree is the decision of the member agent then the agent sends the information searched from the web to the chair agent. Chair agent compiles the list on its own side along with other information for the CFP.

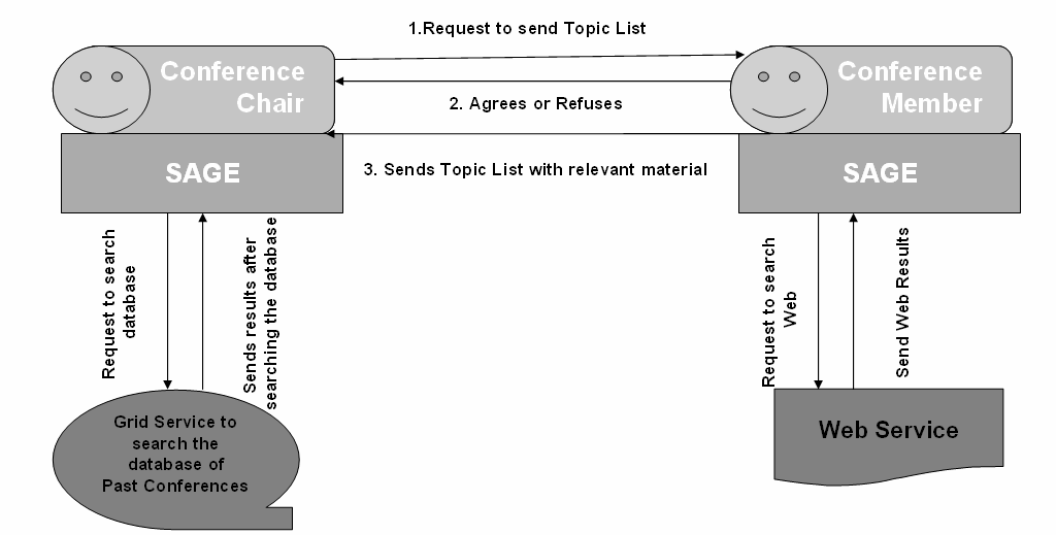

Figure 9.4: Negotiation Scenario : FIPA Request

After the incorporation of Contract Net in SAGE the negotiation Scenario was updated to Contract Net Protocol as shown in Figure 9.5. The negotiation scenario was now updated to this protocol and now the conversation was two phase. The Chair agent now sends the call for proposal to all the member agents in the active member list. The member agent interprets the CFP sent by the Chair Agent in the light of its own preferences and decides whether to send the proposal or not. If it agrees then it send the preferred topic list. The Chair Agent when receives the list from all the members it compiles the list in the light of its own preferences and facts in its belief base. The decisions are taken by utilizing the inference mechanism of JESS inference engine. After the list compilation this time the unlike the FIPA Request protocol the Chair agent sends the compiled topic list. While the chair agent gets past conferences information the member agents search the web via web services to get information on the selected topics. After the information is send the Chair agents compiles the final CFP for the conference.

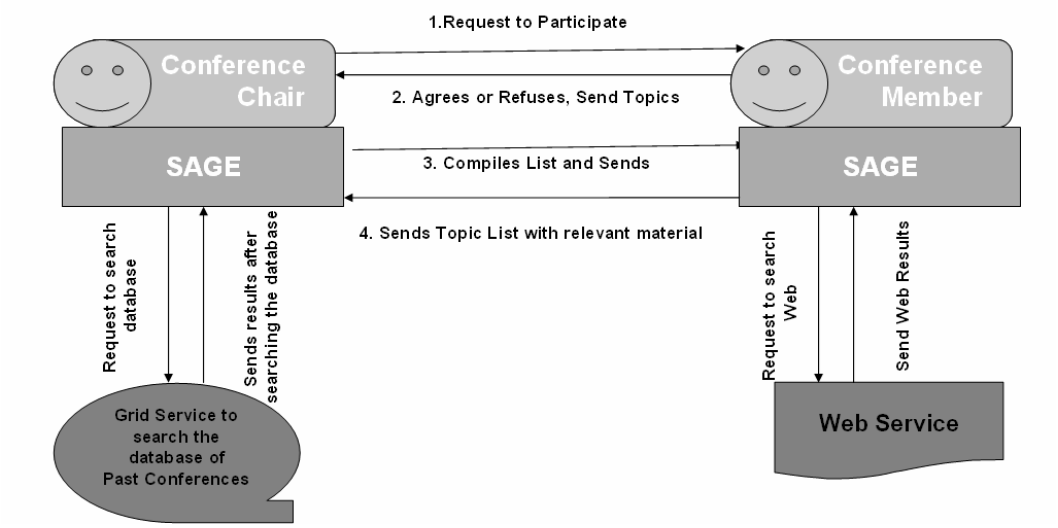

Figure 9.5: Negotiation Scenario : FIPA Contract Net

1: N Negotiation scenario was implemented. The chair agent is responsible for maintaining negotiation with many member agents at the same time. The concurrent negotiations are managed by the execution controller. The protocols are modeled within Finite state machine and each state in turn consists of various TaskUnits modeled to execute various agent actions. The application not only validates the architecture but also illustrates the coordination of different entities by agents acting as owners in heterogeneous and dynamically changing environments. To add more dynamicity to the application the integration with BDI mentalistic concepts is still underway.

9.4.5 Conference Planner Application Design

There are two main interfaces defined for the application: One for the Conference Chair and the other for the Conference Members.

9.4.5.1 Chair Agent Features

The Chair Agent Interface has significant features including: (i) Allows Members to be dynamically added. (ii) CFP to be dynamically viewed and Modified (iii) Complete status information of the Negotiation, (iv) Search from the Grid about the past conferences.

The Conference Chair agent maintains member profiles of all the member agents. The member profile contains member Name, Status, Topic List and Interests ,Academic Specialties and the information they send via web.

9.4.5.2 Features of Conference Members

The features of Conference Member include: (i)Interface for receiving the conference CFP, (ii) Performing search from the desired web service given the conference key interest area, (iii) Filtering of the results based on Keywords, (iv) Dynamic addition and deletion of keywords (v)Dynamic status of the Negotiation with the Conference Chair.

9.4.6 Highlights of the Application

The high light of the application is that it allows the agent to interact with each other utilizing all the aspects of agent architecture. The application clearly depicts that the agent architecture makes it easy for the agents to interact with high level technologies like Web Services and Grid. Also the application allows for dynamic Task Delegation to the Agent upon occurrence of various events. Also Dynamic status is maintained for the Negotiation Scenario.

9.4.7 Negotiation Protocol Design and Implementation

The customization of the 3-level protocol design for the application can be seen in detail in Appendix C. The implementation details specific to the application are as follows: The ConferenceChair Agent and the ConferenceMember Agent extend from the main Agent class of the Agent package. Both Agents have interface objects for handling user interactions. The ConferenceChair Agent maintains member profiles for all members using the member profile object. In addition, it allows the user to view member profiles using view member profile object. The object relationships are depicted in Figure 9.6.

The deliberative role of the conference Chair Agent comes into play when it finalizes and compiles the list after receiving the replies and notifications from all members. For this compilation this agent is dependent on the Theoretical reasoning controller. It keeps track of the information received from the members in the form of facts.

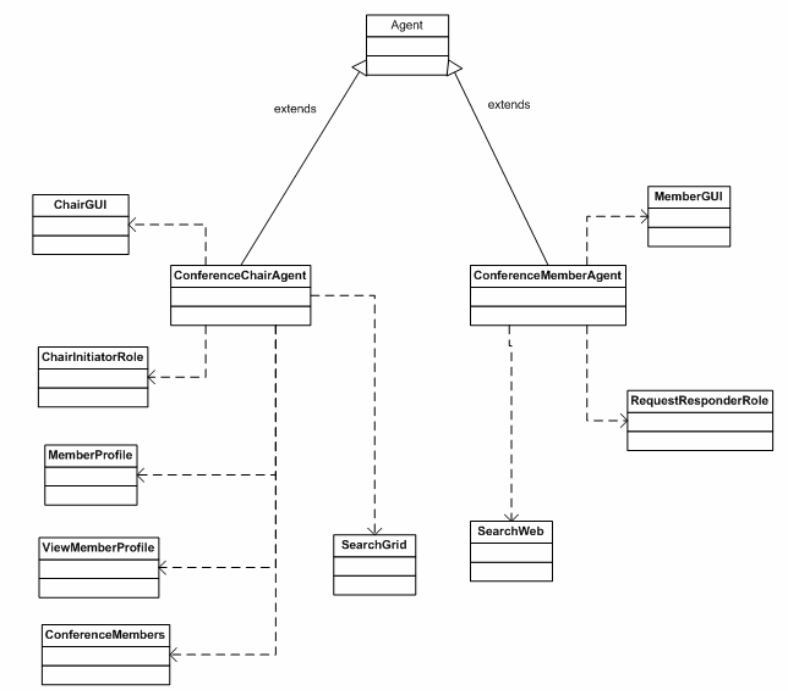

Figure 9.6: Class Diagram for Conference Planner Application

9.5 ANALYSIS AND EVALUATION

9.5.1 Evaluation scenario

The performance analysis for the conference planner application was carried out in a comprehensive and systematic manner. Initially the evaluation of the web, Grid and the Agents were done individually and then an overall evaluation scenario was created. The purpose for this two phase testing was to realize the extent to which these features are contributing in the performance of the application. In addition to that the integrated testing was carried out to realize performance of the application with all the features integrated together.

9.5.2 Results for Web Services

Network delay analysis among distributed services was carried out between Comtec, Japan and Google USA. The results are shown in Figure 9.7 where Average delay = 0.95373.

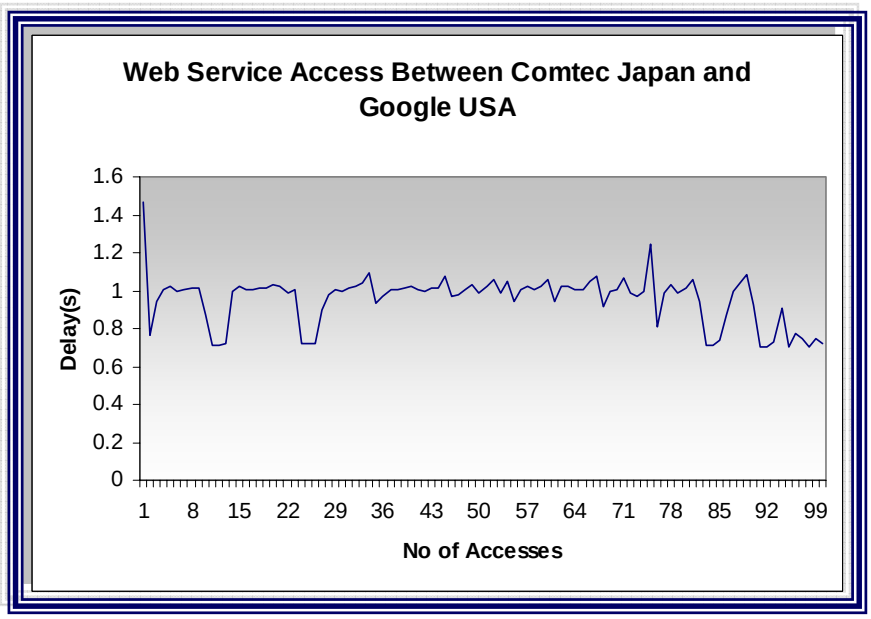

Figure 9.7: Network Delay Between Comtec Japan and Google USA

Network delay analysis among distributed services was carried out between NUST, Pakistan and Google USA. The results are shown in Figure 9.8 where Average delay = 3.68548.

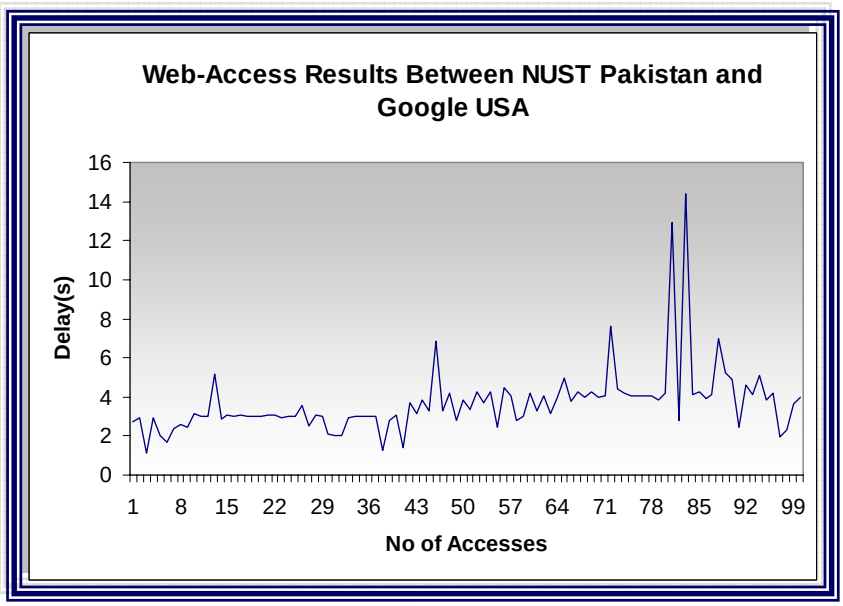

Figure 9.8: Between NUST Pakistan and Google USA

9.5.3 Analysis of Web Services Results

The results for the network delay clearly depicts that the application is bandwidth dependent. As the average delay when agents deployed at Comtec, Japan access the Google, USA is less then one second where as the average delay when the agents deployed at NUST, Pakistan access the Google, USA is more than three seconds. This significant difference in the delay is occurring due to different bandwidths available at Comtec and NUST and variation in the network congestion over time.

Thus the results are dependent on the Network Congestion which varies non-linearly with time. The results thus clearly depict that the performance of the agent accessing the web-service is bandwidth dependent and has an impact on the over all performance of the application.

9.5.4 Results for Grid Service

The Grid services (the Grid node) was deployed at NUST and were accessed from Japan. The results are shown in Figure 9.9 where Average delay = 0.92795.

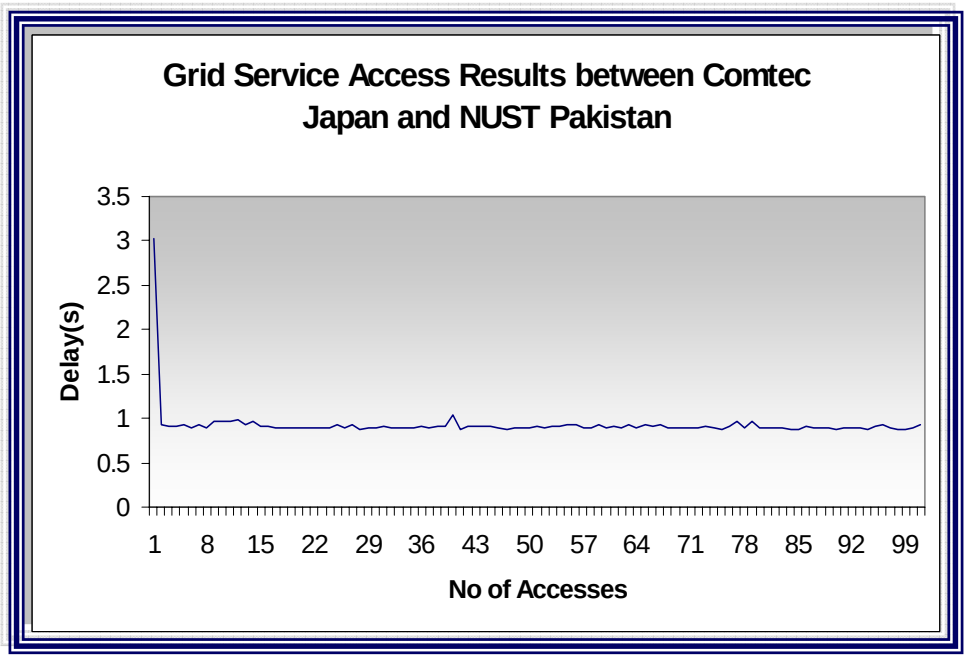

Figure 9.9: Between Comtec Japan and NUST Pakistan

9.5.5 Performance Testing Of Overall Application

The results of the individual testing of main features of the application clearly depicted that performance of the application is dependent on the network delays associated with accessing the Grid and Web Service. In addition the performance of the overall application is also dependent on the number of agents negotiating, precisely the number of member agents receiving the requests. This can be verified based on the results of the communication protocols described in Section 8.

Over all Application Testing was carried out with all the features integrated. The testing of the application was carried out in a distributed manner, with Chair Agent residing on one PC and the Member Agents residing on the other. The testing environment was controlled to as much extent as possible. The results were taken assuming no Network Congestion through out the testing duration.

9.5.6 Results for Overall Application

The overall results for the testing of the application are depicted in Figure 9.10 and Figure 9.11. The results depict that as the number of member agents are increased the execution time of the application increases linearly. This linear trend is the result of controlled environment testing.

Figure 9.10: Negotiation Time for Conference Planner Application

Since the Internet is an asynchronous distributed system, the amount of Network Congestion varies non-linearly with time therefore in real world applications the access time for web and grid may vary and the linear trend of the application execution will be disturbed.

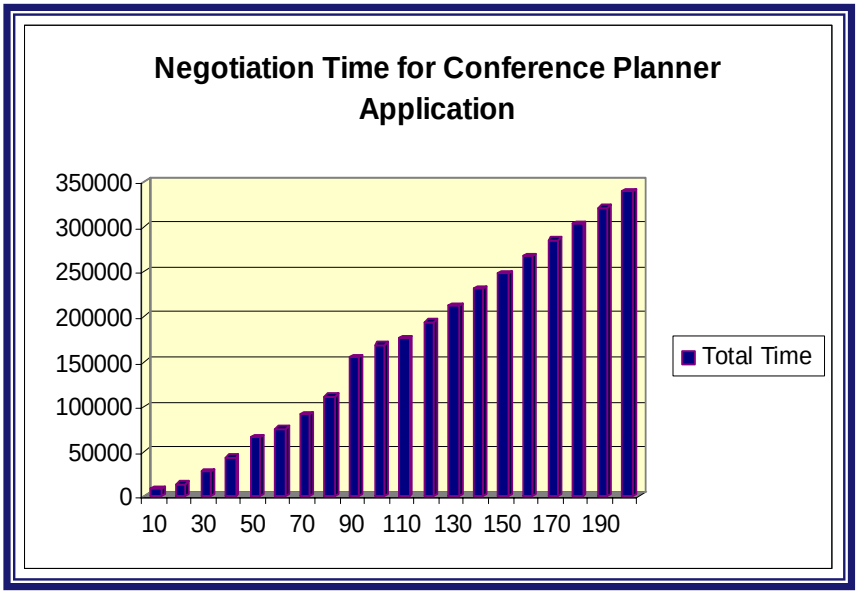

Figure 9.11: Time Analysis for Conference Planner Application

The application execution results also verify the scalability of the underlying system as the number of member agents on one machine may extended to more than hundred agents with out hampering the performance of the application.

Chapter 10

FUTURE WORKS AND CONCLUSIONS

10 FUTURE WORK AND CONCLUSIONS

MAS community have recognized the advantages of an agent-based approach to building deployable solutions in a number of application domains comprising complex, distributed systems. The project targeted some of the key challenges faced when developing autonomic and autonomous entities in the domain of FIPA-compliant Multi-Agent Architectures. By modeling autonomic entities in the form of *SAgents* and SAGE an autonomic Multi-Agent system we have made a reasonably successful attempt at coupling agent architectural concepts with features of autonomic systems.

The novelty in our approach is the two-fold approach towards autonomic computing. Firstly at the system (framework) level, where the MAS figures as a distributed collaborative environment, as a collection of system and application agents where the autonomic behaviour is controlled by the system level agents through a decentralized architecture. For the development of autonomic application agents and autonomic agent based application systems an autonomic agent construction model is proposed, which is the highlight of this paper. It defines an agent structure with well-defined functional components, which contribute towards their autonomic behaviour. Through preliminary development and experimentation within SAGE using the SAgent's Architecture, we have found it to be a valuable model for studying and testing ideas about autonomic systems.

We feel that we have laid a strong enough base for developing the next generation agent systems which Agentlink describes as "truly-open and fully-scalable multi-agent systems, across domains, with agents capable of learning appropriate communications protocols upon entry to a system, and with protocols emerging and evolving through actual agent interactions. This bears strong similarities to IBM's ongoing research project on *autonomic computing* and existing *Semantic Web* objectives."

However we feel that our challenge has only yet begun. While our implemented model shows feasibility on a small scale, there is much work remaining to be done in customizing the model to handle the complexities encountered in real-world applications. Our first and foremost aim shall be to provide a generic learning sub-engine for *SAgents'* which shall aim at providing self-optimization. We then aim to use the model we have described to develop strong market based autonomic applications. The Conference Planner Application developed with the synergy of

Agents, Grid and web-services [35] serves as a strong validation for the possible use of the architecture.

In future work, many of the current features will be customized for domain specific purposes. In addition we would also like to explore the use of utility and resource allocation functions and what general architectural support can be provided in this regard.

Although we have just started down this road, our experience with building other higher-level agents, gives us confidence. Our firm belief is that any truly autonomic system will require one or more agents of the type proposed by us in our model as part of the architecture.

APPENDICES

APPENDIX A – ALGORITHMIC DESIGN OF TASK-API

Algorithmic Design of Derived Task Units

For Derived Task Units the execution and state change notification algorithms were designed which are depicted in the flowcharts. For Derived Task Units these algorithms had to be designed explicitly because in case of these Task Units the execution of subtask Units is inter dependent. The general Execution Pattern of Derived Task Unit is depicted in the flow chart.

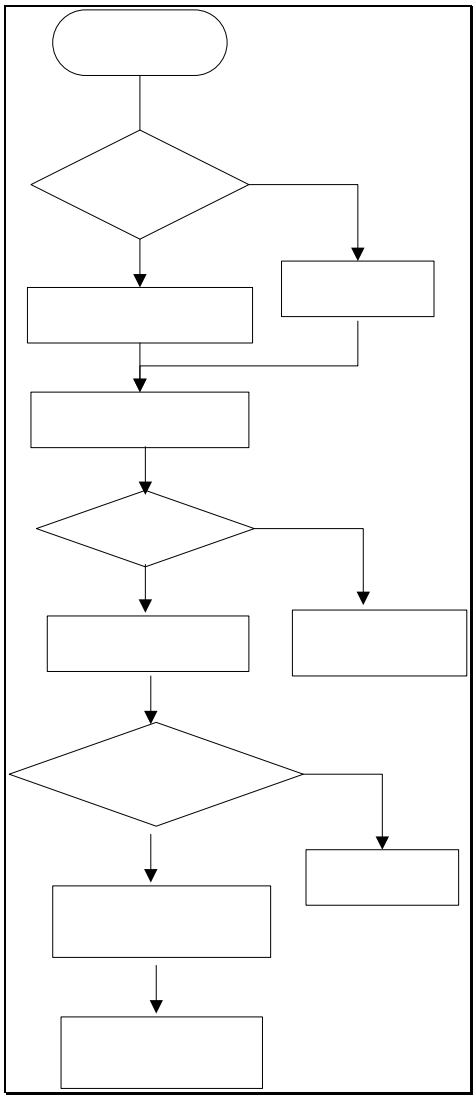

Figure 1: Execution Pattern for Derived Task Unit

Start

IF TaskUnit execution Begining

Yes

Schedule first sub task unit

Get the current child

The state notification handling in case of Ordered and Concurrent Task Units is different from each other because of their different execution patterns. In case of Ordered Task Unit the subtask units are executed sequentially so the state change notification made by the parent is required by its currently executing child only. Where as in case of Concurrent Task Units the notification of any change in the Parent Task Unit is required by all the children of the Task Unit, as all of them are executing concurrently. The design of State change handling in both Ordered and Concurrent Task Units is depicted in form of flow charts.

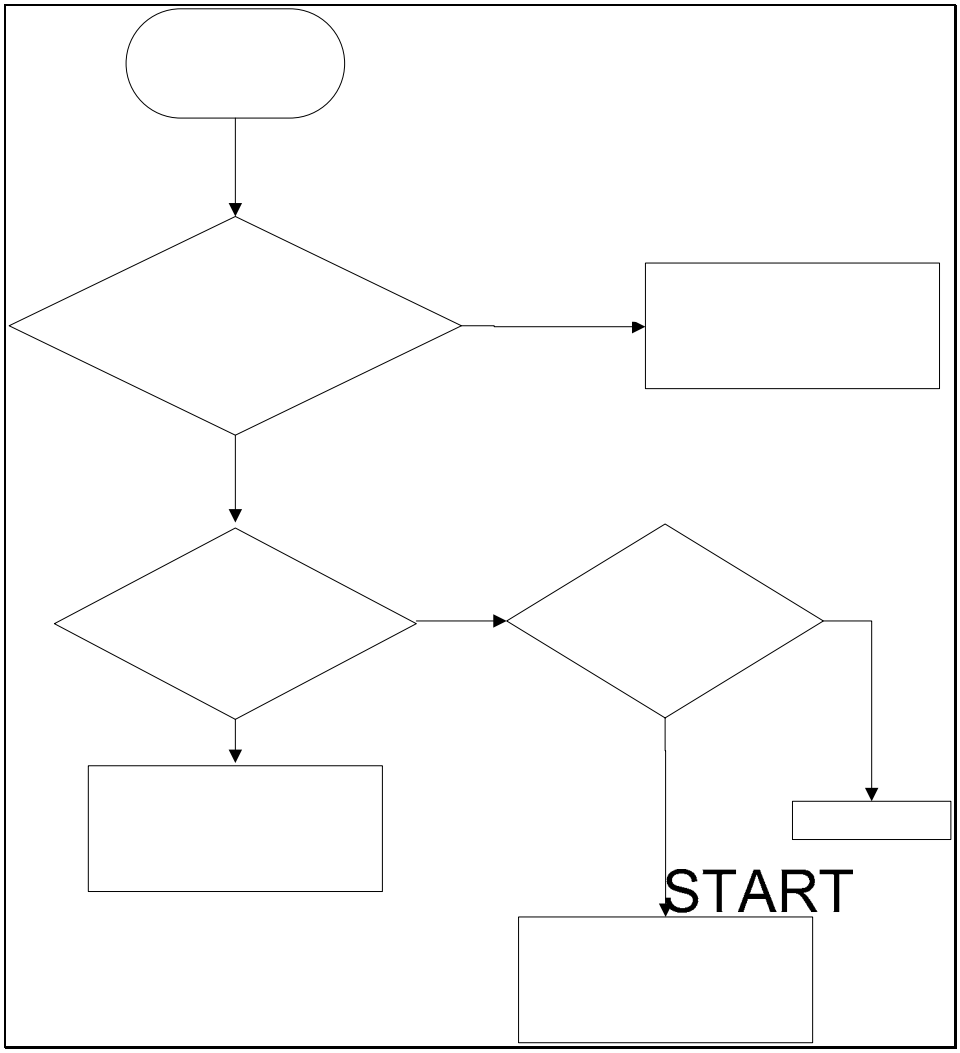

Figure 2: Execution Pattern for Ordered Task Unit

If upwards notification

Child N

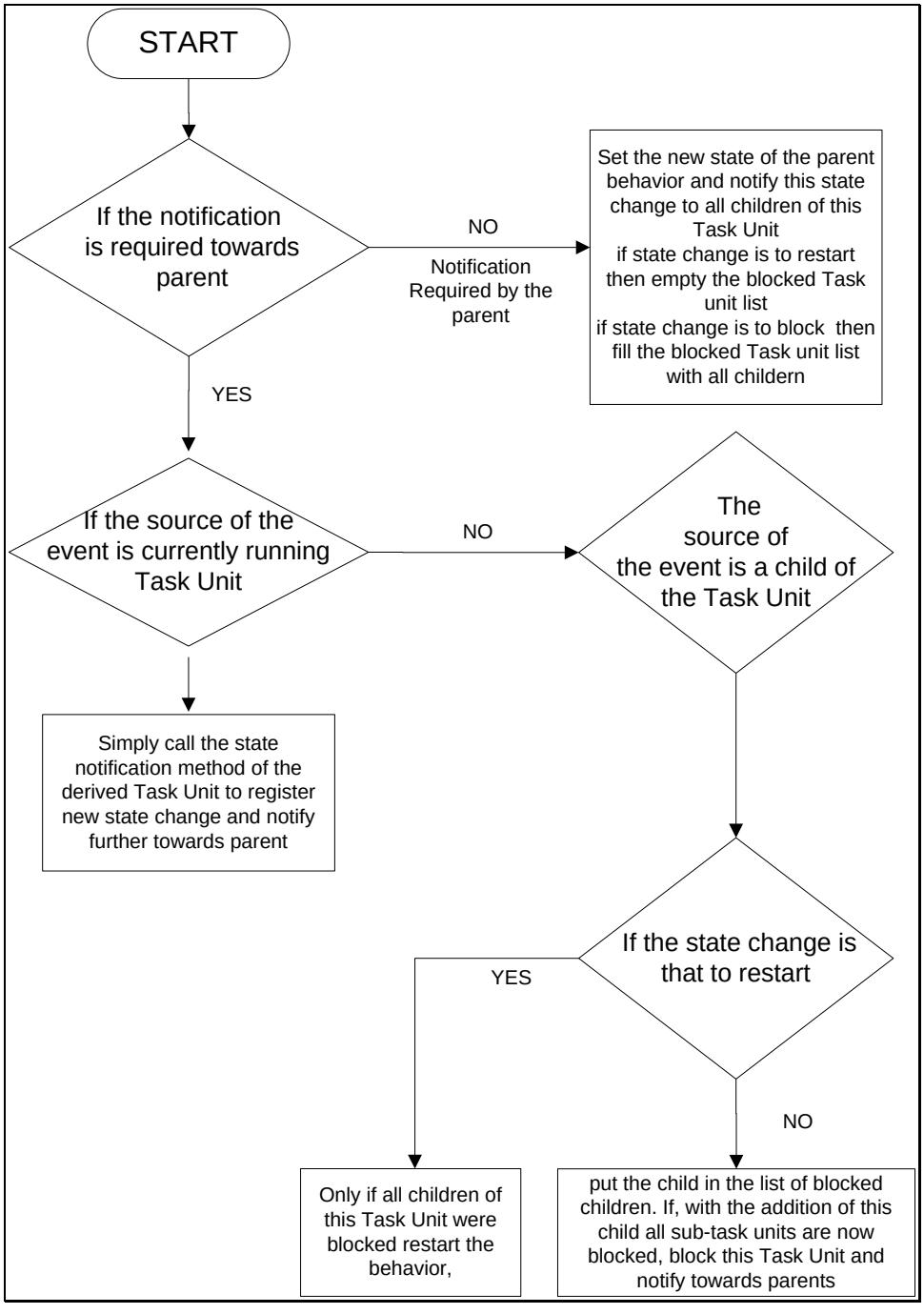

Figure 3: Execution Pattern for Concurrent Task Unit

Design and Implementation Model of FSM

Design of FSM Task Unit

- Derived Taskunit with Finite State Machine based children scheduling.
- It is a DerivedTaskunit that executes its children Taskunits according to a FSM defined by the user.
- More specifically each child Task represents a state in the FSM.
- The class provides methods to register states (sub-Taskunits) and transitions that defines how sub-Taskunits will be scheduled.
- FSM provides the minimal state based behaviour support for the agent.

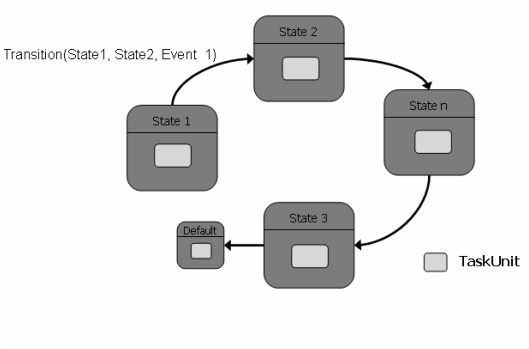

Figure 4: Finite State Machine Model

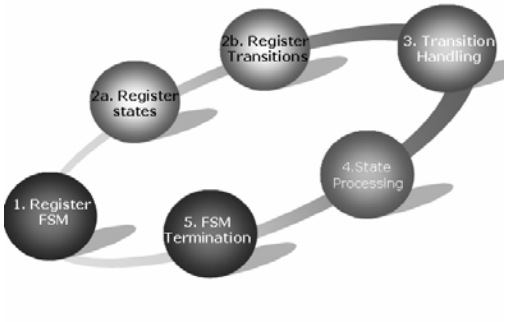

Figure 5: FSM Model

States and Transitions

- States represent processing that goes on internal to the agent. This processing is denoted by a sequence of actions.
- Transitions denote communication between agents or between tasks.

Steps for registering a FSM

At a minimum, the following steps are needed in order to properly define a FSMTaskunit

- Register a single Taskunit as the initial state of the FSM by calling the method registerFirstState()
- Register one or more Taskunits as the final states of the FSM by calling the method registerLastState()
- Register one or more Taskunits as the intermediate states of the FSM by calling the method registerState()
- For each state of the FSM, register the transitions to the other states by calling the method registerTransition()

Types of States

- First and Last States
- Intermediate states
- Register a Taskunit as the initial state of this FSMTaskunit
- Register a Taskunit as a final state of FSMTaskunit .
  - When the FSM reaches this state the registered Taskunit will be executed and, when completed, the FSMTaskunit will terminate too.

Types of Transitions

- **GeneralTransition on an event**
  - Register/Associate an event that triggers transition from a source to destination.

- **SpecialTransition**
  - May need to reset certain states.

  - Resetting States means that the taskunits encapsulated within these states must be reexecuted or reset.
  - This is particularly useful for transitions that lead to states that have already been visited.

- **GeneralDefault Transition**
  - This transition will be fired when state terminates with an event that is not explicitly associated to any transition.

- **SpecialDefaultTransition**
  - To be used when it is needed to reset certain states.
  - This is particularly useful for transitions that lead to states that have already been visited.

Events that trigger Transitions

- An event called Termination event is associated with each state.
- The termination event determines which transition is fired
- Events can be modeled/represented differently
- Kept simple at this stage
- Modeled in the form of a simple datastructure that is returned upon completion of the state.

Forcing Transitions on states at runtime

- **ForcedTransition**
  - Temporarily disregards the FSM structure, and jumps to the given state.
  - Similar to a unconditional branch or GOTO statement between states, and replaces the currently active state without considering the trigger event or whether a transition was registered. It should be used only to handle exceptional conditions, if default transitions are not effective enough.

State Scheduling

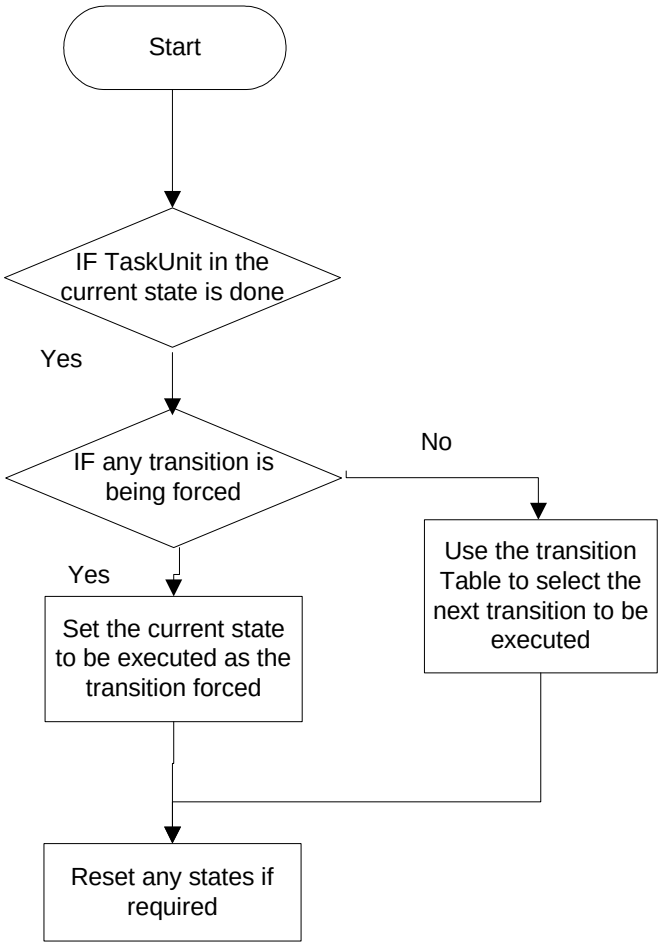

Figure 6: Execution Pattern for FSMTask Unit

Transition Handling

- Transition DataStructure
- Information
  - Source FSM TaskUnit
  - TaskUnit
  - Source State
  - Destination State
  - Event that triggers this transition

  - Special Properties such as Default, forced etc
- A table for States and their corresponding transitions
- Table for events corresponding to various states and the transitions triggered by those events.
- Transition Table

State Change Notification

State Changing Behaviour of FSM:

- FSMTaskUnit may be blocked or restarted
- The corresponding states must also be notified of such events accordingly.
- The notification process is similar to the one for DerivedTaskUnit
- The FSMTask Unit is considered as the parent and the taskunits in its states as its children.

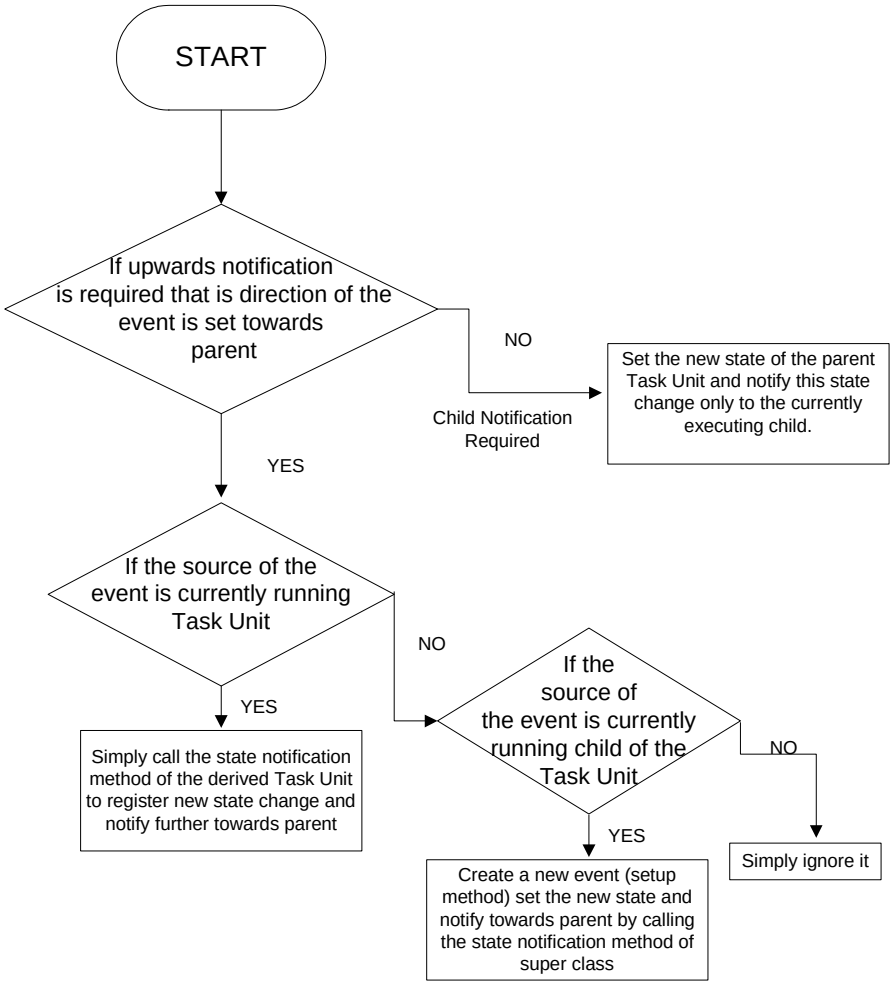

Figure 7: State Change Notification for FSM

Overall Implementation Model of FSM

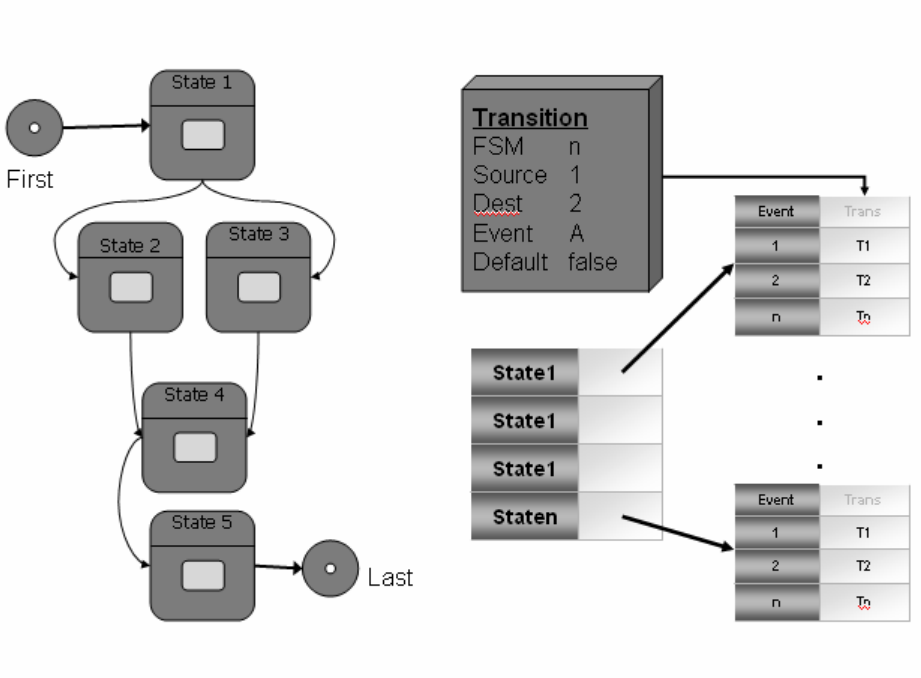

Figure 8: Implementation Model of FSM

Role of FSM

- Enhancing the communication infrastructure for the agents
- Some support for reasoning is needed.
- Need for Rule-Based Behaviour support
- With the integration of Rule-Based support and State Based model, we can then enhance the present communication infrastructure.

Suitability of FSM to Model Agent Conversation

- For modeling a conversation from the point of view an agent sending and receiving messages such an agent behavior is required which supports input and output operations.
- Incase of FSM the transition only occur incase of an input whose output may stimulate the execution of another state by becoming its input, thus the interactions can be modeled by these inputs and outputs.

- Thus FSM supports the modeling of effective agent communication. This behaviors allows to define the sequence of messages

Implementation of the Task API

The implementation of the Task API has been done as a hierarchy of classes.As seen from class diagram in the Figure, the root class Task Unit lies at the top. This is an abstract class that implements the handlers common to all subclasses. Each handler is implemented as a set of methods that define the functionality of that handler. The programmer can put the functionality of specific actions, it wants the agents to perform in the execute() method of the Task Unit. But the programmer is not responsible for controlling the underlying execution of the Task Unit.This is done automatically by the Execution Handler.Also incase of Derived Task Unit the programmer is only responsible for defining the relationships between Task Units but their management is kept transparent to the programmer by using the functionality of the associated parent handler. Each type of Task Unit is implemented as separate class extended from this generic root class.

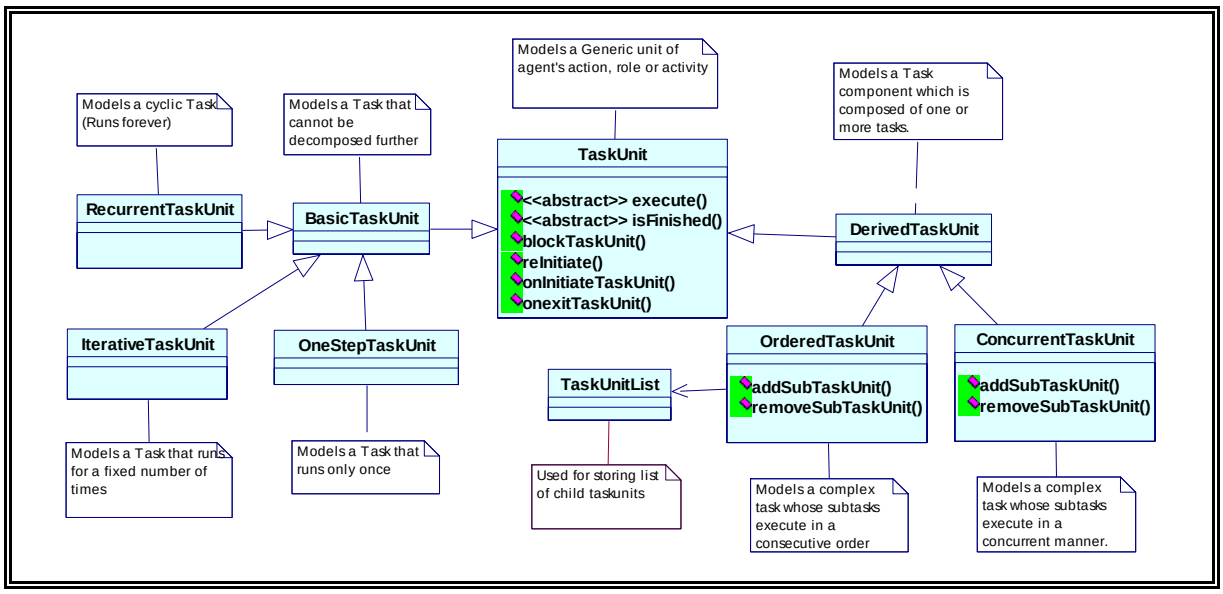

Figure 9: Class Diagram for Task API

APPENDIX B - COMMUNICATION CONTROLLER – 3 LEVEL DESIGN

Layered Protocol Design for SAGE

The 3-level AUML Design was carefully done for the two main FIPA Interaction Protocols namely:

- FIPA Request
- FIPA Contract Net

Layered Design of FIPA Request

Level 1 Deign For FIPA Request

FIPA has specified the level 1 for all the interaction Protocols it has defined. Protocol is defined as a package, a conceptual aggregation of Interaction Sequences of FIPA Request Protocol. In addition, a template specification which specifies unbound entities within the package is given. It needs to be bound when package is instantiated.

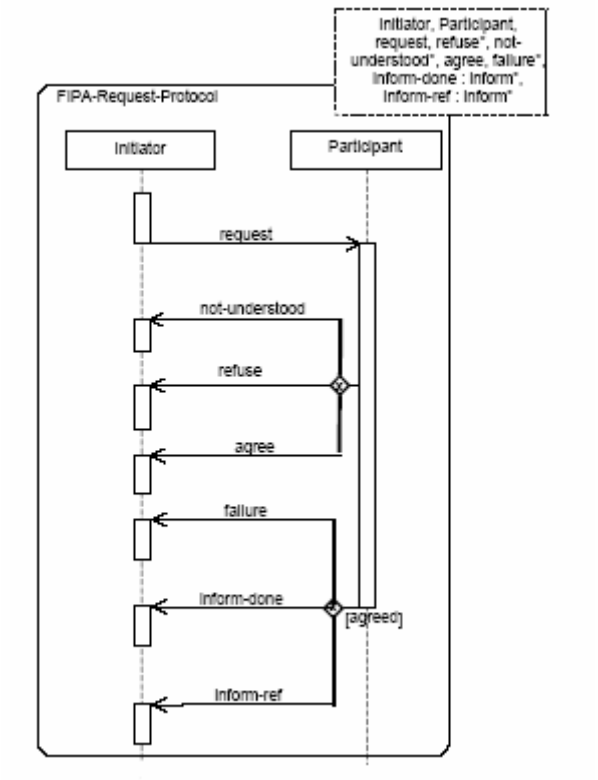

Figure 1: Level 1 Design of FIPA Request

Level 2 Deign For FIPA Request

FIPA has specified two main roles for a protocol. Initiator role which initiates and control the protocol. There can be only one initiator for one session of the protocol. The other role is that of the Responder which responds to the initiator and be more than one in number. The level2 of the protocol design deals with the depiction of interaction between Initiator and responder.

FIPA has specified the level 2 design for all the protocols in form of extended sequence diagram. As described above there are many diagrams possible for describing the interactions among agents. For SAGE we have depicted the level 2 design for interaction protocols inform of State diagrams.

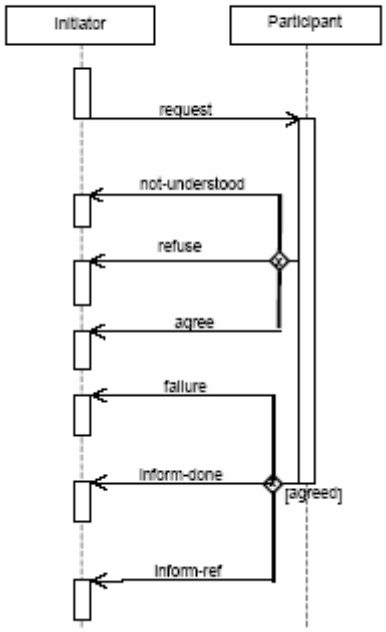

Figure 2: Level 2 Design of FIPA Request

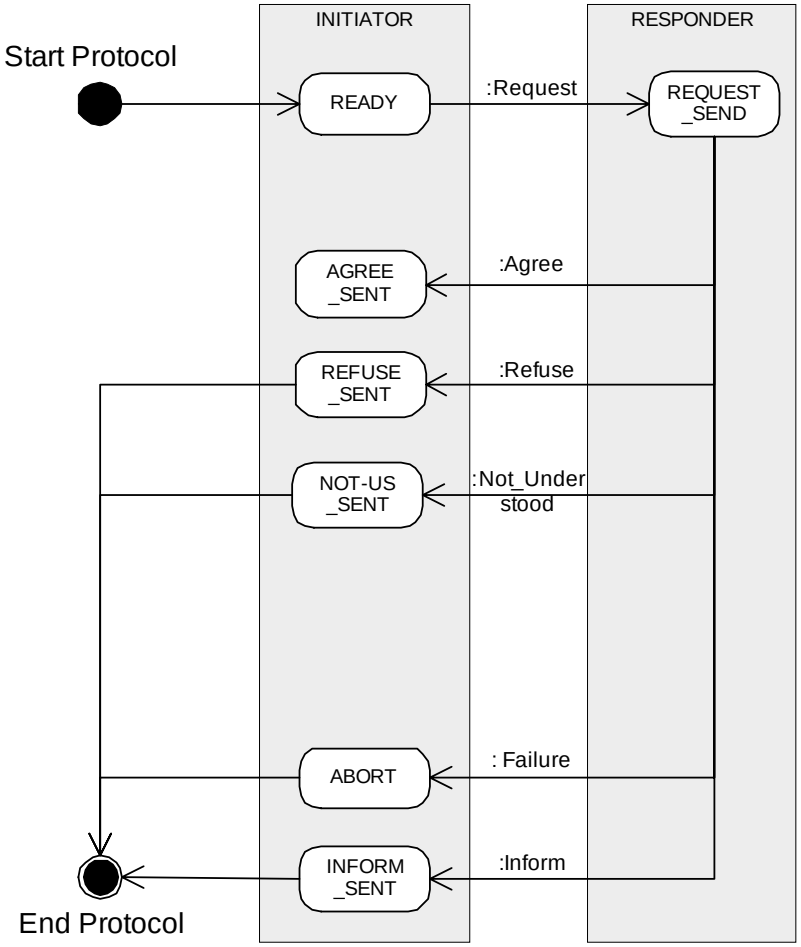

Figure 3: Level 2 Design of FIPA Request: State Machine

Level 3 Deign For FIPA Request

Level 3 design describes the Internal Processing going on within the Request Responder and Request Initiator separately. As SAGE Agents have built in support for presenting the Agent behavior as FSM. For this reason we chose State charts to describe the internal agent processing going on with in the agent.

Level 3 Deign For FIPA Request – Initiator Role

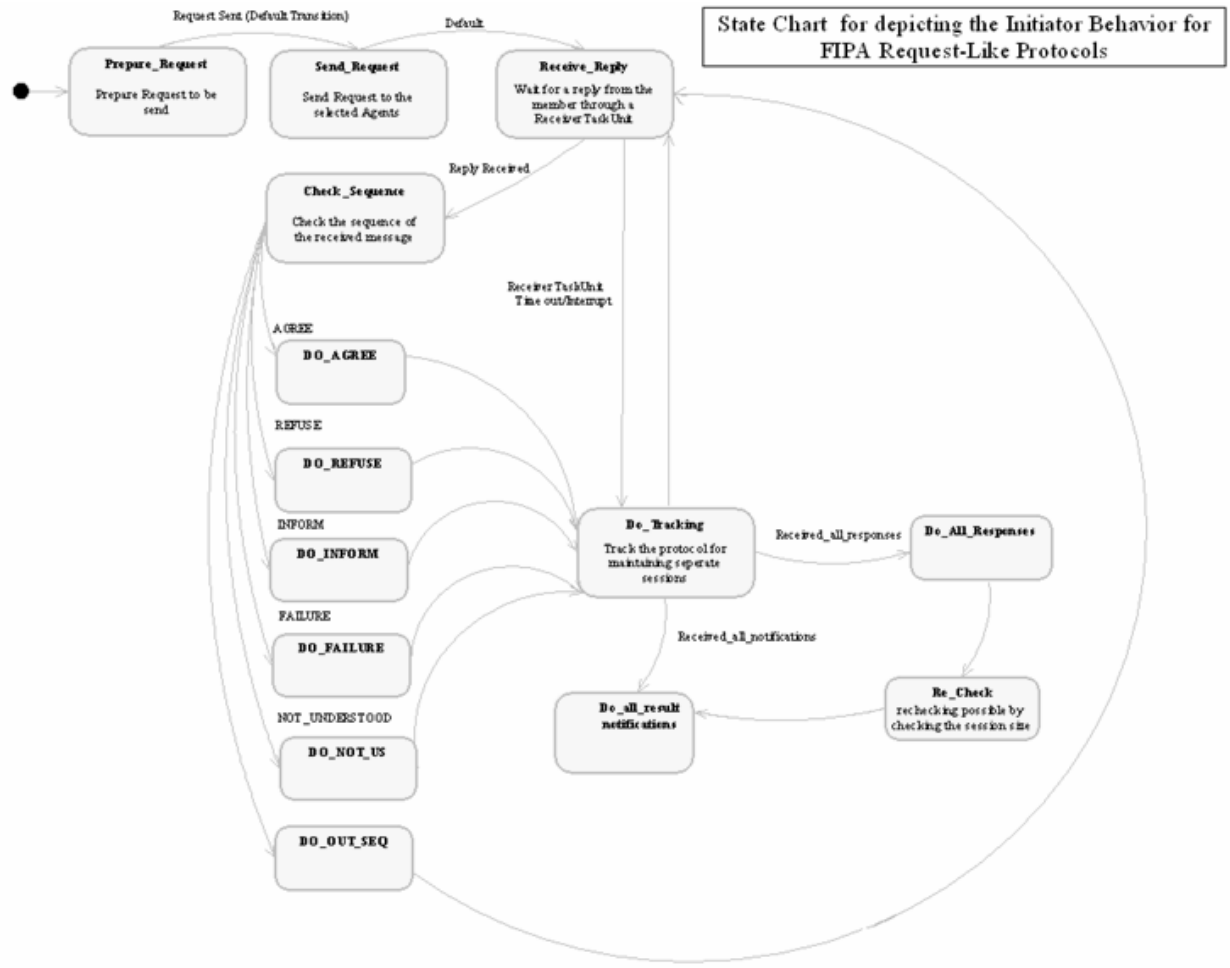

Figure 4: Level 3 Design of FIPA Request: Internal State Processing of Initiator

Level 3 Deign For FIPA Request – Responder Role

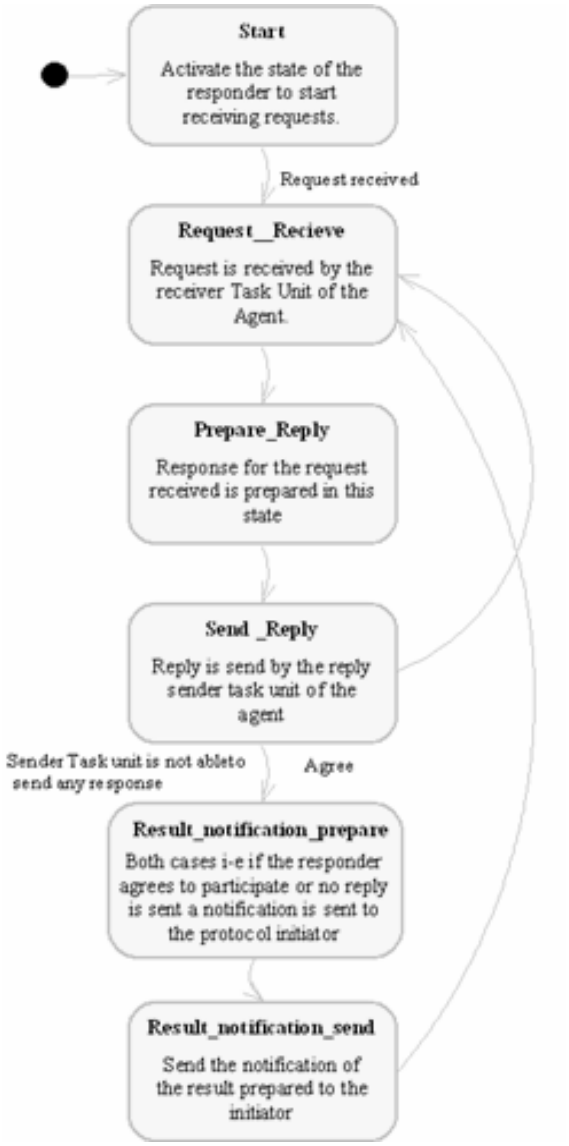

Figure 5: Level 3 Design of FIPA Request: Internal State Processing of Responder

Layered Design of FIPA Contract Net

Level 1 Deign For FIPA Contract Net

FIPA has specified the level 1 for all the interaction Protocols it has defined. Protocol is defined as a package, a conceptual aggregation of Interaction Sequences of FIPA Contract Net Protocol. In addition, a template specification which specifies unbound entities within the package is given. It needs to be bound when package is instantiated.

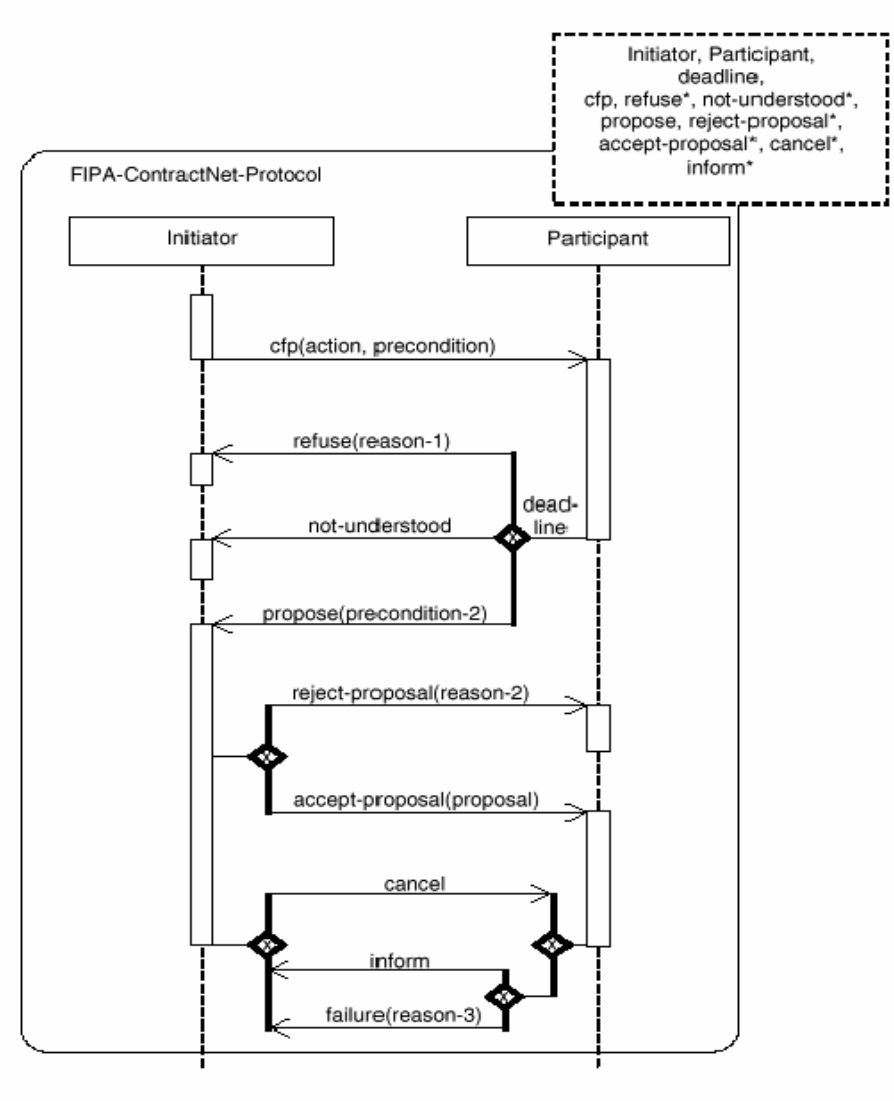

Figure 6: Level 1 Design of FIPA Contract Net

Level 2 Deign For FIPA Contract Net

As FIPA has specified the level 2 design for all the protocols in form of extended sequence diagram. As described above there are many diagrams possible for describing the interactions among agents. For SAGE we have depicted the level 2 design for interaction protocols inform of State diagrams.

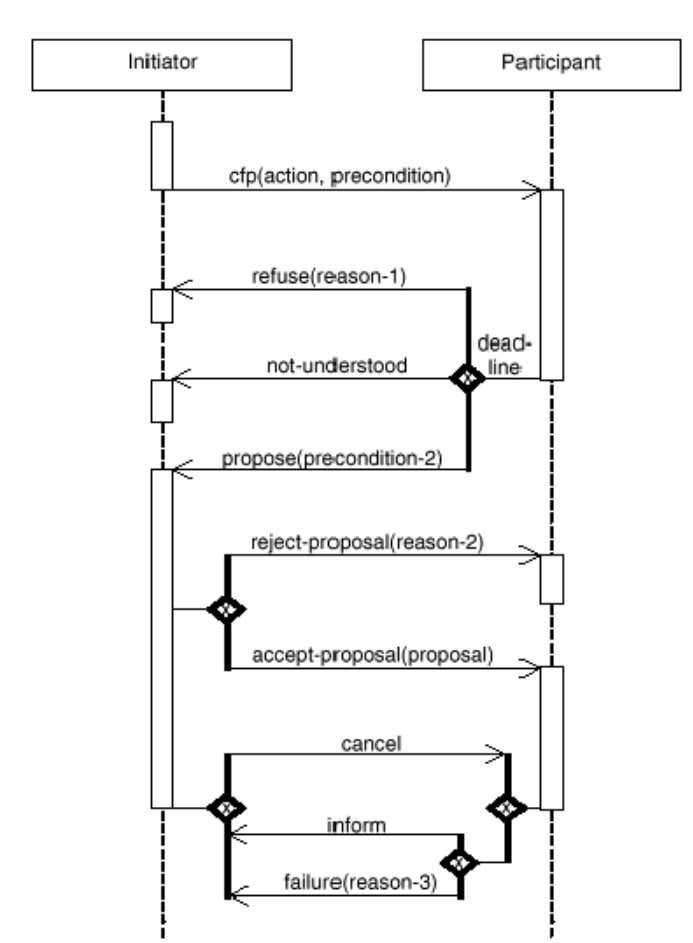

Figure 7: Level 2 Design of FIPA Contract Net: Extended Sequence Diagram

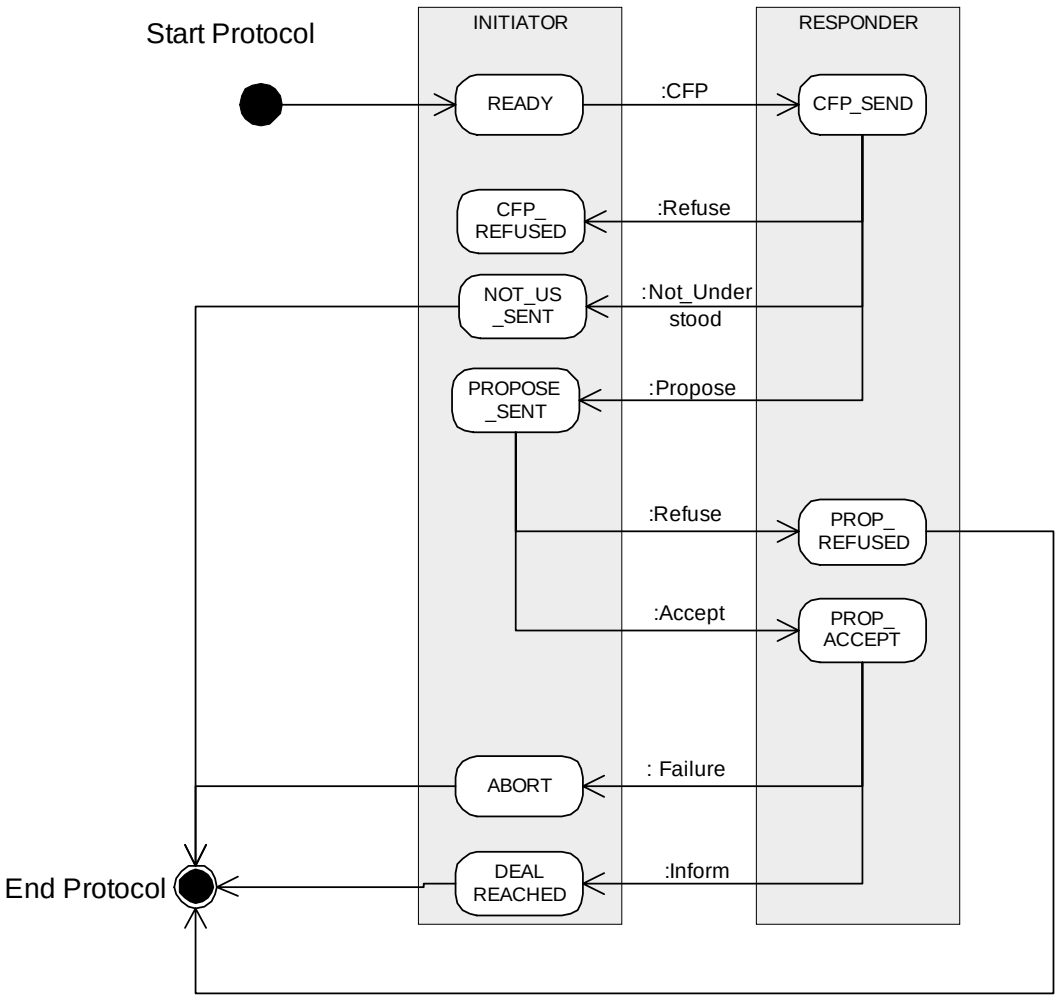

Figure 8: Level 2 Design of FIPA Contract Net: State Machine

Level 3 Deign For FIPA Contact Net

Level 3 Deign For FIPA Contact Net – Initiator Role

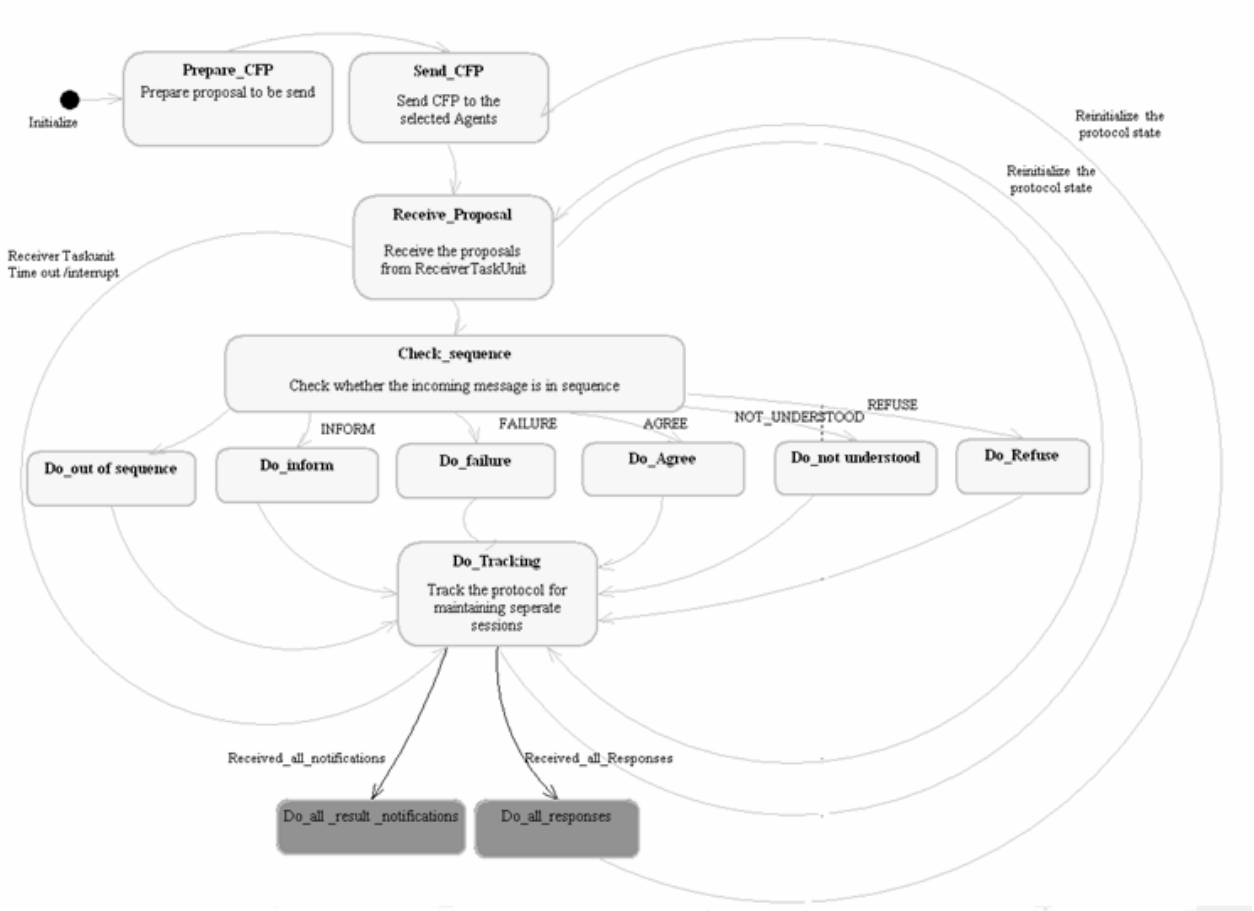

Figure 9: Level 3 Design of FIPA Contract Net: Internal State Processing of Initiator

Level 3 Deign For FIPA Contact Net – Responder Role

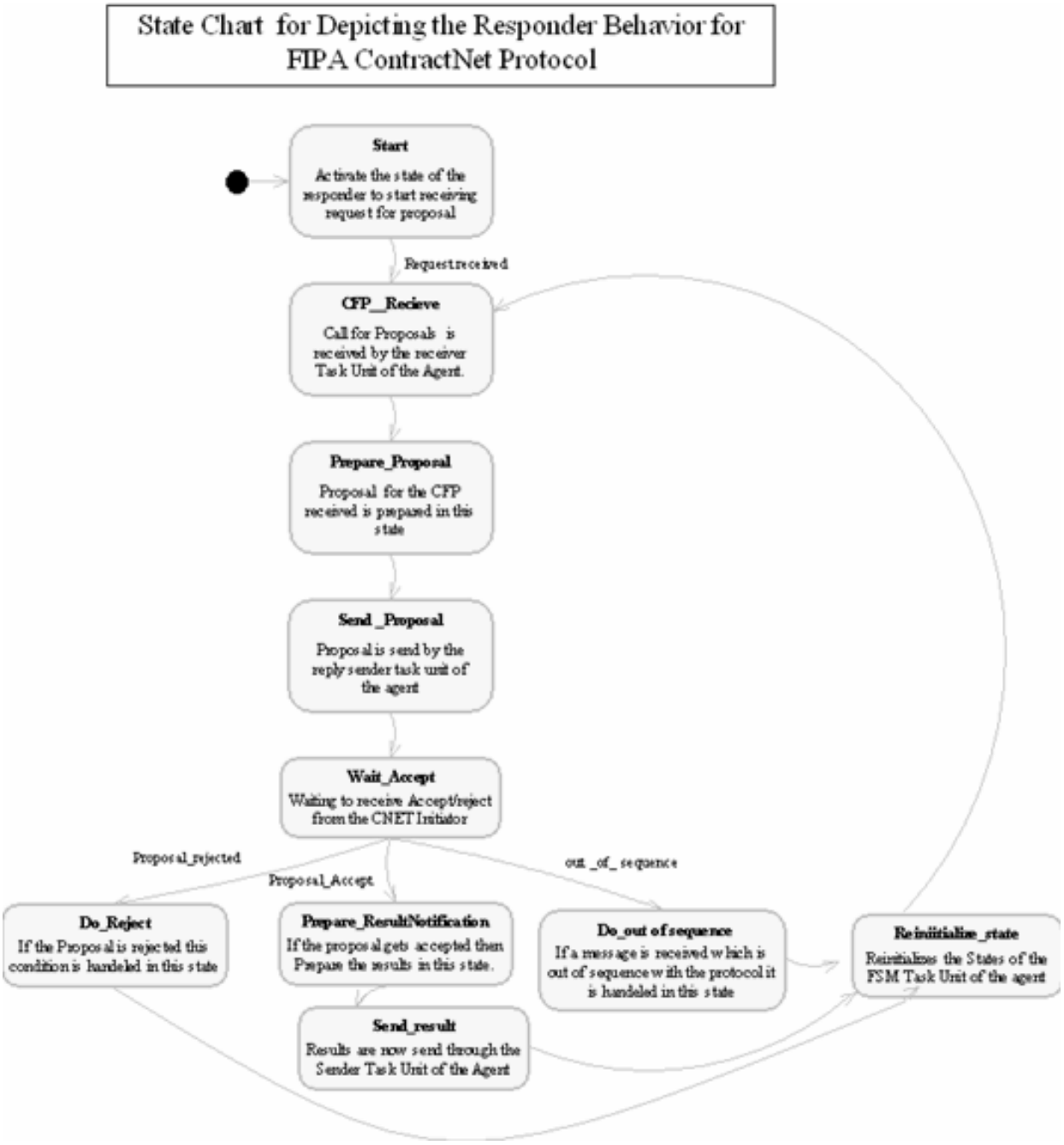

Figure 9: Level 3 Design of FIPA Contract Net: Internal State Processing Responder

APPENDIX C – DESIGN OF CONFERENCE PLANNER APPLICATION

Extended Sequence Diagram For Conference Planner Application

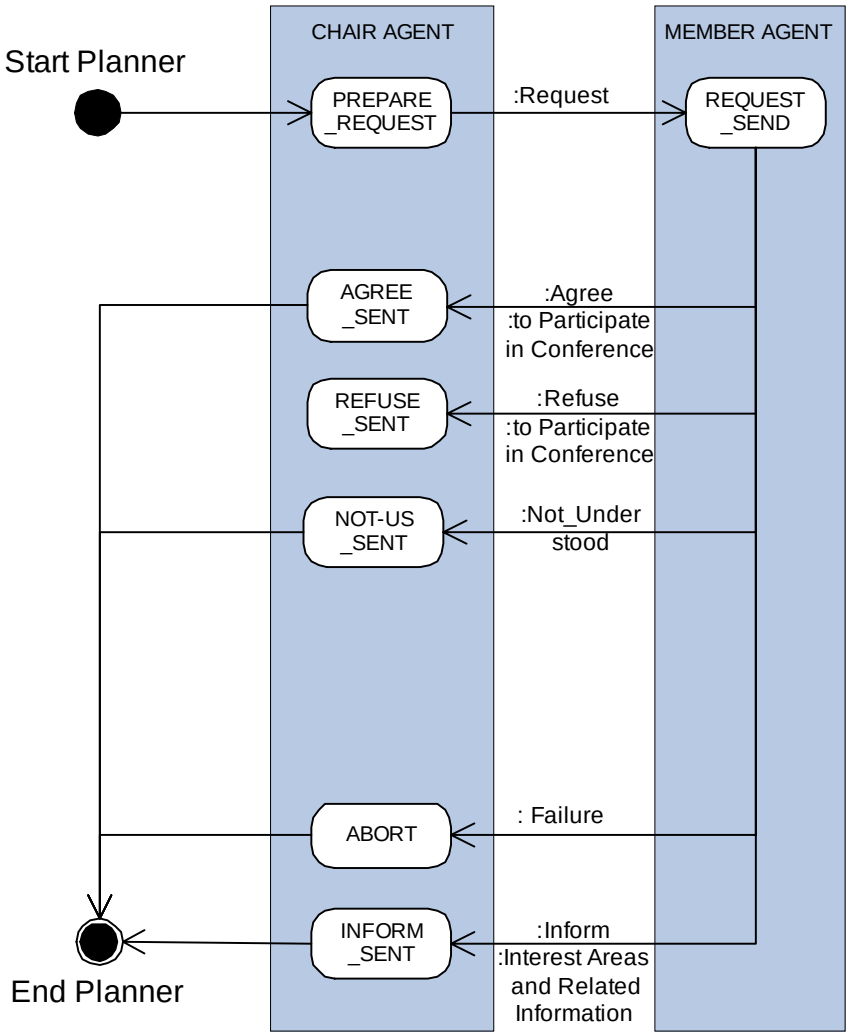

Figure 1: Level 2 Design of Conference Planner

Internal Agent Processing of Member Agent

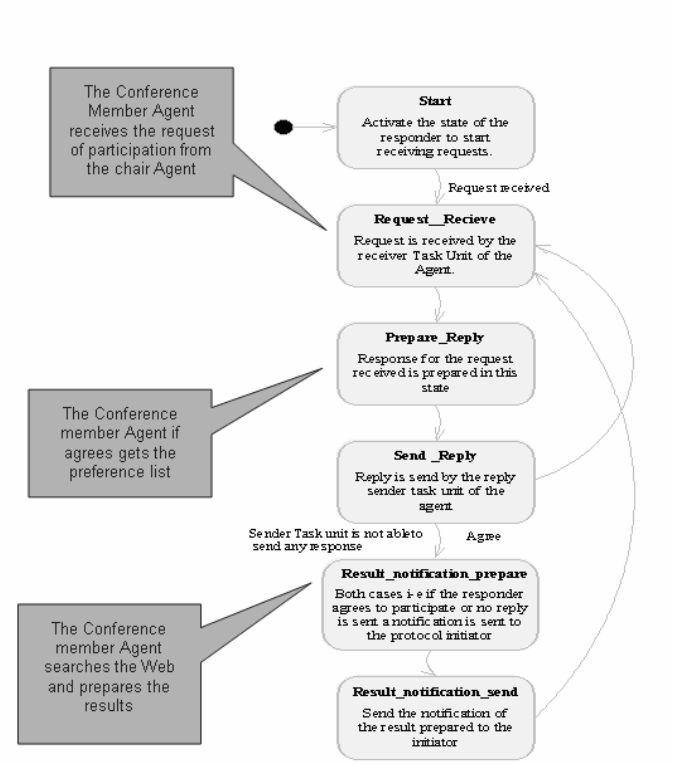

Figure 2: Level 3 Design of Conference Planner: Member

Internal Agent Processing of Chair Agent

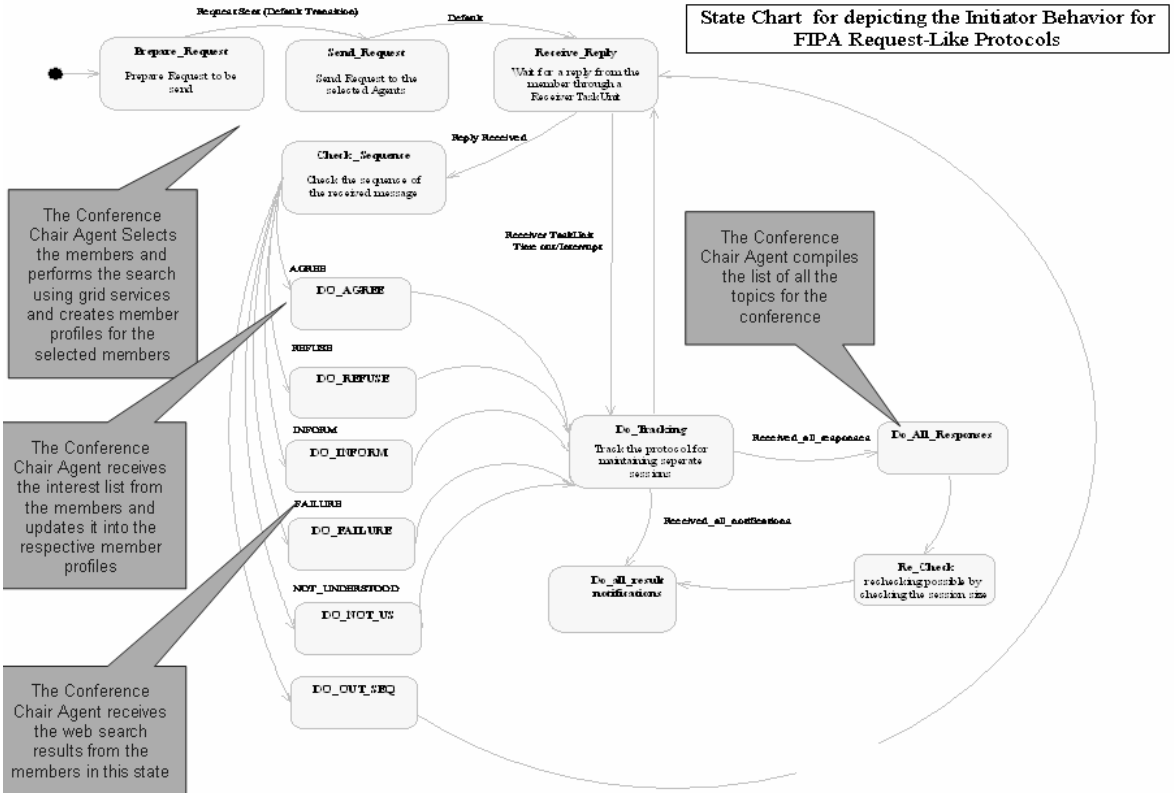

Figure 3: Level 3 Design of Conference Planner: Chair

Bibliography

BIBLIOGRAPHY

BIBLIOGRAPHY

1. N. R. Jennings. On agent-based software engineering. Artificial Intelligence, 117:277–296, 2000.
2. Foundation for Intelligent Physical Agents (FIPA). http://www.fipa.org
3. Abdul Ghafoor, Mujahid ur Rehman, Zaheer Abbas Khan, H. Farooq Ahmad, Arshad Ali, "SAGE: Next Generation Multi-Agent System", PDPTA'04, pp.139-145, Vol. 1, (2004).
4. Zaheer Abbas Khan, H. Farooq Ahmad, Arshad Ali, Hiroki Suguri, "Decentralized Architecture for Fault Tolerant Multi Agent System", ISADS, 04, China, April 5-7, 2005 (2004) (accepted).
5. F. Bellifemine, A. Poggi & G. Rimassi, "JADE: A FIPA-Compliant agent framework", Proc. Practical Applications of Intelligent Agents and Multi-Agents, April 1999, pg 97-108.
6. Poslad, Stefan et al. The FIPS-OS agent Platform: Open Source for Open standards. Nortel Networks-Manchester UK.(2000).
7. H. Nwana, D. Nduma, L. Lee, J. Collis, "ZEUS: a toolkit for building distributed multi-agent systems", in Artificial Intelligence Journal, Vol. 13, No. 1, 1999, pp. 129-186.
8. M. Wooldridge and N. R. Jennings, Intelligent agents: Theory and practice, Knowledge Engineering Review, Vol.10, No.2, 1995
9. E. Mangina, Review of Software Products for Multi-Agent Systems, AgentLink, software report 2002.
10. Wooldridge, Micheal J: An Introduction to Multi-Agent Systems. John Wiley & Sons, 2002.
11. Munindar P. Singh and Michael N. Huhns, Service-Oriented Computing, Wiley, 2005.
12. Steven P. Fonseca1, Martin L. Griss, Reed Letsinger Agent Behavior Architectures, A MAS Framework Comparison, Hewlett-Packard Laboratories, Technical Report, HPL-2001-332
13. Singh, M.P.: Know-how. In Wooldridge, M., Rao, A., eds.: Foundations of Rational Agency. Kluwer Academic, Dordrecht (1999) 81–104
14. Bell, J. 1995. "A Planning Theory of Practical Rationality". Proceedings of AAAI'95 Fall Symposium on Rational Agency. 1-4.

15. J. O. Kephart and D. M. Chess. The vision of autonomic computing. Computer, 36(1):41–52, 2003.
16. Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart and Steve R. White. A Multi-Agent Systems Approach to Autonomic Computing. AAMAS'04, July 19-23, 2004, New York, New York, USA. pp 464 – 471.
17. "Autonomic computing: IBMs perspective on the State of Information technology", International Business Machines corporation 2001, http://www.research.ibm.com/autonomic/manifesto/
18. IBM research projects on Autonomic Computing, http://www.research.ibm.com/autonomic/research/projects.html
19. Jana Koehler, Chris Giblin, Dieter Gantenbein and Rainer Hauser On Autonomic Computing Architectures. Research Report. IBM Research Zurich Research Laboratory. June 2003
20. Ousterhout, J. K. Why threads are a bad idea (for most purposes). USENIX Technical conference, 1996.
21. Programming Java threads in the real world : A Java programmer's guide to threading architectures. Available at: http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads_p.html
22. A. Begel, J. MacDonald, and M. Shilman. PicoThreads. Lightweight threads in Java. Technical report, UC Berkeley, 2000.
23. Stallings, William.Operating systems: Internals and Design principles. Fourth Edition, Alan Apt, 2000
24. Stephen Cranefield, Martin Purvis, Mariusz Nowostawski and Peter Hwang. Ontologies for Interaction Protocols. Proceedings of the Second International Workshop on Ontologies in Agent Systems, Bologna, Italy, 15-19 July 2002
25. Allen, J. 1984. Towards a General Theory of Action and Time. Artificial Intelligence 23: 123-154.
26. Griss, M.L., Fonseca, S., Cowan, D., and Kessler, R. Using UML State Machine Models for More Precise and Flexible JADE Agent Behaviors. In Proceedings of the Third Int. Workshop on Agent-Oriented Software Engineering. Volume 2585 LNCS. Berlin, 2002.
27. FIPA interaction protocol library. http://www.fipa.org/repository/ips.html, 2001.

28. Odell, J., Van Dyke Parunak, H., and Bauer, B. Representing Agent Interaction Protocols in UML. In First international workshop, AOSE 2000 on Agentoriented software engineering, p.121-140, Springer- Verlag, Berlin, 2001.
29. Friedman Hill JESS- Java Expert System Shell. http://herzberg.ca.sandia.gov/jess/
30. Rao and M. Georgeff, "Modeling rational agents within a BDI architecture," Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, Cambridge, MA, 1991, pp. 473-484.
31. Bratman, M.: Intention, Plans, and Practical Reasoning. Harvard University Press, Cambridge MA., USA (1987)
32. G. Hernandez, A. El Fallah-Seghrouchni and H. Soldano. Learning in BDI Multi-agent Systems. (CLIMA IV) Florida, USA, January 6–7, 2004. pp 185-200.
33. Dastani, M.M., Dignum, F.P.M., & Meyer, J-J.Ch. Autonomy and Agent Deliberation. (Autonomy 2003-AAMAS-2003).
34. Adaptive Agents: http://c2.com/cgi/wiki?AdaptiveAgent
35. H. Farooq Ahmad, Hiroki Suguri, Kashif Iqbal, Arshad Ali, "Autonomous Distributed Service System: Basic Concepts and Evaluation", The Second International Workshop on Grid and Cooperative Computing (GCC2003).
36. H. Farooq Ahmad, Hiroki Suguri, M. Omair Shafiq and Arshad Ali, "Autonomous Distributed Service System: Enabling Web Services Communication with Software Agents", Proceedings of 16th International Conference on Computer Communication (ICCC), pp.1167-1173, September 2004, Beijing China.
37. Hiroki Suguri, H. Farooq Ahmad, M. Omair Shafiq and Arshad Ali: Agent Web Gateway - Enabling Service Discovery and Communication among Software Agents and Web Services. Proc. Third Joint Agent Workshops and Symposium (JAWS2004), pp. 212-218, October 2004, Karuizawa, Japan.