Generation of Program Synchronization Skeletons using declarative descriptions of Interaction Protocols.

GURUDATH.B.SUBBANNA

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE & SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL , QUEBEC, CANADA

AUGUST 2005

©GURUDATH.B.SUBBANNA, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-14337-1
Our file Notre référence
ISBN: 0-494-14337-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ABSTRACT

Generation of Program Synchronization Skeletons using declarative descriptions of Interaction Protocols.

Gurudath.B.Subbanna

Interaction Protocols capture the interactions between agents that communicate with each other to achieve specific goals. These protocols specify the conversation of agents in a Multi Agent System. Many attempts have been made to capture the semantics of Agent Interaction Protocols through the deployment of different tools and formalisms.

This report describes the execution of Interaction Protocols through the use of Agent Unified Modeling Language (AUML), an extension of the Unified Modeling Language (UML) developed specifically for agent interactions. The report describes generation of synchronization skeletons for any user-defined Interaction Protocol based on a meta-model for AUML and the Eclipse Modeling Framework. Validation of the model and scope of extensibility of the model are also discussed. This mainly provides for developers to quickly input any agent conversation protocol and successfully converts the same into code for further development over an agent platform.

Acknowledgement

It would not be possible for me to come through to this stage without the help and support of my professors, Dr. Hon.F.Li and Dr. Hovhannes Harutyunyan. They have always been a constant source of strength and support and have led me through the ups and downs of not only research, but also student life in Montreal. I have not only learnt from them academically, but a great deal of my personality as it is today, stands reformed because of their influence. Not only have I learnt from them to discern issues in research, to look at any research issue from an abstract level and later generate possible solutions, to have a keenness and instinct for tackling problems and to precisely be able to generate correct answers to what seem the most difficult problems but have also learnt from them patience, kindness, understanding and specifically - humility. Dr.Li and Dr.Harutyunyan – for changing me as a student and more so as a human being, thank you.

I would also like to thank my family in Montreal who have always been there for me whenever I needed them and have been a pillar of strength through out. I would also like to thank Dr.Stephen Cranefield from the University of Otago, New Zealand who helped me out on certain issues in my work. I would also like to thank my friends in Montreal who saw to it that I never felt alone in a new city.

Finally and most importantly I would like to thank God for being there with me all through out and showing me the light when it seemed the darkest.

Table of contents

Table of Figures

1. Introduction

This report describes a UML meta-model generated for execution of Interaction Protocols specified in the graphical Agent Unified Modeling Language (AUML). Validation of the meta-model for correctness and provisions for extensibility are also detailed. This meta-model executes graphical declarations of Interaction Protocols specified in the graphical Agent Unified Modeling Language (AUML). AUML is a graphical language based on the Unified Modeling Language for specifying Interaction Protocols in the form of sequence diagrams. Sequence diagrams are used to describe the flow and the sequence of messages or interactions while addressing a problem. Sequence diagrams are slightly different from Collaboration diagrams of UML[51] in the sense that while Collaboration diagrams emphasize on the relationships between objects, Sequence Diagrams emphasize on the sequence of messages between different objects.

The meta-model is developed using specific Computer Aided Software Engineering (CASE) tools and is implemented using the Eclipse Modeling Framework [EMF][1] and the designed meta-model is also validated. Provision for extensibility of the meta-model has been built in keeping in mind possible future additions to the meta-model.

1.1 Motivation

Agent Interaction Protocols (AIP's) capture the interactions between agents that communicate with each other to achieve certain goals during execution. These software agent Interaction Protocols are currently described using a variety of informal and formal notations. These notations are then understood by software developers and are interpreted and translated into program code to simulate the very same Agent Interaction Protocols. A sample situation has been illustrated in the figure below.

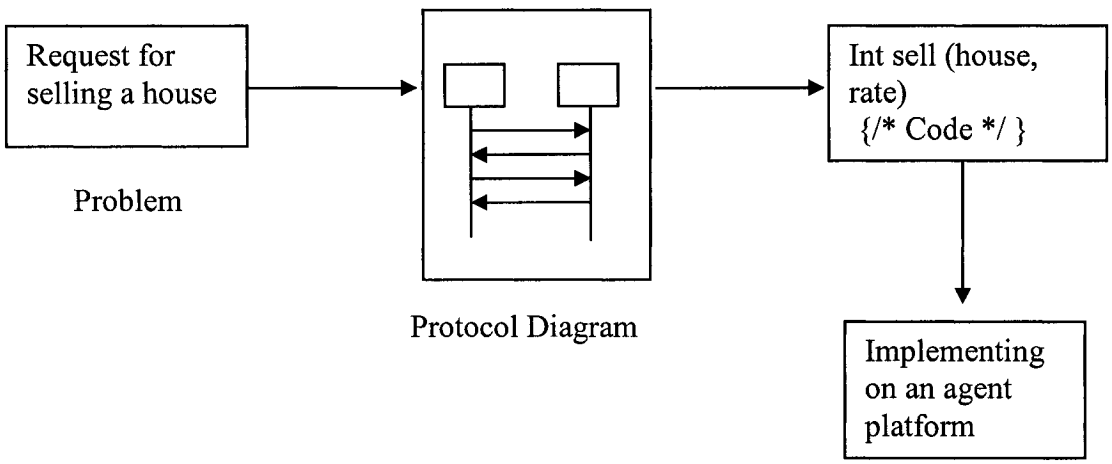


Figure 1: Agent implementation of a protocol

An idea or a problem could be described using a Protocol Diagram. These diagrams specify the sequence of interactions and how the flow of communication takes place. This Interaction Diagram can be written in any implementation language ready for execution. This language is then specifically mapped on to an agent platform and executed using software agents. It must be noted that, to describe a protocol, an explanation of the underlying notation is necessary before the protocol itself could be described. This

protocol model needs to be interpreted by the developer and then a software design needs to be created which is then executed on an agent platform. This implements a conversation for a Multi Agent System (MAS)[28].

The semantics of Agent Interactions Protocols are defined by the sequence of Sequence Diagrams and descriptions in natural languages. These semantics are usually ambiguous and vague. We need formal semantics to address Agent Interaction Protocols. There already exist a few agent development tools that support and make use of different formalisms to model interaction protocols. For example, Petri-nets enjoy widespread support amongst agent development tools for execution as part of the agent control flow. Unified Modeling Language (UML) on the other hand can be used to notate both static and dynamic models. This allows for different aspects of a system to be modeled in a single language. UML is general purpose and extensible. UML is in fact touted a complete modeling language in the sense that it supports modeling needs in all software development phases, from requirements specification through analysis, design and implementation to the final test. Finally, one main advantage of UML is that it is now becoming a new standard and there are quite a few tools these days that support it. Having such a notation that is easier and simple to use and is supported by tools makes it easier for developers to apply and build systems on. All these make UML[51] a widely accepted for modeling in the software development community. Hence a notation for Interaction Protocols that is based on UML would be highly acceptable to software developers as they are already familiar with the basic concepts and working of UML.

Such a notation would be easy to learn and additional features, again based on UML could be added with ease.

Such a notation that is developed needs to be standardized. There might be different developers that develop agents on different platforms. If they need to collaborate with each other, they need to follow a standard Interaction Protocol such that every agent follows the same protocol. In such a scenario an agent developer can just download such a protocol template, attach agent specific application code and later run this code on an agent platform to get the Interaction Protocol working. This facilitates collaboration of agents across different platforms to seamlessly synchronize with each other. Without such a standard notation for developing Interaction Protocols, each software developer has to design his own Interaction Protocol, implement the protocol as code and then execute the same code on an agent platform. Thus conversation code would have to be created each time from scratch. Apart from having the possibility of errors, such a scenario does not allow for inter-agent collaboration built on different platforms. Extensibility of such an application might also pose a problem, as future developers would have to use the same implementation language or the same platform as that of the original application. It would be easier and would limit errors if the developer could design and implement the protocol in a single step and later on build it onto an agent platform.

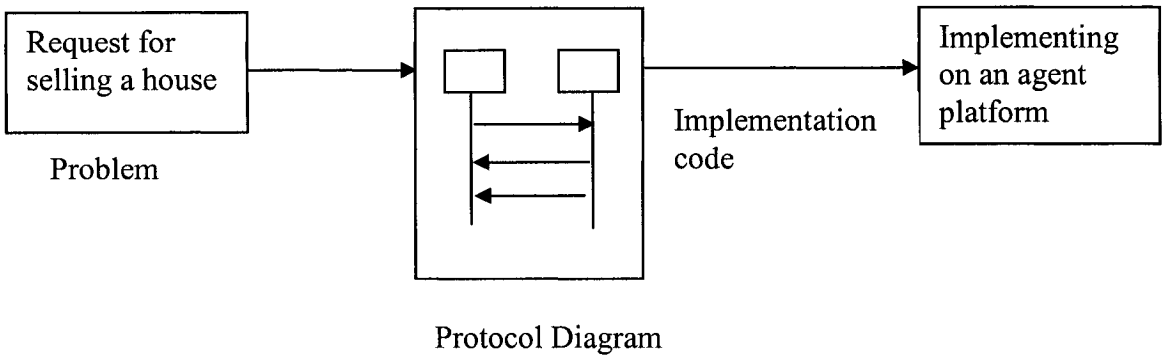


Figure 2: Agent Implementation with direct execution of Interaction Protocols

To make this possible a precise, formal and machine-interpretable notation for Interaction Protocols is necessary. One such graphical notation is the Agent Unified Modeling Language (AUML). How ever, one problem currently faced by this notation is that it does not yet have well-defined semantics, is abstract, not precise and cannot be interpreted by machines. AUML primitives consist of boxes, lines, arrows and comments in English or a programming language that cannot be understood by machines without any meta-data.

AUML needs to define the meaning of its descriptive concepts, the rules for putting them together and the syntax that is used to represent them. Based on these needs, the work described in this thesis mainly details upon the following concepts:

- Understanding of Multi Agent Systems
- Developing a meta-model for the Agent Unified Modeling Language (AUML)

- Validation of the meta-model and provision for extensibility.
- Automatic generation of code that would enable agents to execute these Interaction Protocols.

Possible work in the same field of research could have also included development of formal semantics for AUML and proof of correctness of the meta-model but this remains outside the scope of this thesis.

1.2 Main accomplishments of this thesis

A meta-model based on the UML 2.0 superstructure [24] has been proposed that executes AUML interaction protocol diagrams. The meta-model incorporates most of the main primitives that appear in Interaction Diagrams as proposed by the standards set by the Foundation for Inter-Operable Agents (FIPA). Various tools have to build the meta-model have been evaluated and finally the Eclipse Modeling Framework (EMF) has been chosen for such a purpose. The actual design of the meta-model has been drafted using a Computer Aided Software Engineering (CASE) tool, Rational Rose[18]. Using these tools, the meta-model has been designed, developed and implemented and a few Interaction Protocols have been tested and implemented.

This report is mainly structured in the following way:

Chapter 2 incorporates a background study of agents, Multi-Agent Systems, Interaction Protocols and other tools to evaluate and execute Interaction Protocols.

Chapter 3 describes the proposed meta-model for AUML. It also describes validation of the meta-model and provisions made for extensibility.

Chapter 4 presents a few sample results obtained upon testing of certain standard examples of Interaction Protocols.

Chapter 5 presents the Conclusion and talks about possible future work in this field.

Chapter 6 provides a list of references studied to better understand the field of work.

2. Multi-Agent Systems

2.1 Introduction

There is no general definition for the term agent in the context of distributed systems or for software agents in general. Wooldridge and Jennings[26] point out that ' An agent is generally a hardware or more usually a software based computer system that enjoys the following properties.

- Autonomy: Agents operate without the direct intervention of humans or any others and have some kind of control over their actions or internal state.
- Social ability: Agents interact with other agents and possible humans in some form of interaction language.
- Reactivity: Agents perceive their environment (which may be the physical world, a user via a graphical interface , a collection of other agents , the INTERNET or perhaps all of these combined) and they respond in a timely fashion to the changes that occur in it.
- Pro-activeness: Agents do not simply act in response to their environment. They are able to exhibit goal-directed behaviour by taking the initiative.'

Maes[64] defines agents as 'Autonomous agents are computational systems that inhabit some complex dynamic environment , sense and act autonomously in this environment and by doing so realize a set of goals or tasks for which they have been designed.' Hayes-Roth [25] define the same as: 'Agents continually perform three functions-perception of

dynamic conditions in the environment, actions to affect conditions in the environment and reasoning to interpret perceptions, solve problems, draw interferences and determine actions.'

Bradshaw [22] lists nine main attributes of an agent: reactivity, autonomy, collaborative behaviour, 'knowledge-level' communication ability, inferential ability, temporal continuity, personality, adaptivity and mobility. In any problem, that needs to be solved by an agent, the agent should more or less possess some of the above attributes. Wooldridge [12] discusses the difference between agents and objects. He says that the main difference between the two is autonomy of agents. An object has no control over its behaviour. If the method of an object is called, the object has no control over the method. An agent can only be asked to do something. It cannot be forced. Agents also have flexible behaviour. They react either pro-actively or reactively. Hence an agent is in control of itself and this is what exactly defines the autonomy of an agent. An agent may either co-operate with another agent to carry out complex tasks that they alone cannot handle or an agent can also move to another system to access remote resources or to even meet other agents. These are the properties of the agent paradigm that developers need to solve applications and find good solutions. Most uses of agents are mainly emphasized under two main headings- Agents used in simplified distributed computing as Intelligent Resource Managers or agents as Personal Assistants for overcoming interface problems. This solving ability of agents along with their properties, mainly autonomy and collaborative behavior led to the development of Multi-Agent Systems in the field of Distributed Computing. Durfee and Lesser [27] state that ' A Multi Agent System can be

defined as a loosely coupled network of problem solvers that interact to solve problems which are beyond the individual capabilities or knowledge of each problem solver. Sycara [28] points out that the characteristics of Multi-Agent systems are that a) each agent has incomplete information or resources for solving the problem by itself and thus has a limited viewpoint b) there is no system global control c) data is de-centralized and d) computation is asynchronous.

Multi-agent systems deal with autonomous agents that interact either selfishly or co-operatively with each other. These agents could either pursue a common goal or follow a free market economy where they are in pursuit of their own individual goals. Hence, one main advantage of a multi-agent system is its modularity. These kind of multi-agent systems are most ideally suited to open systems as in the field of Distributed Computing wherein the system is continually changing. The characteristics of such a system are that its components are not known in advance and can consist of heterogeneous agents implemented by different people at different times.

The main motivations of an agent system are as follows. The first is to solve problems that are too large for a centralized agent to solve because of resource limitations or the sheer risk of having one centralized system that could be a performance bottleneck or even fail at certain times. The second is to allow for the inter-connection and inter-operation of multiple existing legacy systems. Third is to provide solutions to problems that can naturally be regarded as a society of interacting-component agents. The fourth is to provide solutions that efficiently use information resources that are spatially

distributed. The fifth is to provide solutions in domains in which the expertise is distributed. The sixth and the most important motivation is to increase performance along the lines of a) computational efficiency as concurrency of computation is exploited b) reliability as other agents can take up the responsibility if a particular agent fails c) extensibility because the number and capabilities of agents working on a particular system can be altered d) robustness because of the system's ability to tolerate uncertainty e) maintainability because MAS are easier to maintain because of their modularity f) flexibility because agents can adaptively organize to solve the current problem and g) reusability because the same agents can be used in different systems. A few main advantages of Multi-Agent Systems are that: Sophisticated individual agent reasoning can increase MAS coherence because each individual agent can reason about non-local effects of local actions, form expectations of the behaviour of others, or explain and possibly repair conflicts and harmful interactions. Secondly, reactive agents do not have representations of their environment and act using a stimulus-response type of behaviour. They respond to the present state of the environment in which they are situated. Thus one can allow for a dynamically changing system without knowing each component in advance. Multi-agent systems also provide an extremely high degree of encapsulation. This is because even though different developers might have built the agents, they can still work together as long as they understand the common means of communication. Agent communication is one of the most important issues that an agent developer has to deal with. The field of agent communication deals with the messages, the message exchange interactions and finally the content language representations.

2.2 Agent Communication Languages (ACL's)

The most noble and profitable invention of man was that of speech, consisting of names or appellations and their connexions whereby men register their thoughts, recall them when they are past and also declare them to one another for mutual utility, understanding and conversation without which there had been amongst men neither Commonwealth, nor society nor contract nor peace no more than amongst lions, bears or wolves or any other of the kind [Leviathan, 30]. Language is defined as communication of thoughts or feelings through a system of arbitrary signals, voice or symbols. This system also proposes its rules for combining the components of the system. It could also refer to a particular manner of expression. Language fosters complex interactions between members of any community. Not all languages are natural languages. Some languages such as the language for the deaf and dumb exhibit a fundamental property of languages. The meanings of their tokens are shared and are understood by all members of that particular community. This in turn leads to the fundamental meaning of a language: Communication between willing and occasionally unwilling participants.

Languages for communication agents seem to play the same role, as does natural language in its human counterparts. An agent communication language allows agents to interact while hiding the details of the internal workings or thoughts of the agent. This helps for greater understanding and correspondence amongst agents in inter-agent communities. In short, an Agent Communication Language provides agents with means of exchanging information and knowledge [Finin, 31]. An ACL handles propositions, rules and actions rather than simple objects with some well-formed meaning attached to

each of these rules. An ACL message expresses a desired state or meaning in a declarative form rather than as a procedure. An ACL message must be described as a speech-act [Burkhadt, 32] that states a belief, desire and motivation on behalf of the agent to express itself to another agent, either within the same community or in a different community of agents. The challenge with respect to developing an Agent Communication language is to make an agent of one community talk to any other agent, not just another agent of its own community. Hence the common language developed should be mutually understandable across agents.

Some of the needs of an Agent Communication Language are that a) It must have a well defined syntax such that all agents can parse their communication statements in a similar fashion b) The communication language must have well-defined meaning or semantics such that all agents can understand the statement in the same way c) It must be following a certain standard such that developers across different organizations can use the same standard to develop further on the communication language and d) The communication language should be able to express exactly the intended ideas that the agent wants to express.

Finin[31] states that there are certain requirements that must be met by every Agent Communication Language. They are as:

- Form: A good Agent Communication Language should be declarative, syntactically simple and should be easily readable, easy to parse and easy to generate.

- Reliability: A communication language must support reliable and secure communication amongst agents. In other words, agents should be able to express to each other their basic ideas without having to go into details of their internal working.
- Semantics: This is especially important if communicative acts can take place across a range of applications. This enables application designers to have a shared understanding of the primitives and those protocols that come with the use of those primitives. The semantics of a communication language must exhibit the same properties expected by the semantics of any other language. It should be grounded in theory and should be unambiguous. It should exhibit a canonical form wherein similarity in meaning would lead to similarity in expression.
- Implementation: The implementation of the communication language must be efficient and must fit well with the existing software technologies. The interface should be easy to use.
- Environment: The environment in which agents work will be highly distributed, heterogeneous and very dynamic. It must support inter-operability protocols, as the communication language might have to regularly communicate with the outside world.
- Content: A communication language should be designed such that it fits well with other systems. The language should commit to a well-defined set of primitives termed as Communicative Acts. There has to be a distinction between the Communicative Language that expresses the Communicative Acts and the Content Language that specifies the domain.

Similarly there are certain requirements on agents that have to be met if they follow a certain Communication Language. Some of them are as:

- The agent should send across a Not-Understood message anytime it receives a communicative act the contents of which it cannot comprehend.
- The agent should only choose any subset of the Communicative Acts provided by the Communication Language that it chooses to implement.
- An ACL compliant agent must implement the Communicative Act correctly with respect to its definition.
- An ACL compliant agent should not generate a Communicative Act on it's own.
- ACL compliant agents that send or receive Communicative Acts must execute based on the semantics of the Act as defined by the Communication Language.

At the moment there are mainly two standard Agent Communication Languages used by the agent developer community. The first is the Knowledge Query and Manipulation language (KQML) that was developed mainly for the exchange of information amongst knowledge based systems that are sharable and usable. It is a message format and a message handling protocol to share run time information amongst agents. An application program to interact with intelligent systems to share knowledge in co-ordinated problem solving can use KQML. KQML consists of a set of performatives [33] that perform some action on virtue of being sent. The primary definition of KQML extensions was by definition of new performatives that conform to the standard set by KQML. Another standard of an Agent Communication language developed was the ACL by the Foundation of Inter-Operable Agents (FIPA) [34], a non-profit organization. The FIPA

ACL message contains a set of one or more parameters. Precisely, which parameter is used depends on the situation when the Communicative Act takes place. The FIPA Communicative-Acts too are based on the speech-act theory. The act allows for the sending agent to communicate to the receiver that it intends the receiver to perform some action that it wishes to. This action performed could be a query, a response or in turn another action based on the sender's belief or proposition.

How ever, one major drawback of the FIPA ACL is that it attempts to capture the mental state of the agents (such as beliefs, uncertainty and expectations) that participate in the Communicative Acts, the semantics of which cannot exactly be captured.

2.3 Ontologies

A body of formally represented knowledge is based on a *conceptualization*: the objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them (Genesereth & Nilsson, [59]). A conceptualization is an abstract view of our world. It is a simplified version of the abstractness of an idea that roots from the human mind. According to Gruber[35], an Ontology is the explicit specification of a conceptualisation. Labrou[29] states that 'An Ontology is a particular conceptualisation of a set of objects, concepts and of other entities of which knowledge is expressed, and of the relationships amongst them. An Ontology consists of terms, their definitions and axioms relating to them'. Ontology is the description of the concepts and relationships that can exist for an agent or for a community of agents. In short, and ontology is a formal statement of a theory. It could also be defined as a set of

specifications in declarative formalism. These specifications in turn ensure that agents use the same term with the same meanings during communication. Ontologies are important so that agents can communicate within a particular domain using just the shared ontological entities without sharing any global theory or without sharing of any local information. A common ontology enables agent to share queries and assertions. An agent commits to an ontology if its visible actions are consistent with the specifications as defined in the ontology. Agents have to use the ontological statements in a coherent and co-operative manner. Hence, we can say that Ontologies lead to information sharing amongst agents across different platforms.

Once ontology has been specified for a set of agents, the next thing that should be considered is the sequence of messages passed amongst agents. Given that agents follow a certain ontology, how exactly do they interact with each other? The way the interactions between agents are captured and the sequence of interactions between agents becomes an important issue in multi-agent systems. These concepts are captured by what the agent community defines as Interaction Protocols.

2.4 Interaction Protocols (IP's)

The main inspiration of using an agent communication language for agent communication comes from communication between humans. Humans basically use predefined, commonly understood languages and parameters for daily life communication. This way of communication is very effective when there is no expected

behavior required from both parties that want to communicate. But when there is a need for some expected behavior in communication between humans, there are certain ways of behaving. Societies of agent collaborate to collectively perform tasks by entering into conversations with each other. These conversations consist of messages sent by one agent to another and with the receiver responding back to the sender agent by performing some action or by responding back with another message. These messages are placed spatially in time and they follow a certain order. Sequence of messages that might be as simple as a request for information / release information duo might have multiple negotiations involved between the two agents. In order to enable to communication in such a fashion with each other such that they are hidden from each other's implementation details, a concept called as Interaction Protocols has emerged. Interaction Protocols has been one of the foremost standards for agent inter-operability. Since Interaction Protocols are tools for defining Agent Communication, public Interaction Protocols might even lead to open, dynamic agent societies.

Wikipedia [39] defines Interaction Protocols as possible communication scenarios between individual agents in a multi-agent system . The basic foundation for Interaction Protocols was built by Bradshaw and Greaves, [66] when they defined the problem for conversation amongst agents and termed it as the Conversation Policy. They state that agents must agree on the range of possible sequences and contents of messages when they are collaboratively communicating on solving a long-term goal. How ever, they also state that the existing ACL's do not suffice for this purpose and the precise semantics of messages are not clearly defined.

Based upon this, they came up with what they termed as the Basic Problem which stated:

> Modern ACL's, especially those based on logic are frequently powerful enough to encompass several different semantically coherent ways to achieve the same communicative goal and conversely also powerful enough to achieve several different goals with the same message.

As a solution to this problem, Greaves et al suggest that agents need to conform to a specific policy to follow when establishing conversations with each other. This policy is modeled and defined by the Interaction Protocol. The Interaction Protocol functions as a template to order the form, content and ordering of possible messages such that a single communicative goal could be reached co-operatively. In short, an Interaction Protocol describes the sequence and flow of messages between agents in a multi-agent system. They constrain the possibility of messages to form sequences of a particular type as Interaction Protocols significantly reduce the search space for possible responses. In comparison to human interactions, the Interaction Protocol could be viewed as a set of responses in answer to a particular question.

Cranefield [38] states that an Interaction Protocol should satisfy the following:

- Other developers should be able to implement the protocol without confusion.
- Software agents should be able to interpret these protocols for generating

meaningful conversations.

- The Interaction Protocols should specify the generality of meaning. It should hold the same meaning for a given set of messages in any situation.

There should exist a clear distinction between Interaction Protocols that defines the agreement with respect to the flow of messages between agents and the Conversation Policy that states how an agent communicates with another agent. While the first approach is more policy-centric the second approach is more agent-centric. Interaction Protocols are better refinement versions of Conversation Policies. Greaves et al [40] pointed out the requirements to be met by any Interaction Protocol. They are as:

- The separation of Protocol and Implementation: The behavior of an agent should be specified independent of the implementation methods used.
- The specification of Interaction Protocols should allow the identification of equivalent classes of interactions across different formalisms.
- Protocols should be flexible enough to allow agents with different levels of sophistication to talk to each other.
- Protocols should be compositional. Different pieces should be able to be put together so that they could govern different domain specific interactions.

One fundamental advantage of an Interaction Protocol is that it restricts flexibility of the agent to choose from a variety of responses. At the same time, it gives agents enough freedom to talk to any other agent of their choice and establish a pattern or sequence of

messages with them. To facilitate such communication amongst agents that satisfy the requirements as presented above, the protocol by itself must be a) unambiguous (should not be vague and must be specific) b) correct (there must be no contradictory states in the Protocol) c) complete (no state in the protocol must be half defined) and d) verifiable (the correctness properties of the protocol can be proved).

The ability to express correct protocols depends on a large part on the specification language or the tool that is used to model the protocol. There are a number of requirements for specification language / modeling tools to be able to represent an Interaction Protocol. There is much formalization available that can be used to describe Interaction Protocols. Each one of them has their own advantages and disadvantages. How ever, one main criterion for using a formal tool for denoting Interaction Protocols is that the protocol modeled using the tool could later be verified for correctness. Jennings [41] states that there are a few requirements that need to be considered while developing or choosing a language that can represent protocols.

- Provide a graphical representation for ready perception of structure by developers. Be close to an executable language for implementation purposes.
- Have an unambiguous formal specification language with clear semantics for verification.
- Provide well-defined program logic for ensuring complete protocols and validating the properties of the protocol.
- Exhibit enough expressiveness for agent interactions and nested interactions.
- Allow ease of re-use and abstraction of protocols.

- Maintain a pre-positional form of a formal language.
- Allow a methodology for compatibility with existing methodologies and Interaction Protocols. For the sake of referring to part of an interaction, the modeling language has to represent both the possible states and the possible actions.

There are many tools and description languages available that are used to specify Interaction Protocols that comply with the above requirements.

2.5 Modeling Formalisms for Interaction Protocols

Interaction Protocols modeled with Petri-nets

Jensen [43] states that Colored Petri nets form a graphically oriented language with a formal specification for design, specification, simulation and verification of systems. Colored Petri nets aid in modeling concurrent conversations in an integrated fashion. Colored Petri Nets are similar to ordinary Petri nets in the sense that they use the same notations- places and transitions and arcs connecting these two elements. The notation of Colored Petri Nets (CPN's) introduces the concept of tokens over Petri nets, each of which are marked by different colors that represent arbitrary data values. Each place has an associated type that may determine the type of data that the place may contain. These structured tokens could be evaluated to yield new net marking when transitions are fired. To be able to occur, a transition must have sufficient tokens on its input places and these tokens must have values that match the arc expressions. An advantage is that CPN

models can be analyzed in different ways. Automatic simulation is one option wherein the Colored Petri Net model developed can be simulated to establish the working of the model. It is also possible to specify time delays that describe the duration of different actions that are modeled in the system. Colored Petri Nets have formal semantics that make the execution and simulation of Petri Net models unambiguous. Another major advantage of Color Petri Nets is that they can be represented graphically and also support concurrency. They can be well understood and can be applied to many real-world applications. Finally, Petri nets can model both states and events which is an important aspect required to model agent communication.

Purvis et al [42] have described Interaction Protocols using Colored Petri-nets. The authors illustrate the FIPA Request Protocol [46]. In this protocol, there are two participating roles, an Initiator and a Participant, which are modeled using agent(s). The Initiator requests the Participant for some information. The Participant, upon receiving the message has two options. In the first case, it does not understand the message sent and responds by sending a Not-Understood message. In the second case, when the Participant understands the message, it again has two options. It can either agree to the Initiator's request and provide the Initiator with the information or it can refuse to provide the Initiator with the information requested. This is the protocol that has been modeled using Colored Petri Nets. The model for the Request Interaction Protocol for the Initiator has been shown below.

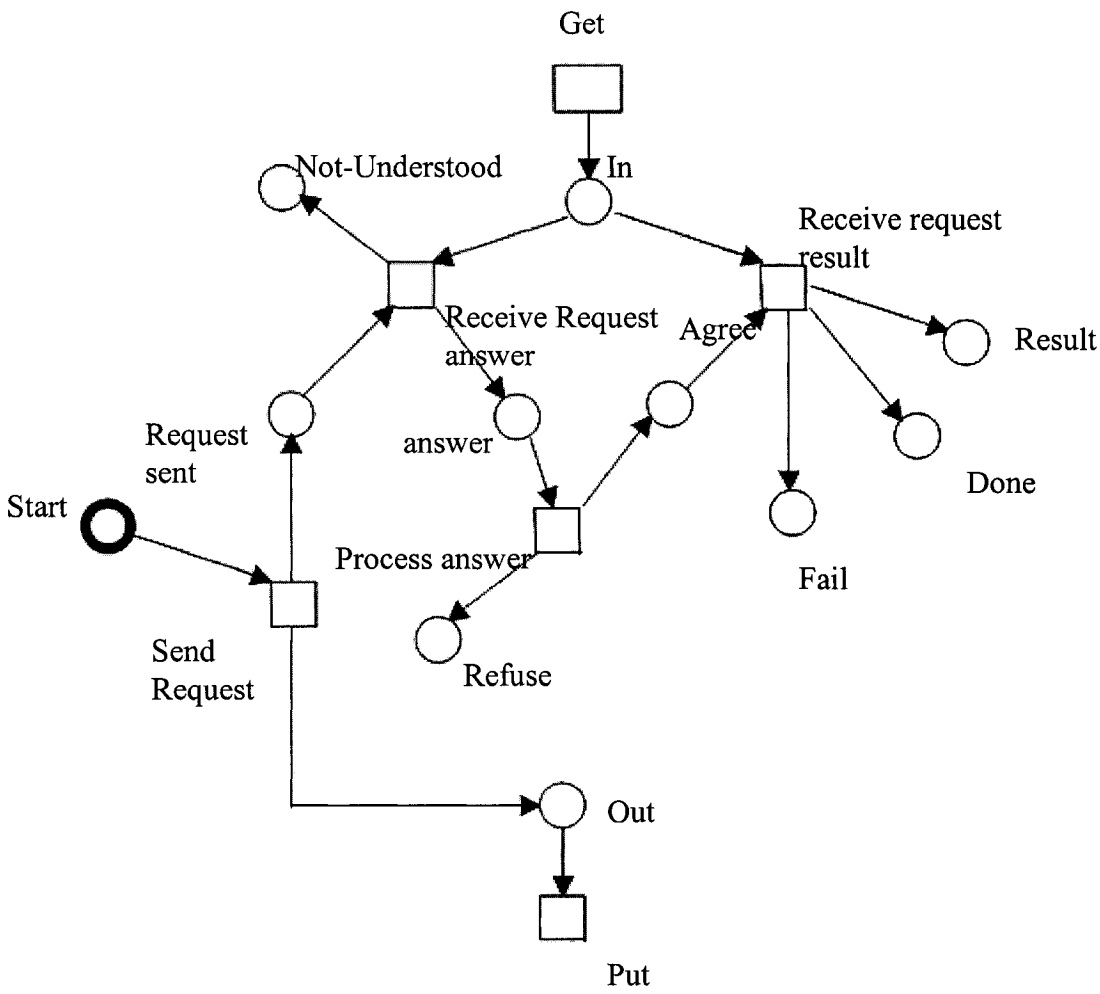


Fig 3: The Request Interaction Protocol for the Initiator role

The Petri Net scheme for the Participant role is shown below. The role-names Initiator and Participant are provided by FIPA. The 'Get' transition has code associated with it that obtains information from the agent's message receiving module and places it in the 'In' place. Transitions connected to 'In' have the guards on them such that transitions are only enabled by the tokens at the 'In' place with appropriate qualification.

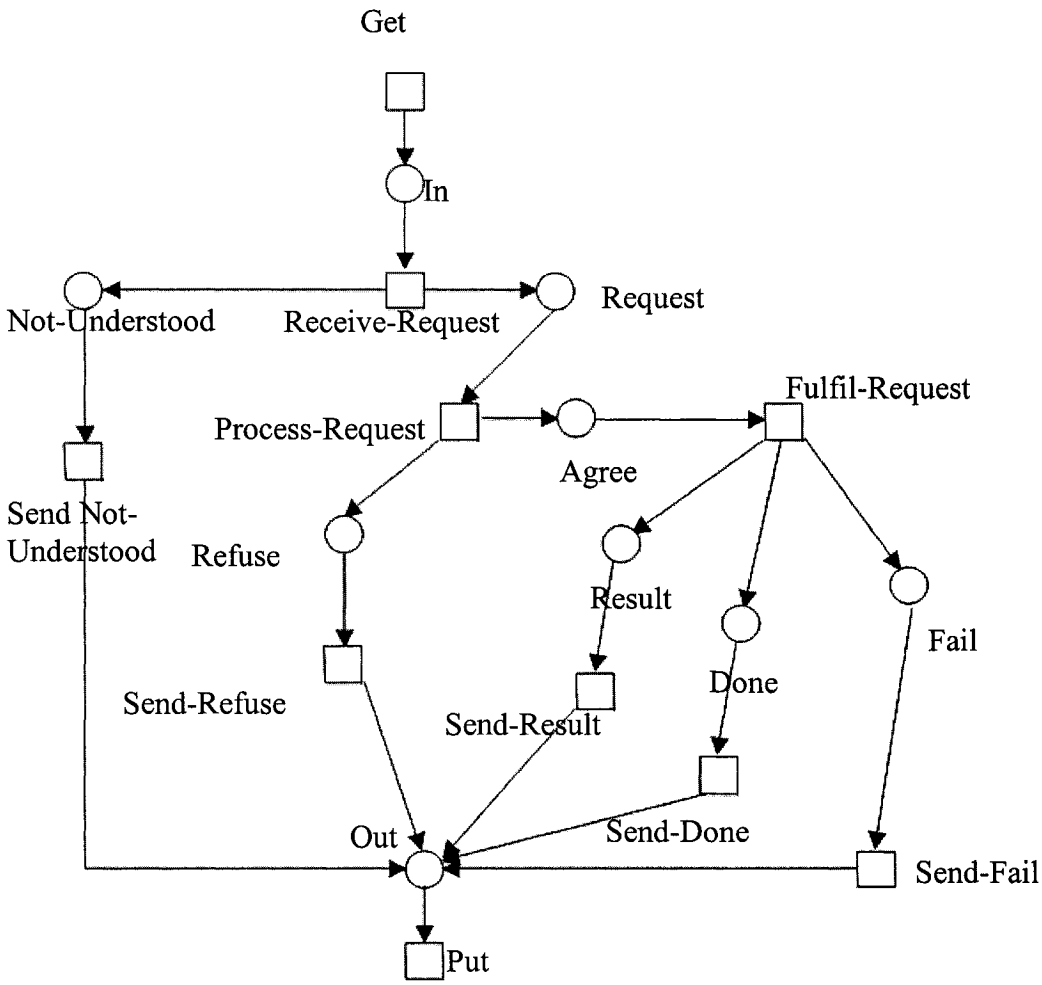


Fig 4: The Request Interaction Protocol for the Participant role

The Participant role-playing agent gets the message and places it in the 'In' place. Once that is done the transitions are fired as specified by the protocol. In any case, if either the Initiator or the Participant does not understand a particular message, it fires a NotUnderstood transition, one of the basic features of an ACL compliant agent. The

dashed lines indicate that these nodes are placed on parallel nets that deal with exceptions. Thus an Interaction Protocol has a specific Petri Net for each node in the conversation. The participating roles can make use of the Petri Nets to keep track of what stage they are in the conversation. Cost et al [45] proposed another method of modeling Interaction Protocols based on Colored Petri Nets wherein they use a single graphical model for description of the entire Protocol. There are no separate Petri Nets for different roles and the entire Protocol is integrated into one single Petri Net. According to this model any conversation always has just one final or accepting state and the final state denotes the result of the conversation. How ever, in some situations the authors state that it might be desirable to possess multiple accepting states. If denoted using this method, the FIPA Request Protocol could be denoted as below.

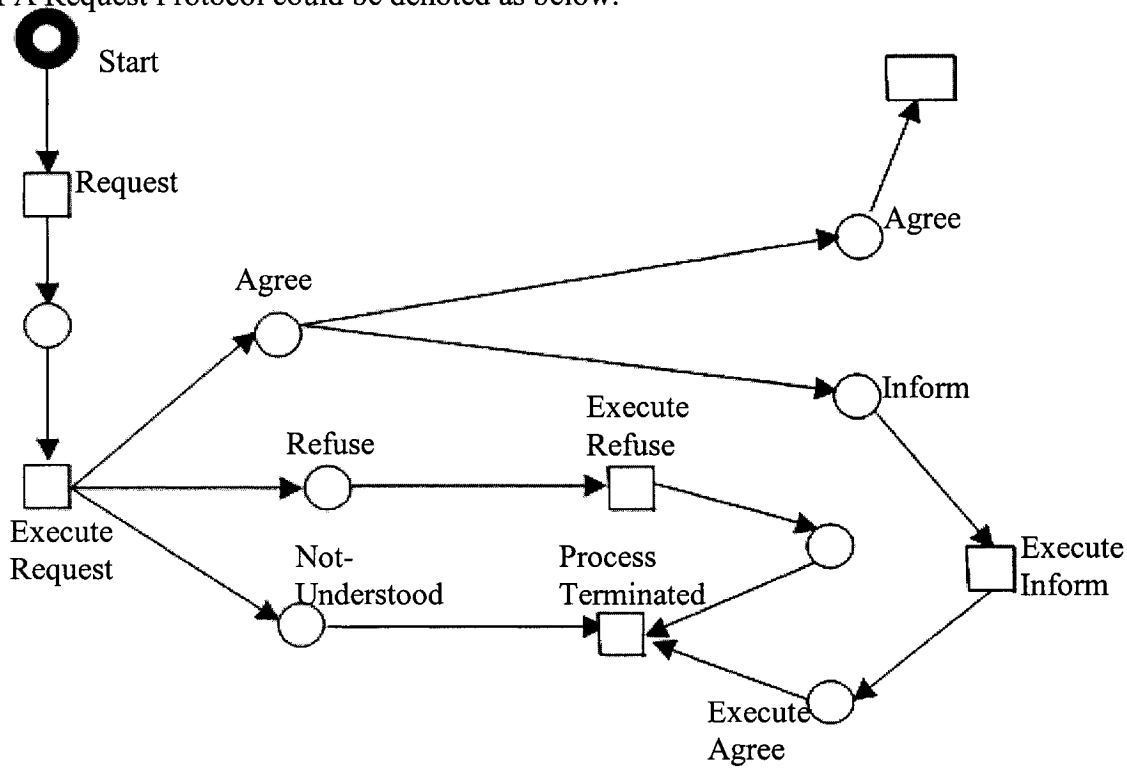


Fig 5: Depiction of the Request Protocol in the Petri Net Model using Costs method

Colored Petri Nets appear to be one of the formalisms for specifying Interaction Protocols where tools are available for integrating these protocols into a multi-agent system. Since they are easily verifiable too, Colored Petri Nets are widely used in the agent community for modeling Interaction Protocols and implementing them.

2.6 Interaction Protocols in Agent Unified Modeling Language

Agent Unified Modeling Language

Bauer [48] proposed that Agent Unified Modeling Language (AUML) be another formalism for specifying multi-agent interactions. Agent UML is an extension of the Unified Modeling Language, a de-facto standard for object oriented analysis and design. Let us first take a look at the Unified Modeling Language, its standards and formalism and how it led to the development of its agent extension, AUML.

Unified Modeling Language

Model-Driven Architecture and the Object Management Group.

Computing infrastructures are expanding into new horizons and technologies are developing everyday. These new technologies now face 'challenges of integration' as they expand into organizations. As computers and networks become faster day-by-day, interconnection standards must evolve. To maximize flexibility, investors are buying

hardware and software that implement open, inter-connection standards. Because this speed of innovation and invention is very high, it becomes very hard to develop systems that can be used efficiently for a very long time. The Object Management group (OMG) addresses this problem with the concept of a Model Driven Architecture (MDA) [48][49]. Modeling is the design of a system before coding. A model ensures that functionality is complete and correct, that end-user needs are met and program design supports requirements for scalability, robustness, security, extendibility and portability before implementation in code that makes design changes expensive to achieve at a later stage. The MDA states that building an abstract model that is independent of implementation issues can be used to specify any system or a subsystem. Hence MDA allows long-term flexibility of a) Implementation b) Integration c) Maintenance and d) Testing and Simulation.

The MDA separates the specification of the operation of a system from the way the capabilities of the system are implemented. In brief, MDA provides an approach for specifying a system independently of the platform that implements this system. This platform-independent model is then transformed onto a platform specific model and by applying suitable transformations that are required for that particular platform. This way, if technology changes or if new sub-systems are implemented or if the model itself is slightly changed, all that needs to be done is a slight modification of the mapping from the Model-specified Architecture to the platform. Portability, Re-usability and Interoperability are three main concerns of the MDA. The OMG states that some of the core concepts of MDA are as described below:

- System: This may include a program, a computer, a network or combination of hardware and software parts.
- Model: The description of a system and its environment for a particular purpose.
- Architecture: Specifications of parts and connectors of the system and rules for interactions using these parts.
- Platform: A subset of technologies and sub-systems that provide for a set of functionality though interfaces and specified usage patterns.
- Application: The functionality being developed that is based on the specification of a model.

Interaction Protocols can be viewed as being Model Driven. They are independent of the platform on which they would be implemented and provide a specification of the operation and behavior of the system. They are later implemented on a specific platform and any addition to an Interaction Protocol would just involve addition of the new mapping in the Protocol onto the specification. Interaction Protocols provide an abstract viewpoint on the generic behavior of a system in a particular environment. Hence, we can make use of Model Driven Architectures and concepts to approach Interaction Protocols.

The Unified Modeling Language (UML)

The Unified Modeling Language [51] is specified by the Object Management Group (OMG) and the foremost thought leaders were Rumbaugh, Jacobson and Booch. According to them ' UML is a general-purpose tool for a Visual Modeling Language that

is used to specify, visualize, construct and document the artefacts of a software system. It captures decisions and understanding about systems that must be constructed'. There were different approaches to Object Oriented Analysis and Design before the OMG was set up. How ever, later on they merged to become the standard set by the OMG defined as the UML. The latest standard of UML is UML 2.0 [24] on which most of this work is based.

Rumbaugh et al [50] state that UML models the following properties of an Object Oriented System.

- Static Structure: This models the key concepts of an application, the internal properties and the relationships between the different concepts. This includes class diagrams and package diagrams as well.
- Dynamic Behavior: This models the life history of an object or an application as it interacts with the rest of the world and also models the communication patterns of a set of connected objects as they interact to implement a certain behavior. This includes Interaction Diagrams, State Charts and Activity Diagrams.
- Use Cases: The specification of actions that a system can perform through interactions with actors in the outside world.
- Implementation Constructs: UML models both logical analysis and physical implementation and certain constructs used can model implementation too.
- Model Organization: The modeling information is divided into coherent pieces so that different teams could work on different pieces concurrently. The entire model is divided into sub-models.

- Extensibility Mechanisms: UML provided limited extensions capability for day to-day needs. Constraints, tagged values and stereotypes are some of the most common extensibility mechanisms used in UML.

UML is used as of the tools for the Model Driven Architecture as it is platform independent and can be used to specify models of systems as described above. Moreover, any UML model combines two major factors-semantic representation and visual notations. The semantic model also includes the syntactic structure, well formedness and execution rules. This is used for code-generation, validity checking etc. The visual presentation represents this semantic information in a form that can be seen, browsed and validated by humans. UML also pushes the following factors to be considered while developing a model- a) Abstraction vs. Detail b) Specification vs. Implementation c) Description vs. Instance and d) Variations in Interpretation. Due to these aspects of UML it is tightly bound to the development of Model Driven Architecture.

The Unified Modeling Language and Multi-Systems.

Since UML is well known in the software engineering community, many researchers and organizations proposed that UML be used for development of multi-agent systems. Botelho and Bento [52] proposed that UML could be used for representing Ontologies in multi-agent systems. According to them some of the main benefits of doing so are the graphical representations, the use of UML primitives and the availability of an Object Constraint Language. Parunak et al [53] make use of the concepts of UML to represent

agent interactions and state that UML based tools are ideal to model agent based Interaction Protocols. Odell et al [54] go further to represent social structures in UML. They compare agent societies to human societies and state that capturing of agent interactions is the foremost step in understanding agent collaborative behavior. They also propose ideas that capture the concepts of groups, roles, dependencies and speech-acts into a coherent structure for describing organizational structures and propose UML extensions to support the analysis, design and implementation of multi-agent systems.

Object oriented methodologies encourage encapsulation but it is sometimes essential to share knowledge amongst different interacting agents. The behavior of agents might differ from methods in the sense that they are non-deterministic and can vary over times. But at the same time agents can inherit knowledge from plans (similar to methods) and beliefs which are similar to instance variables. With appropriate modification, agent interactions can be easily compared to that of objects and can be captured using UML. This along with the ease of use of AUML, its visual notations and growing popularity make it an ideal tool for capturing agent interactions in multi-agent systems.

2.7 The Agent Unified Modeling Language (AUML)

Since multi-agent systems were always characterized as extensions of object oriented systems, system designers always had difficulties trying to capture the unique features of MAS using object oriented tools. In response to this difficulty, the Agent Unified Modeling Language or AUML was developed. Rather than relying completely on UML,

AUML reuses the Unified Modeling Language where necessary. AUML is developed in association with FIPA and the OMG with the main aim of specifying formalism for agent-based systems.

Bauer [55] was one of the first few who suggested the application of UML for agents to and suggested the development of layered protocols. FIPA adopted this and developed a standard for binding agent interactions and UML together and is standardized by the FIPA modeling Technical Committee [57]. Some of the main objectives of the FIPA Modeling TC are as:

- Facilitate advances in state-of-the-art agent modeling.
- Enable developers to better understand hot to develop agent based models
- Recommend technologies to enable adoption of common semantics, meta-models and syntax for agent based models and applications.
- Promote inter-operability across the lifecycles of AUML tools / products.

Currently FIPA is underway developing specifications for AUML based on the latest version of UML release by the Object Management Group-UML 2.0 and their first drafts are in progress. One of the main specifications that FIPA tries to model is the communication interaction between agents. Normally, agents follow a specified pattern

of communication with other agents in a multi-agent system to reach a common goal. This pattern of communication can be defined as an Agent Interaction Protocol.

Parunak et al[38] define an Agent Interaction Protocol as a communication pattern of an allowed sequence of messages between agents and the constraints of the content on those messages. AUML is normally used to model Interaction Protocols because they are complex enough to illustrate the non-trivial use of AUML and are used commonly enough to make this useful to other agent-application developers. Agent Interaction Protocols are said to be good examples of software patterns which are useful in one particular context and probably useful in others too. A specification of an Agent Interaction Protocol provides an example that could be used in solving problems in an agent-oriented system design. AUML provides a specification technique for AIP's with both formal and intuitive semantics with a user-friendly graphical notation. The semantics should provide precise, usable definitions of the model and the graphical representation should make the Agent Interaction Protocol understandable to humans. An Agent Interaction Protocol is a slight modification of the Sequence Diagram as denoted by UML, except that it has abilities to capture and represent interactions between communicating agents. Agent Interaction Protocols are the foremost steps a developer has to follow when establishing multi-agent societies.

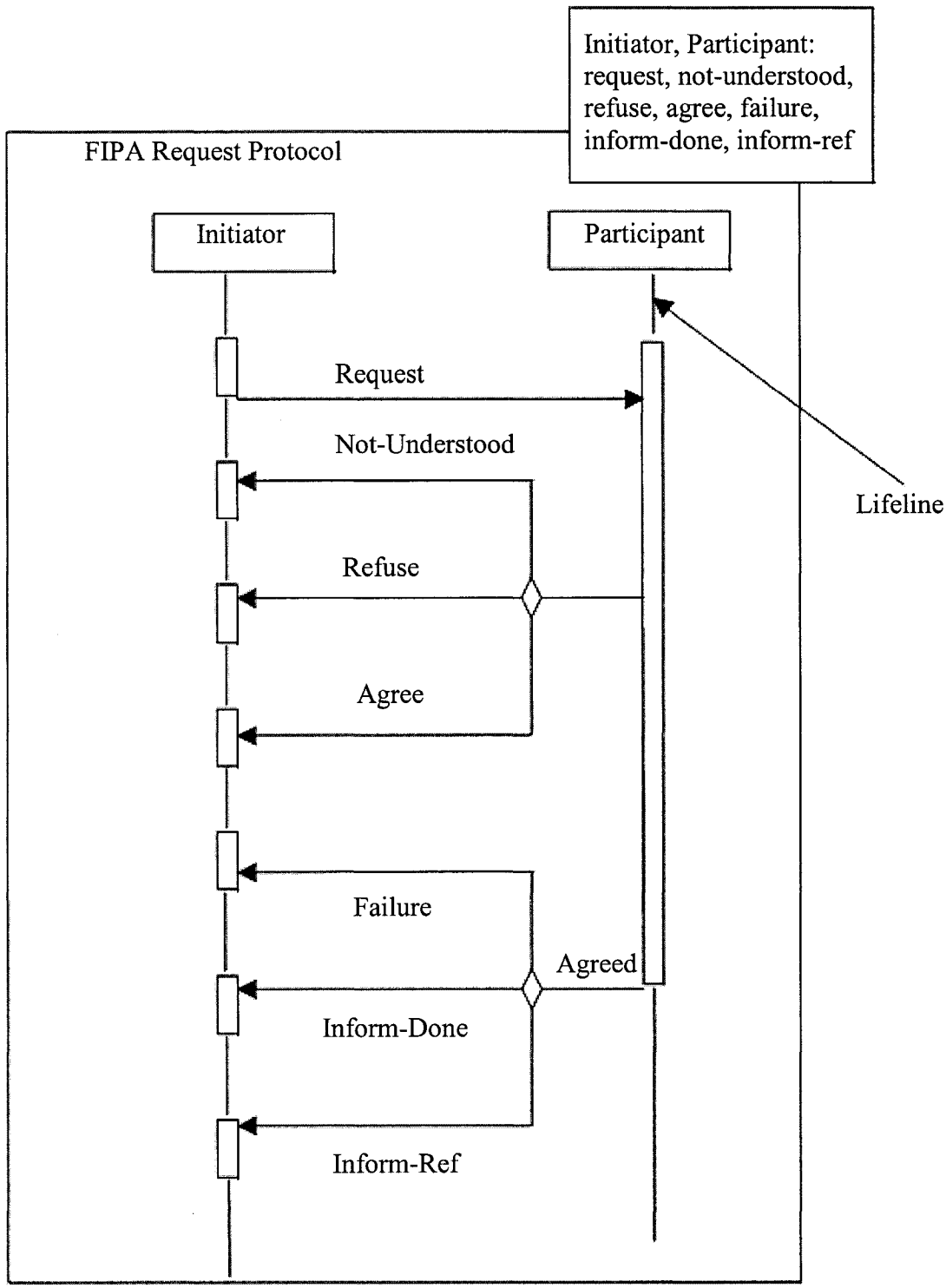


Fig 6: The FIPA Request Protocol

This is a standard FIPA AUML representation of an Agent Interaction Protocol, in this case the FIPA Request Protocol. The UML Sequence Diagram depicts the inter-agent transactions that are needed to implement the protocol. An Interaction Protocol consists of the following elements: Lifelines that denote the duration of life of a particular agent. A lifeline is represented by a vertical dashed line with the name of the lifeline enclosed within a box at the top of the lifeline. A lifeline depicts one or more actors or in the case of agents one or more agent roles. Agent Roles depict the different roles or types that are participating in the conversation. A role is an instance- focused term. In this case, the roles are those of the Initiator and the Participant. A single agent is capable of representing multiple roles too. Different lifelines are connected through messages represented by a horizontal line and an arrow at one end. The lifeline to which the head of the arrow points to is the receiver of the message while the lifeline at the tail of the arrow indicates the sender of the message. A lifeline can be associated with actions that are represented by boxes represented on the lifelines. These boxes are points along lifelines wherein the sending or receiving of messages take place.

Any Sequence Diagram is read from the top to bottom and the messages along lifelines are ordered along time. This means that the first message in reading the diagram from top to bottom is the first message interaction, the second message appearing in the same order is the second message interaction and so on. Most of the primitives used in Sequence Diagrams are described by the FIPA Interaction Protocol Specifications [58] that specify how different Interaction Protocols can be modeled using the provided primitives. AUML also supports nested and interleaved protocols and protocol templates.

2.8 Interaction Protocols specified using other formalisms

Various tools have been proposed by researchers for specifying Interaction Protocols. Yan and Qi [65] have proposed the development of ROMAS: A Role Based Methodology for Agent Systems. This approach is mainly role based and proposes to express interactions as constraints applied on the role. Gervais et al [60] focus on building an Architecture Description Language. Mylopolous et al [61] devised a tool, TROPOS that tried to mainly capitulate on the mental states of an agent during software development and required both the understanding of the environment in which the agents operated as well as their interactions with other systems. Burmeister et al [62] tried capturing agent interactions using Dooley graphs. Wooldridge, Jennings and Kinny [63] present the Gaia methodology for agent-oriented analysis and design. Gaia is a general methodology that supports both the micro-level (agent structure) and macro-level (agent society and organization structure) of agent development. The motivation behind Gaia is that existing methodologies fail to represent the autonomous and problem-solving nature of agents; they also fail to model agents' ways of performing interactions and creating organizations. These apart, many other formalisms have been specified for representing Agent Interactions. So far, only Colored Petri Nets and the Agent Unified Modeling Language have been able to gain widespread popularity mainly because of their a) ease of use b) graphical representations c) exact specifications of syntax and semantics and d) provisions for scalability. Also, the Petri Net and UML models are widely known and can be easily learnt. The other formalism methods are slightly tougher to learn when compared to these two methodologies.

One of the main advantages of Colored Petri Nets over Agent Unified Modeling Language is that there are currently no tools or editors supporting the graphical representations of AUML. How ever, at the same time Colored Petri Nets enjoy enormous support when it comes to translating Petri Net models into executable code. There are numerous tools that aid developers in validating a CPN model , accepting it from the developer and inputting it into the system. As far as we know, AUML does not enjoy this tool-based support. Currently there are no graphical editors that can accept an AUML Protocol Diagram and generate code that can act as a synchronization skeleton for the provided Interaction Protocol.

3. The AUML meta-model.

3. 1 Definition of a meta-model

Ingenias [36] describes the concept of a meta-model as " A meta-model describes formally the model elements, and the syntax and semantics of the notation that allows their manipulation. In other words, a meta-model describes the elements of a modeling language. Hence, they are used to specify other models. For example, the meta-model of a relational database system would specify the types ' Table', 'Record' and 'Field'. A meta-model can also be seen as a precise definition of the constructs and rules needed for creating semantic models. A meta-model is an explicit model of the constructs and rules needed to build specific models within a domain of interest. A valid meta-model is an ontology but not all ontologies are built explicitly as meta-models. Overall, a meta-model can be viewed from three different facades:

- As a set of building blocks and rules used to build other models.
- As a model of a domain of interest and
- As an instance of another model.

Hence, we can understand that meta-models are always made for a particular purpose.

A few common purposes for meta-models are as:

- A 'schema' for semantic data to be interchanged.

- As a language that supports a particular methodology or a process, for example as in the UML meta-model
- As a language to express additional semantics of existing information.

Meta-models are always about precise meaning so that an understanding of the elements being modeled can be created and its very important that precision is taken care of when those meanings are defined. Hence meta-models are typically built according to a strict rule-set. Meta modeling is gaining popularity in the recent days because of the following reasons:

- The advent of meta-model driven packages such as CASE tools which are fully extensible or fully configurable. This allows customisation of software development methodologies.
- The increasing availability of meta-model driven technologies and standards.
- A general need to raise the level of abstraction.

Consider a simple AUML Interaction Diagram as shown below.

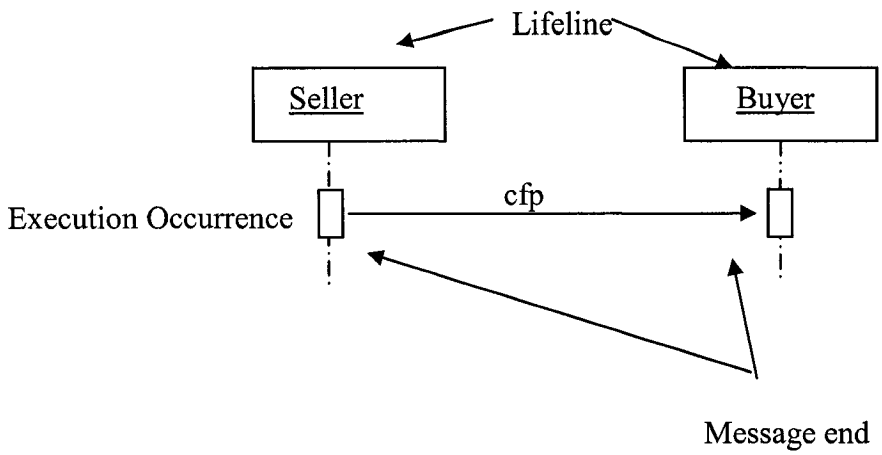


Figure 7 : An example of a AUML Interaction Diagram depicting a few sample AUML elements

As we can see an AUML Interaction Diagram consists of many AUML primitives- lifelines, messages, message endings and so on which are represented as boxes, lines and arrows. Exactly these elements and the relationships between then are captured in the AUML meta-model. The AUML meta-model tries to impress upon the underlying semantics of the AUML primitives as defined by the FIPA specifications. As an example: It is always mandatory that messages are exchanged between lifelines, every message has a corresponding message ending and so on. Thus a meta-model becomes important because every user-defined Interaction Diagram can be expressed as an instance of the meta-model. In other words, the described meta-model supports the methodology of model-development as described above. Fig 7 is now expressed as an instance of the meta-model as shown below.

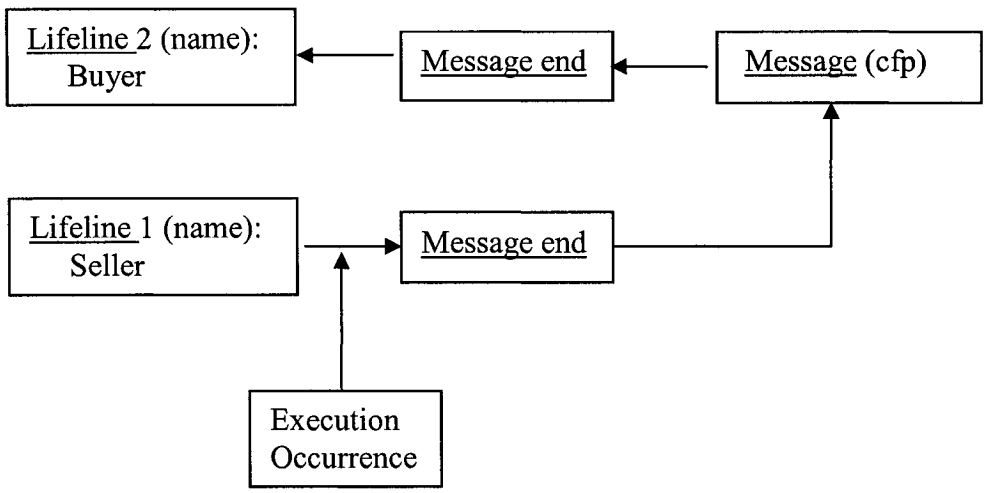


Figure 8 : The AUML Interaction Diagram expressed as an instance of the meta- model.

Since all these objects are expressed graphically, they cannot be understood by a machine or a computer. Hence, the next step would be to express the meta-model into a machine-understandable format. AUML is based and adapted from the Unified Modeling language and many of the AUML primitives are derived from UML and are modified based on the needs of agent interactions. Following this, the meta-model for AUML too has been based on the UML 2.0 superstructure. The following section deals with the proposed meta-model.

3.2 Description of the meta-model

A detailed description of the different classes referred to in the meta-model and their relationships with each other are explained in this section .

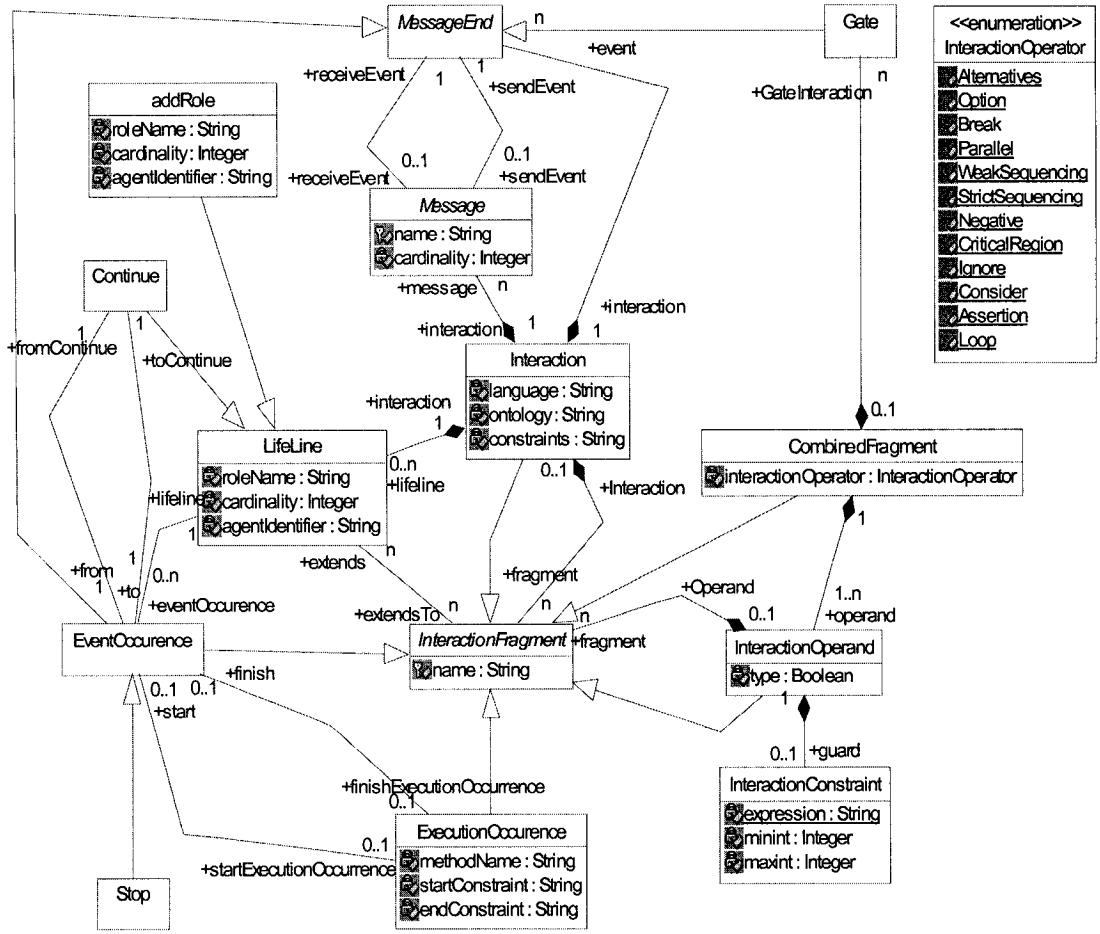


Figure 9: The AUML meta-model.

Each of the above classes along with their intended semantics, their associations and attributes are described in the following pages.

1. Interaction Fragment

Interaction Fragment is an abstract notion of the most general Interaction unit. An Interaction fragment is just a piece of an interaction and can refer to any interaction that takes place in the model. It is described as an abstract class.

Attributes:

name:String- The name of the fragment

Associations:

operand: InteractionOperand[0..1]- The operand that encloses this fragment.

extends: Lifeline[*]- References the lifelines that the fragment involves.

interaction:Interaction[0..1]- The Interaction enclosing this fragment.

2. Interaction

An interaction is defined as a unit of behaviour that deals with the observable exchange of information. Interaction is a specialization of Interaction Fragment and also encloses the class.

Attributes:

language:String- The language of the Interaction

ontology:String- The ontology for this Interaction Protocol

variables-String Global set variables.

Associations:

lifeline:Lifeline[*]- This association specifies the participating entities in the Interaction.

event:MessageEnd[*]- The Message Ends refered to by this Interaction.

message:Message[*]- The Messages contained in the Interaction.

fragment:InteractionFragment[*]- The set of Interaction Fragments contained within this Interaction.

3. Message

A Message is defined by the UML 2.0 superstructure as a particular communication between Lifelines of an Interaction. Hence, the messages not only involve just an act, but they also specify a sender and a receiver and correspondingly a sending event and a receiving event. Hence, a Message associates with itself two EventOccurences- a sending EventOccurrence and a receiving EventOccurrence. Message is an abstract class.

Associations:

interaction:Interaction[1]- The enclosing interaction that contains the message.

sendEvent:MessageEnd[1]- Refers to the sending of a message.

receiveEvent:MessageEnd[1]- Refers to the receiving of a message.

Constraints

- If the sendEvent and the receiveEvent of a message are ordered on the same lifeline, i.e. if a lifeline is sending a message to itself, the sendEvent must be executed before the receiveEvent.
- sendEvent and receiveEvent are mutually exclusive and cannot be referenced by a single MessageEnd at the same time.

4. MessageEnd

As described above, a MessageEnd represents the start or the end of a message during an interaction.

Associations:

sendMessage:Message[0..1]- References the message that contains the information of a sendEvent.

receiveMessage:Message[0..1]- References the message that contains the information of a receiveEvent.

interaction:Interaction[1]- The enclosing interaction that owns the message end.

Subclasses of MessageEnd define the appropriate semantics that they represent. This shall be looked into later.

5. Lifeline

A Lifeline represents one or more agents. It can either represent a particular role the agent is playing or can refer to the specific agent itself.

Attributes:

name:String- The name of the Lifeline.

Associations:

interaction:Interaction[1]- References the Interactions enclosing this Lifeline.

eventOccurrence:Interaction[*]- References the EventOccurrences of the Lifeline.

extendsTo:InteractionFragment[*]- References the InteractionFragments that cover the Lifeline.

Semantics of Lifeline:

The order of EventOccurrences along a Lifeline is significant and they denote the order in which these EventOccurrences are going to occur.

6. EventOccurrences.

EventOccurrences represent specific moments of time with which certain actions are associated. An EventOccurrence is the basic semantic unit of Interactions. The sequences of EventOccurrences make up the meaning of Interactions in their entirety. An EventOccurrence is a specialization of InteractionFragment and of MessageEnd. EventOccurences are ordered along a Lifeline. The namespace of an EventOccurrence is the Interaction in which it is contained.

Associations

startExecutionOccurrence:ExecutionOccurrence[0..1]- References the ExecutionOccurrence of the sta rt of an action

finishExecutionOccurrence:ExecutionOccurrence[0..1]- References the ExecutionOccurrence of the finish of an action.

lifeline:Lifeline[1]- References the Lifeline on which the EventOccurence appears.

It has to be noted here that each ExecutionOccurrence is described and generated as an instance of an EventOccurrence.

fromContinue:Continue[1]- References the EventOccurrence of the point from where the execution previously stopped

toContinue:Continue[1]- References the EventOccurrence of the point from where the execution would Continue

7. Stop

Stop is an EventOccurrence that occurs along a particular Lifeline. It defines the termination of the instance specified by the Lifeline along which Stop occurs.

Constraints:

No other EventOccurrences may appear below a Stop on any given Lifeline in an InteractionOperand.

8. ExecutionOccurrence

An ExecutionOccurrence is defined by the UML 2.0 superstructure as being the instantiation of an action. Since the ExecutionOccurrence is normally of some duration it is represented by two EventOccurrences. One is the start EventOccurrence which specifies the beginning of the action of the ExecutionOccurrence and the other is the end EventOccurrence which specifies the ending of the action. An ExecutionOccurrence is an InteractionFragment.

Attributes:

methodName: String- The name of the method which should be called by the application specific code.

startConstraint:String- This specifies the startConstraint of the action and this usually specifies the input parameters of the method used.

finishConstraint: String- This specifies the finishConstraint of the action and this usually specifies the handling of the result produced by the method.

Associations:

start:EventOccurrence[1]- References the EventOccurrence that specifies the start of the action.

finish:EventOccurrence[1]- References the EventOccurrence that designates the finish of the action.

startingCombinedFragment:EventOccurrence[0..1]- References the Combined Fragment that is started by this ExecutionOccurrence.

endingCombinedFragment:EventOccurrence[0..1]- References the Combined Fragment that is finished by this ExecutionOccurrence.

Constraints:

The startCombinedFragment and the endCombinedFragment must be on the same Lifeline

9. InteractionOperand

An InteractionOperand is contained in a CombinedFragment. It always represents one operand of the expression given by the enclosing CombinedFragment. An InteractionOperand is an InteractionFragment which may be guarded by an InteractionConstraint. Only those InteractionOperands with a guard that evaluates to true at some point in the Interaction would be considered for production of traces for the enclosing CombinedFragment. Traces define the semantics of Interactions as given by the UML 2.0 superstructure. If no guard is present this is taken to be a true guard. InteractionOperands always contain an ordered set of InteractionFragments. These InteractionFragments are ordered vertically in any Interaction Diagram. The geometrical position of the InteractionFragment is given by the topmost vertical co-ordinate of its contained EventOccurrences.

Associations:

fragment:InteractionFragment[*]- Specifies the InteractionFragment enclosed by the operand.

guard:InteractionConstraint[0..1]- Specifies the constraint of the operand.

Constraints:

The guard must always be placed above the EventOccurrence that will become the first EventOccurrence within this InteractionOperand. The guard must contain only references to values local to the Lifeline on which it resides, or values global to the whole Interaction.

Attributes:

type:Boolean- Determines the truth value of the Operand

10. CombinedFragment

A CombinedFragment defines an expression of interaction fragments. A CombinedFragment is always defined by an interaction operator and corresponding interaction operands. CombinedFragment is a specialization of InteractionFragment.

Attributes:

interactionOperator:InteractionOperator: This specifies the operation that defines the semantics of the combination of interactionFragments.

Associations:

cfragmentGate:Gate[*]: Specifies the Gate that forms the interface between a combinedFragment and its surroundings.

operand:InteractionOperand[1..*]- The set of operands of the CombinedFragment.

startAction:ExecutionOccurrence[1]- The Action executed before this fragment.

endAction:ExecutionOccurrence[1]- The action executed after this fragment.

Constraints:

If the interactionOperator is neg,loop or op, there can be only one operand.

Semantics:

The semantics of CombinedFragment depends on the attribute of the interactionOperator.

Alternative: This means that several paths are possible to follow the interaction and agents have to choose at most one to continue. Guards are associated with different alternatives. Hence, the alternative that is chosen is the alternative that has its guards evaluated to true. In case no guard is evaluated to be true, no alternative is chosen. In such a case, the alternative CombinedFragment is not executed.

Option:

The Option operator considers only one path in the CombinedFragment. If the conditions associated with this path is evaluated to true, then that path is executed otherwise nothing happens and the interaction continues after this CombinedFragment. The Option operator can be represented as a CombinedFragment with an Alternative operator and two paths: the first one contains the set of messages that are executed if the conditions are satisfied and the second one is empty corresponding to the conditions not satisfied.

Break:

The Break operator defined a breaking scenario that stops the current execution and executes the scenario in the Break CombinedFragment. The broken current execution will not be resumed.

Parallel:

The parallel operator depicts the parallel execution of paths in any order. It allows representation of concurrent sending of messages.

Weak Sequencing:

The Weak Sequencing operator means that the messages within this CombinedFragment on the same Lifeline are ordered but it is not safe to make the

same assumption about the ordering of messages coming from different Lifelines in the same CombinedFragment.

Strict Sequencing:

This operator refines the Weak Sequencing operator and ensures that all messages in the CombinedFragment over all Lifelines are ordered vertically.

Negative:

This operator describes the set of messages that will be considered as invalid in the interaction.

CriticalRegion:

The CriticalRegion operator implies that the sequence of messages in the CriticalRegion should occur atomically. This sequence of messages cannot be interleaved with any other sequence of messages. This is akin to the critical region of Distributed Systems.

Ignore:

This represents the set of messages that should be ignored during the interaction.

Assertion:

The Assertion operator indicates that the sequence of messages in the CombinedFragment is the only acceptable sequence of messages for the current state of the interaction.

Loop:

The Loop operator allows designers to represent that an ordered set of messages need to be repeated a certain number of times. Upper bounds, lower bounds or a Boolean

expression could be used for the same purpose. As long as these conditions are satisfied, the loop is executed and the messages are sent or received.

11. InteractionConstraint

An InteractionConstraint is a Boolean expression that guards an operand in a CombinedFragment. The InteractionConstraint also contains two expressions designating the maximum and minimum number of times a loop should execute.

Attributes:

minint:int[0..1]- The minimum number of times the loop should execute.

maxint:int[0..1]- The maximum number of times the loop should execute.

expression:String[1]- The Boolean expression which is evaluated.

Constraints:

- Both minint and maxint, must evaluate to a non-negative integer.
- maxint and minint can be specified only if the InteractionConstraint is associated with a loop CombinedFragment.
- maxint must always be greater than or equal to minint , if specified.

12. InteractionOperator

The InteractionOperator is just an enumeration designating the different kinds of operators for CombinedFragments. This operator enumerates the possible values for

the interactionOperator attribute for CombinedFragments. The literals for the operators are as alt, opt, par, loop, ignore, consider, neg, seq, strict, critical and assert.

Semantics:

The value of the InteractionOperator is important for the semantics of the CombinedFragment as seen above.

13. Gate

A Gate relates a Message outside an InteractionFragment to a Message inside an InteractionFragment. Gates are connected through Messages. Gate is a specialization of MessageEnd.

Constraints:

The Message leading to or from a Gate from within a CombinedFragment must correspond to the Message leading from or to the CombinedFragment on its outside.

Semantics:

Gates are named either explicitly or implicitly. Gates may be identified either by the name of the Gate if specified, or by a constructed identifier.

Though the meta-model has been based on the meta-model for the UML 2.0 superstructure, the classes GeneralOrdering and Part-Decomposition have been deleted because they addressed specific constructs of UML not required for the meta-model developed. The meta-classes NamedElement and ConnectableElements have been deleted as they provided for generalization of classes, which was again not

required for developing the meta-model. Based on the FIPA specifications for Interaction Protocols, the attributes 'language', 'ontology' and 'variables' were added to the class Interaction. The class Message is an abstract class because of the incorporation of communicative acts[67] defined by FIPA. For every communicative act, there exists a subclass of Message in the meta-model and these classes need to be instantiated. The Messages referred to in the meta-model are always complete, i.e. both the Sender and the receiver have to be known for the Message to be sent across. Message communication, how ever could be either synchronous or asynchronous.

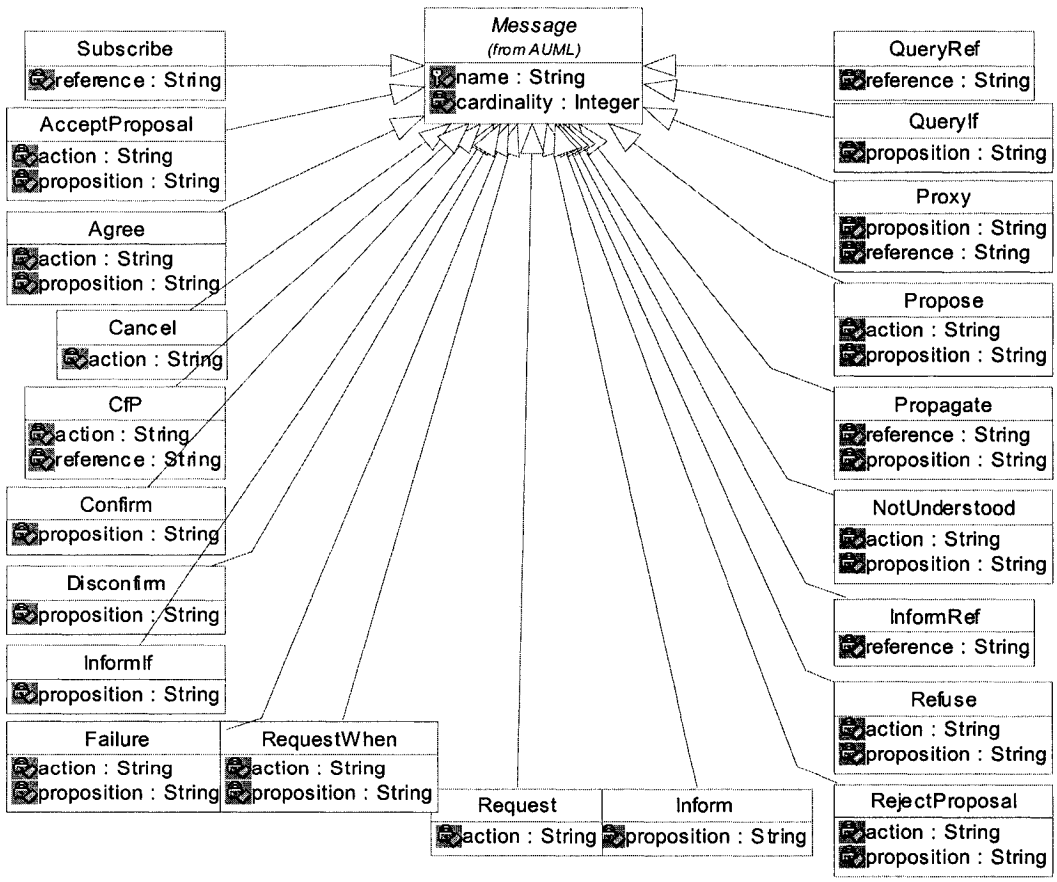


Fig 10: The abstract class Message and its included subclasses

14. AddRole

This class enables multiple roles over a single Lifeline. The agent over the same span of time can take on different roles at different instances of time. This is also termed as role cardinality.

Attributes:

RoleName:String- The name of the role to be played.

Cardinality:Integer- The number of roles being played onto the same Lifeline.

AgentIdentifier: String- The name of the agent taking on the role.

Constraints:

The lifetime of a role is always lesser than or only as long as the lifeline. A role cannot live beyond the longevity of the lifeline.

15. Continue:

This is an abstract class. It defines continuations of different branches of CombinedFragment.

Associations:

from:EventOccurrence[1]- Specifies the EventOccurrence that denotes the end of activity from where continuation would next occur.

to:EventOcurrence[1]- Specifies the EventOccurrence that denotes the activity of the occurring continuation

4. Developing the meta-model

4.1 Introduction to the Eclipse Modeling Framework

There are many tools that aid the use of formal models and development of meta-models in Software Engineering. These tools mostly follow the Model Driven Architecture (MDA) and mostly support the Unified Modeling Language (UML). The Eclipse Modeling Framework (EMF) is one of these tools. EMF is based on Eclipse [15], which is a platform-independent, open-tool platform. The Eclipse Platform provides building blocks and a foundation for constructing and running integrated software-development tools. It allows tool builders to independently develop tools that can integrate seamlessly with other tools and applications.

The Eclipse Modeling Framework (EMF) is a JAVA application for building tools and other applications based on a structured model. Using a combination of UML diagrams such as Class diagrams, Collaboration diagrams and so on, a complete model of an application can be specified. This model could then be used as an input to generate part of or the entire application. EMF basically uses XML Metadata Interchange (XML) as its canonical form of model definition. The core EMF framework provides run time support to create JAVA implementation classes for a given model. The second layer or the EMF.edit framework provides extends and builds on the core framework adding support for generating adapter classes that enable viewing and command based editing of a model.

EMF was chosen because of the following reasons:

- EMF is well documented and well supported. A host of plug-ins are available for model development.
- EMF provides a graphical user interface. Hence Interaction Protocols can be defined by instantiating meta-model instances.

4.2 Using EMF to generate an Interaction Protocol

The meta-model is loaded into the Eclipse Modeling Framework in a tree-based structure. The core framework provides support for the classes while the overlying EMF.edit framework provides support for the attributes and associations for each of the classes. This enables the AUML meta-model to be displayed as a tree-based graphical Interface. Any user-defined Interaction Protocol can now be fed into the system as an instantiation of the AUML meta-model. Instances of the model and the meta-model objects generated using the graphical interface are then serialised to be able to store and transfer them. As discussed before, EMF serializes objects using XMI. XMI is a widely used serialisation standard that supports UML primitives.

The AUML meta-model described in the previous chapter was designed using Rational Rose[18], a Computer Aided Software Engineering (CASE) tool. A Rational Rose file can be imported into EMF and then serialised into the XMI format. Each of the classes, their attributes and associations are loaded and stored in the XMI format. Any later

manipulation in the model, is reflected in the XMI code as well and updated automatically by the EMF framework. This reduces developer effort into updating both the model and its resultant code.

Once serialised, the UML classes can then be converted into JAVA code. Each execution occurrence is labelled as a method name for a class. The details of method execution are then filled in along with both input and output parameters. Hence, a meta-model instance representing the Interaction Protocol can be accessed directly via the JAVA classes and JAVA objects and synchronization skeleton in the form of JAVA code is obtained for every instance of the meta-model entered.

Thus, program skeletons can be generated for any Interaction Protocol Diagram.

4.3 Extensibility of the meta-model

Extensibility is a vital aspect of software systems. Software system environments are continually subject to change and these changes demand corresponding modifications in the software system. Extensibility refers in this context to the ability of creators of information objects to ***extend*** their documents.

Wikipedia defines Extensibility as follows. " Extensibility is a system design principle where the system implementation takes into consideration future growth. It is a systematic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of a new functionality

or through modification of existing functionality. The central theme is to provide for change while minimising impact to existing system functionality."

We make use of Packages for meta-model extensibility. Packages are used to define and structure both the meta-model and its extensions. Different aspects of a meta-element can be defined in different packages that makes it very simple to be edit/delete existing packages or to add a new package. A package allows for an element of a meta-model to be selected and specialized. A given meta-model element can be introduced in any one package and then extended to other packages that have a generalization relationship to it. UML 2.0 specifies that "Any package can have generalizations to other packages. This means that the public and protected elements owned or referenced by a package are also visible to its heirs and can be used in the same way as any element owned or referenced by the heirs themselves. A package can also own or reference other packages". This is a good approach against modeling extensibility by using subclasses as using subclasses provides for extensive multiple inheritance which contaminates very quickly as and when the meta-model is extended. Hence, package extension is an excellent way to support and structure meta-models.

The AUML meta-model too is structured in the form of a package. The specifications for Package have been adopted from the Kernel class of the UML 2.0 superstructure specification. The entire meta-model itself is stored as a package. This package in turn consists of other packages with which it shares a generalization relationship. The package FIPA consists of another package, Agent UML that contains the set of classes (UML

classes) and their associations with each other. The class Message has been defined as a package such that all its subclasses (the different communicative acts as specified by FIPA) have been defined within this package. Thus, any modifications, additions or deletions to the set of communicative acts would not affect the rest of the meta-model. The model has also been divided into different views for future convenience. There is a use case view that could possibly incorporate future use-case additions over the meta-model. There is also a deployment view that consists of all the components used in the model. Thus any addition or deletion of future components would not affect the rest of the packages. Assuming that a developer would later want to introduce new definitions of ACL onto the same meta-model, he can do so by building in the new ACL definitions in a separate package and later specify a generalization onto other packages already existing in the meta-model. Hence a developer can not only define his own package but also continue to use the already existing packages in the meta-model.

Extensibility in the meta-model is described in the diagram below. The yellow boxes denote packages where as the white boxes denote different components within those packages. The relationship between the different meta-model packages as well as the relationship of each package to the main meta-model itself is shown. Packaging makes the meta-model extremely modular, re-usable and scalable without affecting much of its present day performance.

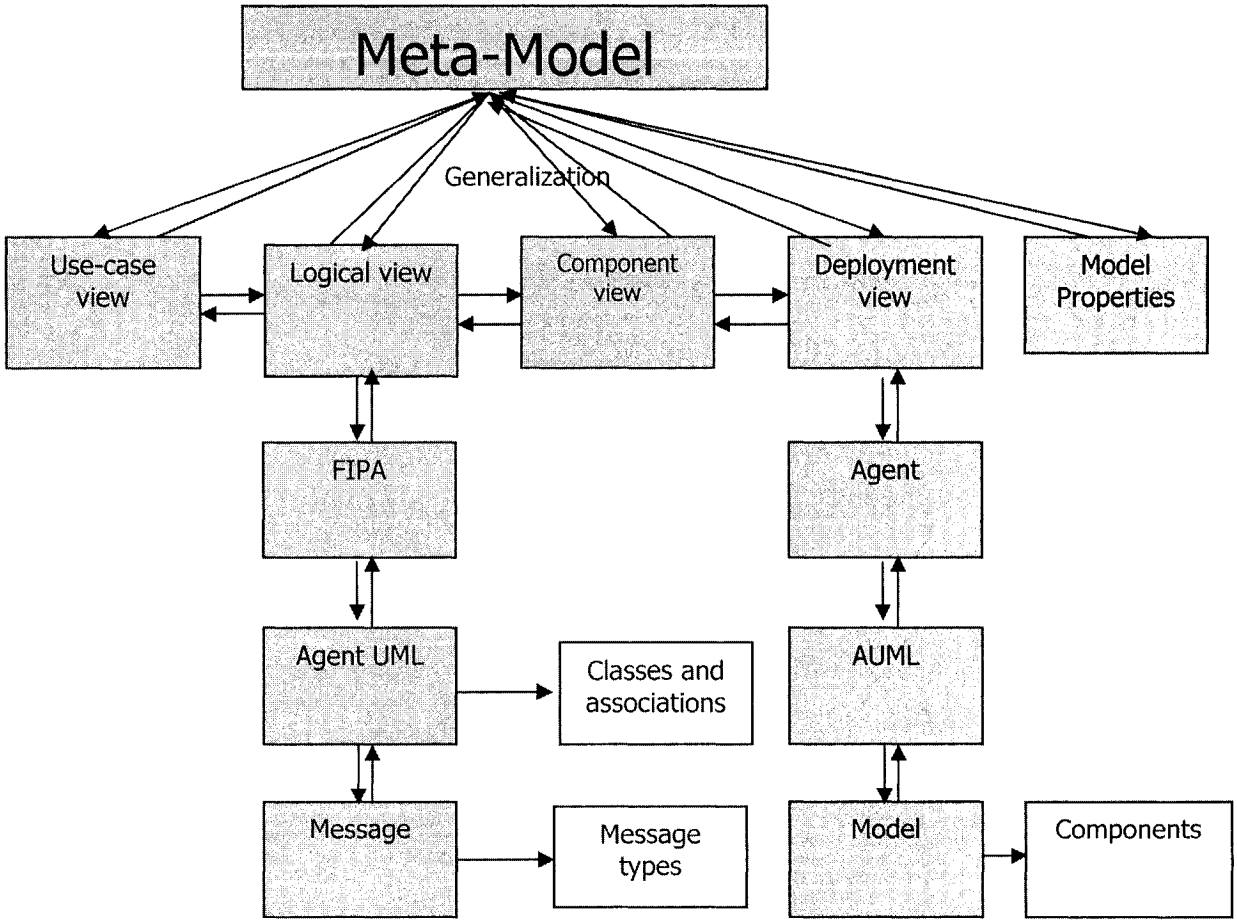


Fig 11: Extensibility of the meta-model

4.4 Validation of the meta-model

Validation, in simple English is defined as finding or testing the truth of something. Validation is also considered as the process of checking if something satisfies a certain criterion. Here, validation is the process of demonstrating that the generated model produces reliable output as the capacity of the model is proved upon generation of credible and verifiable results. We would label validation as "Proof of attainment of a certain level of accuracy with respect to the results of the meta-model". Validation in turn verifies the intended semantics of the model. Though we do not prove the correctness of the meta-model in this dissertation, we do try to attain partial-correctness by means of validation.

We look into the AUML specifications as specified by FIPA and prove that the meta-model tries to achieve the necessary functionality as given in the FIPA specifications. We look into the definition of each term as provided by the FIPA specification [58] and then establish that our meta-model achieves exactly the same functionality as described by the specification.

The Interaction Protocol Library Specification, states that the semantics of any Protocol Diagram, can be represented by an Interaction, which is a set of messages exchanged amongst different roles within collaboration to affect a desired behaviour of other Agent Roles or agent instances. We model exactly the same behaviour in the meta-model by making an Interaction Fragment the basic semantic unit of any Protocol Diagram. An

'Interaction Fragment' is modeled as the general unit of behaviour and can refer to any interaction that takes place in the model.

The FIPA specification defines an 'Interaction' as a sequence of communications that specifies a communication between a sender role and a receiver role. The meta-model defines an 'Interaction' as a unit of behaviour that deals with the observable exchange of information, usually between a sender Lifeline and a receiver Lifeline.

An agent Lifeline, as defined by the FIPA specifications, denotes the time period of the agent , as long as it exists. In short, it defines the lifetime of a particular Agent Role. Only during this period can an agent participate in a protocol. The meta-model defines a Lifeline as representing one or more agents. The Lifeline in the meta-model can represent either the specific agent itself or one or more roles that the agent is participating in. This way, the Lifeline in the meta-model not only supports agent lifetimes, but also provides inherent support for multiple roles as well. The Lifelines in the meta-model may also be split to describe AND / OR parallelism. The FIPA specification states that the order of messages along a Lifeline is important as they represent the order of interactions occurring in the Protocol Diagram. We map the same concept onto the meta-model. Each message being sent by a Sender and being received by the receiver is represented as an Event Occurrence. These Event Occurrences represent specific moments of time with which certain actions are associated and Event Occurrences represent the basic semantic unit of Interactions and the sequences of EventOccurrences make up the meaning of Interactions in their entirety.

Messages in the FIPA specification is defined as the communication from one AgentRole to another that conveys information with the expectation that the receiving AgentRole would react according to the semantics of the communicative act. The protocol specification says nothing about the implementation of the same. Mapping the same concept onto the meta-model, a Message in the meta-model defines a communicative act between a sender and a receiver, each of which are associated with a sending event and a receiving event. A Message in the meta-model always associates itself with two Event Occurrences, a Sending EventOccurrence and a Receive EventOccurrence , each of which are encapsulated within a MessageEnd. A MessageEnd denotes either the start or the end of a message. The FIPA specification states that any message within a Protocol Diagram can be associated with different options-such as cardinality, guard condition and argument list. The same sets of conditions have been mapped on to the Message class in the meta-model such that each of these could be set during runtime. Complex messages as defined by the FIPA specifications, such as in parallel messages could be represented in the meta-model by passing multiple messages associated with multiple EventOccurrences at the same time instant.

The FIPA specification library however, does not specify any formal specifications for Combined Fragments. It just states that a Combined Fragment defines the splitting and merging of message paths, and this is simply re-defined to be a 'Thread of Interaction' between paths. How ever, the FIPA library does specify the operators for a Combined Fragment as in Alt,Break,Opt,Par, Seq,Weak,Neg and Assert as we have seen before. Redefining these same concepts onto our meta-model, by slightly specializing into the

definition of a Combined Fragment, we define a Combined Fragment to be an expression of different Interaction Fragments. Each of these Interaction Fragments are defined by the associated Interaction Operand-Alternative, Break, Parallel, Negative, Assert, Weak Sequencing, Strict Sequencing, Option and Loop, which are in turn defined by the FIPA specifications. We make use of the FIPA specifications to define our Interaction Operands and in turn let the Interaction Operands re-define different Interaction Fragments that sum up to form a Combined Fragment. The FIPA specification associates a Guard value with every Combined Fragment which we specify as an Interaction Constraint in our meta-model. We define an Interaction Constraint to be a Boolean expression that guards an operand in a CombinedFragment.

The primitives described above are described in the FIPA Interaction Protocol Library Specifications, each of which have been applied in the meta-model. As the meta-model captures the inherent behavior of the different FIPA primitives, as defined by the FIPA specifications, we prove that the meta-model has been validated.

4.5 Usability of the meta-model.

The meta-model can be used to describe any Interaction Protocol. The protocol that is fed as input would be defined as an instance of the meta-model. Since a meta-model can specify instances, each of which are in turn models, a user-generated Interaction Protocol can be easily developed. The meta-model developed provides a graphical user interface. It is displayed on an editor as a tree-based structure. All that the developer / user who wishes to input an Interaction Protocol into the system needs to do is to choose options from the tree-based menu as he wishes. A sample screen shot of the menu is as shown .

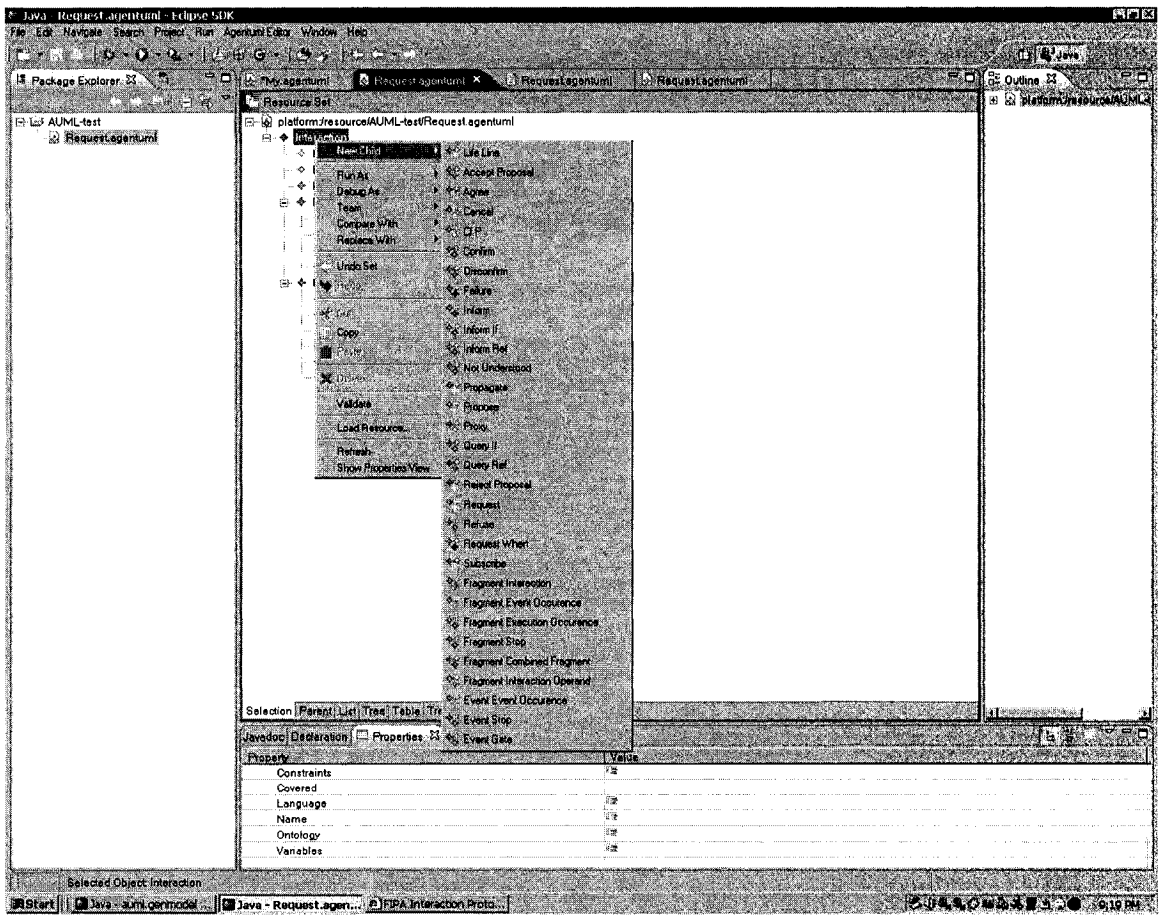


Fig 12: A sample screen shot of the generated editor

The basic starting unit for a model is 'Interaction'. Interaction is the basic unit of activity in the model. Since any Protocol defines the interaction occurring within by means of communication activity between two lifelines, 'Interaction' is normally chosen as the starting point for defining any new instantiations. 'Interaction' can in turn define other activities such as an 'Event Occurrence' or an 'Execution Occurrence' or other AUML primitives such as a Lifeline, Interaction Operand , Message Types and so on either as a new child or a new sibling. A new child establishes a level of hierarchy between the parent and child, whereas 'New Sibling' places events (or primitives) within the same hierarchy.

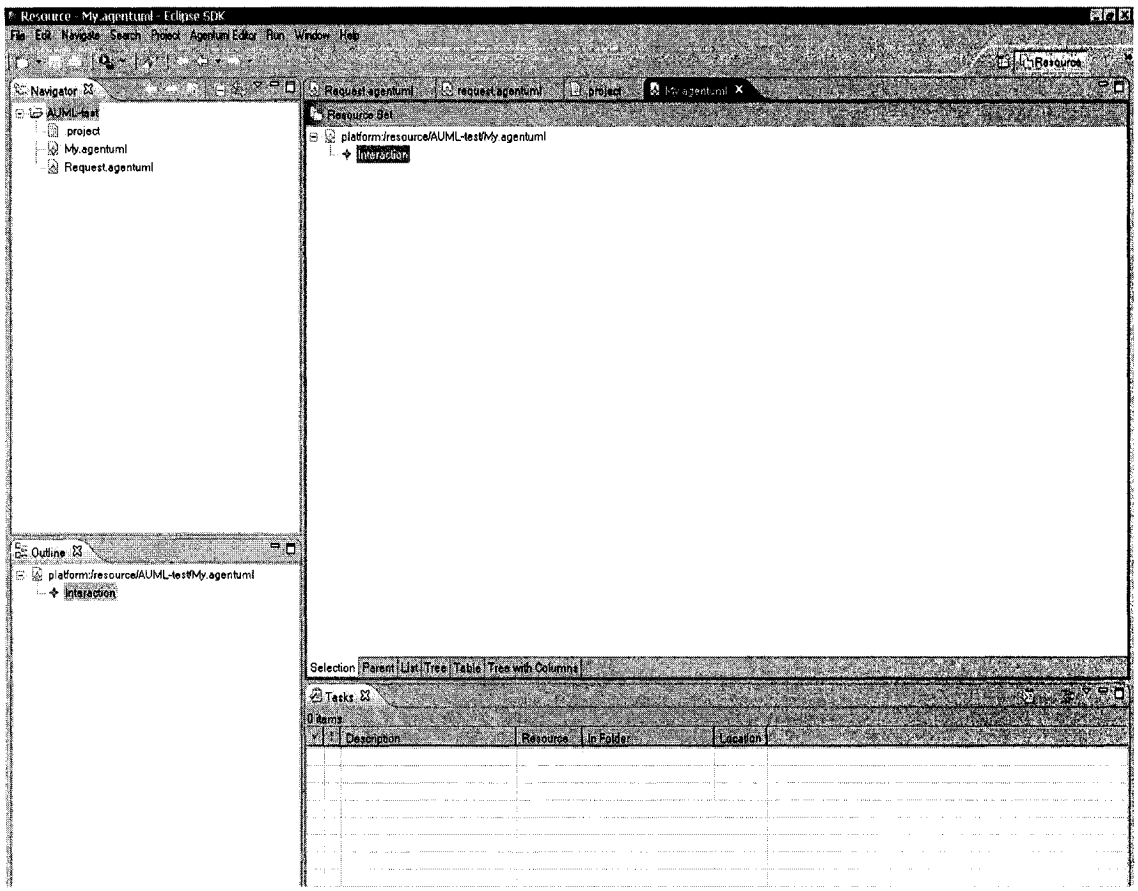
Any primitives entered as objects of the meta-model instance can have their properties defined by the user. The user could define the name, the cardinality, the ontology, the language, the interaction operand from a list of choices and so on. This way, any data that needs to be defined as value attributes of the objects in the meta-model instance can be set to different user-defined values. These values can be edited from time-to-time by viewing the 'Properties' menu. Any labels in the main editor would be updated automatically each time any value attribute in the 'Property' view is changed. Similarly any reference to a child / sibling could be modified and updated at any instant of time. Hence, this ease-of-use enables easy input of Interaction Protocols in a serialized format enabling developers to generate synchronization skeletons for developing applications for agent platforms.

Let us now look into two sample use-cases wherein the meta-model is used to build a Protocol Diagram. First, we would consider the standard example being used elsewhere in this thesis, the FIPA Request Protocol. Secondly, we would consider a user-generated Protocol Diagram and look into how this diagram can be fed into the system using the meta-model.

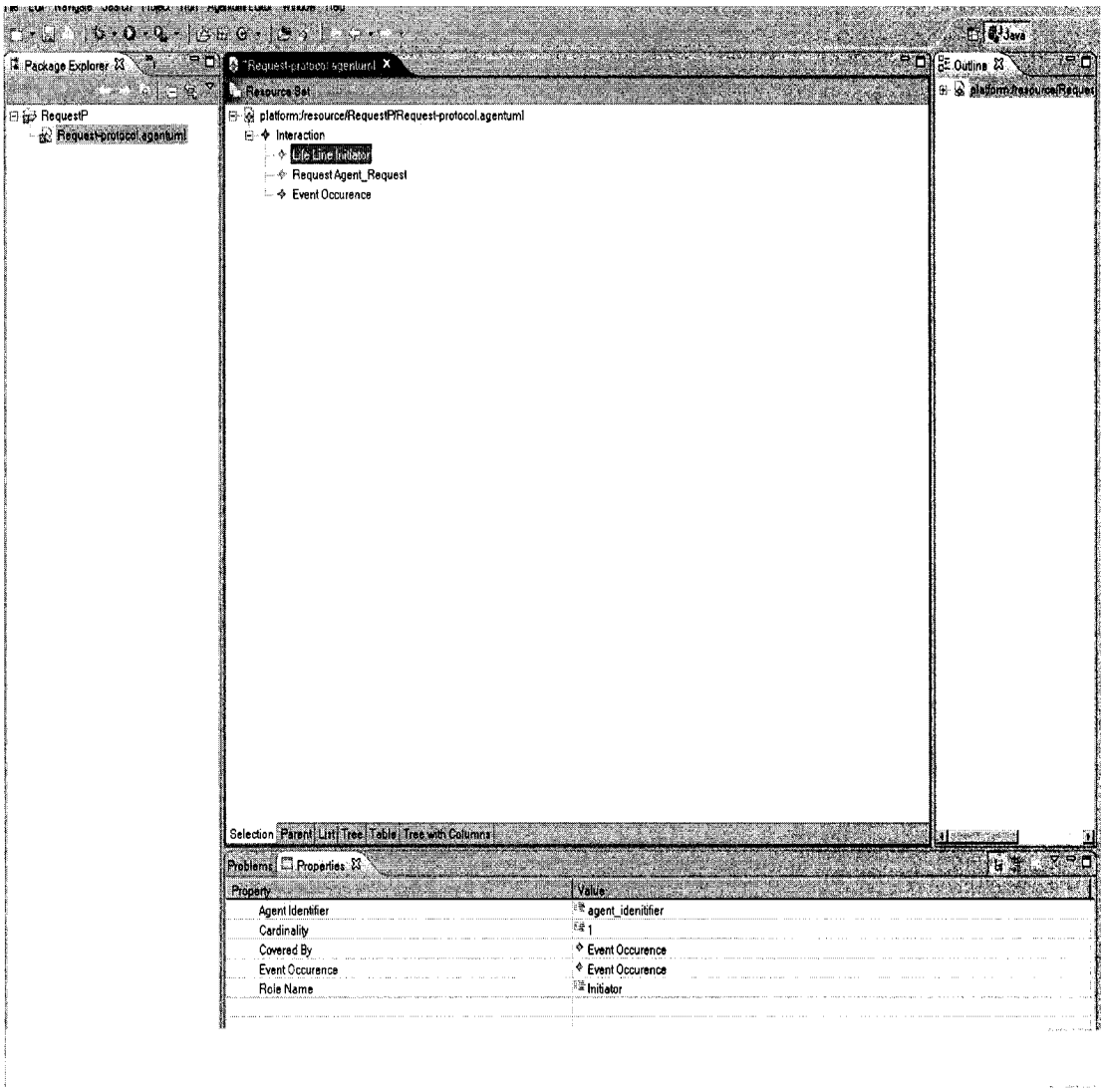
Case Study I: The FIPA Request Protocol.

The FIPA Request Protocol has been depicted in Chapter 2. Now us now take a step-by-step look as to how the same protocol could be fed into the system using the meta-model.

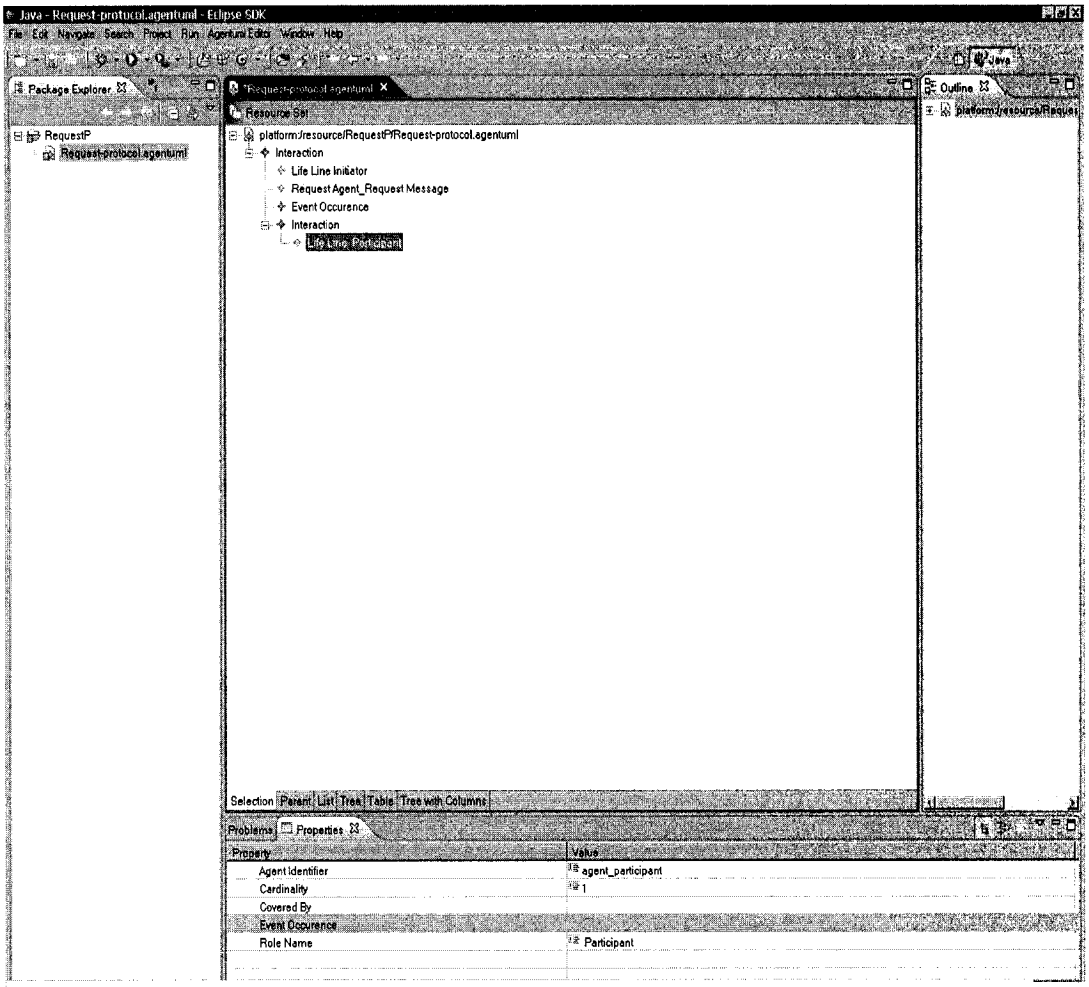
a. The user, as the first step towards building the Protocol Diagram chooses the primitive 'Interaction'. This option is available in the drop down menu and since the basic unit of any Protocol Diagram is 'Interaction', the same is chosen as the root modeling object in the user interface too.



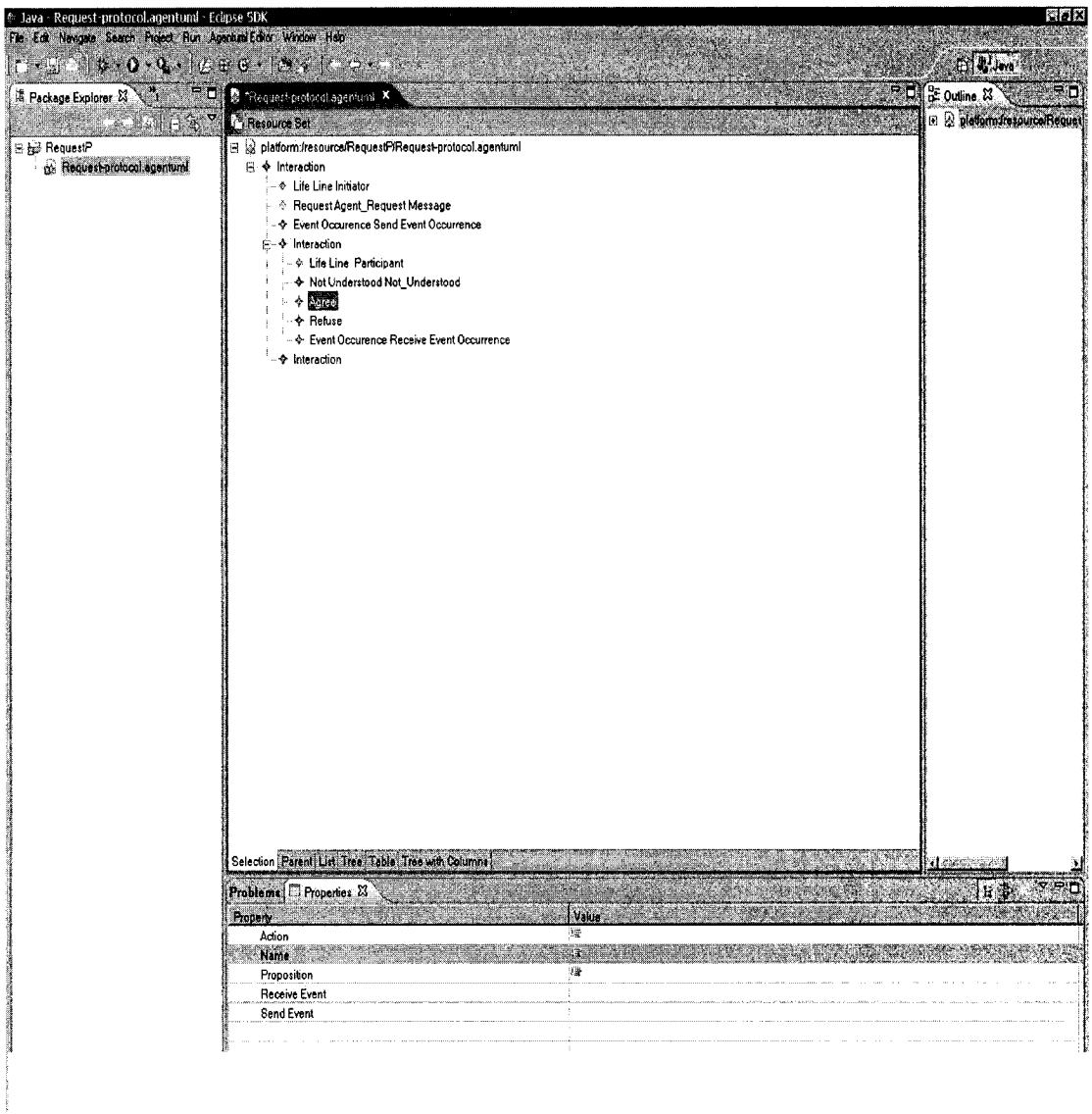
b. The next step would be to now define the rest of the Protocol Diagram. We first define a Lifeline along which the first Interaction would occur. We could define additional properties along this Lifeline that include 'Agent Identifier', 'Agent Cardinality', 'Covered By', 'Event Occurrence' and 'Role Name'. We set the Agent Identifier to 'Agent_Initiator', 'Agent Cardinality' to be 1 and 'Role Name' to be 'Initiator'. The first Interaction along this Lifeline would be the 'Request' message being sent. Since the sending of a message is modeled as an Event Occurrence, we define an Event Occurrence-'Request' along this Lifeline.



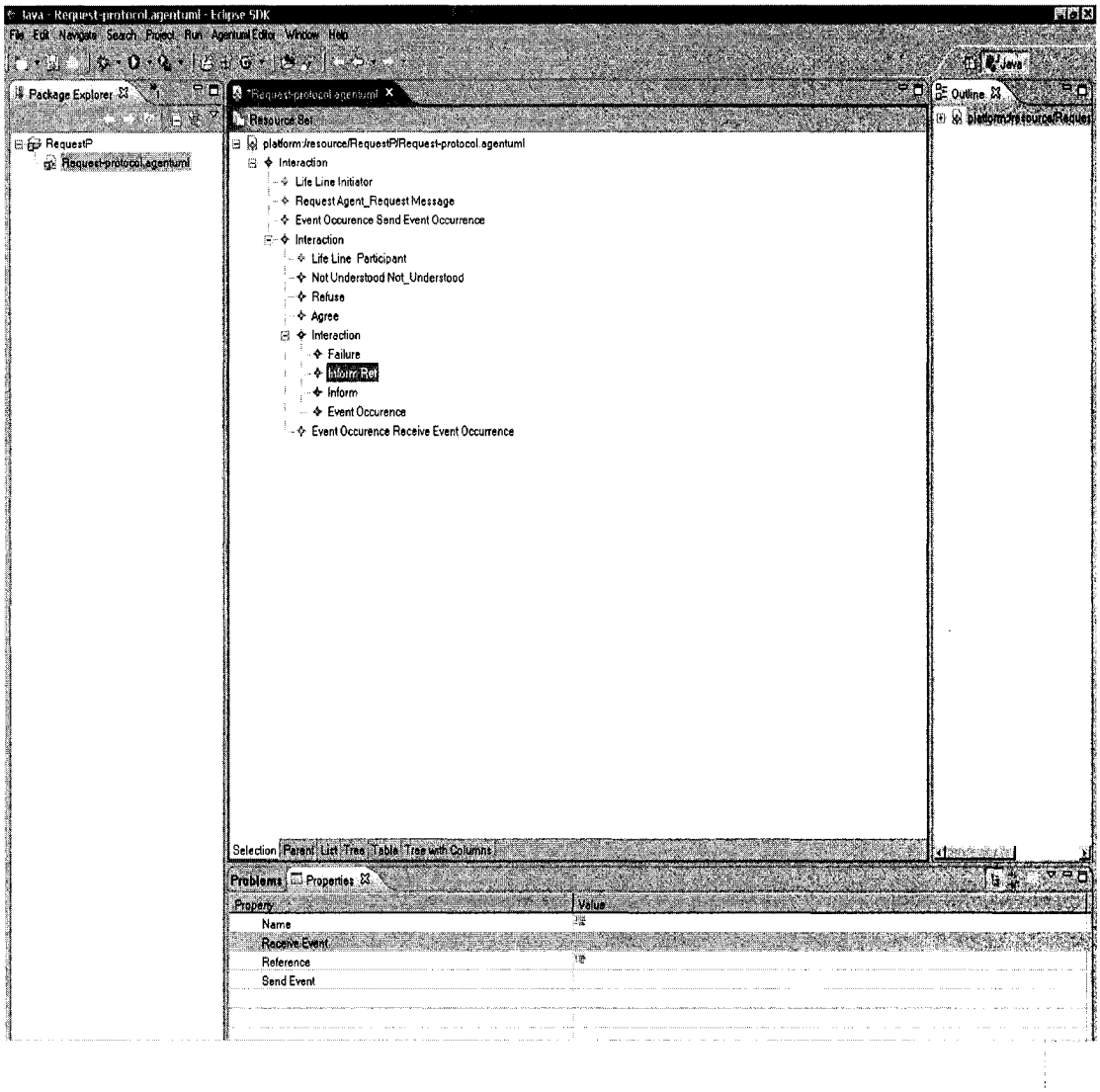
c. Now that we have defined a Lifeline along with the Event Occurrence – Request occurs, we next define a corresponding Participant Lifeline. This lifeline, like the Initiator lifeline is defined along another corresponding 'Interaction' event.



d. Once the Participant Lifeline has been defined, a set of three possible messages are listed-'Not Understood', 'Agree' or 'Refuse' based upon the action that the agent might choose to execute. As this is one choice out of a possible three, these messages point to the same Event Occurrence, establishing the fact that only one message could be sent as response. Each of these messages in turn have properties that could be defined- The Receiving Event Occurrence, Name, Cardinality and so on.



e. If the message sent is 'Agree', the Participant could in turn perform one of three actions- send a 'Failure', an 'Inform-Ref' or an 'Inform-If' message, each of which is an Interaction. Associating these messages with just one Event Occurrence indicates that only one out of the three messages could be sent back to the Initiator as response.



This way the FIPA Request Protocol is generated using the meta-model. Once the Protocol Diagram has been generated, the meta-model can also be used to generate JAVA code.

Case Study II: **A user-defined Interaction Protocol**

Consider a user-defined Protocol Diagram as shown below:

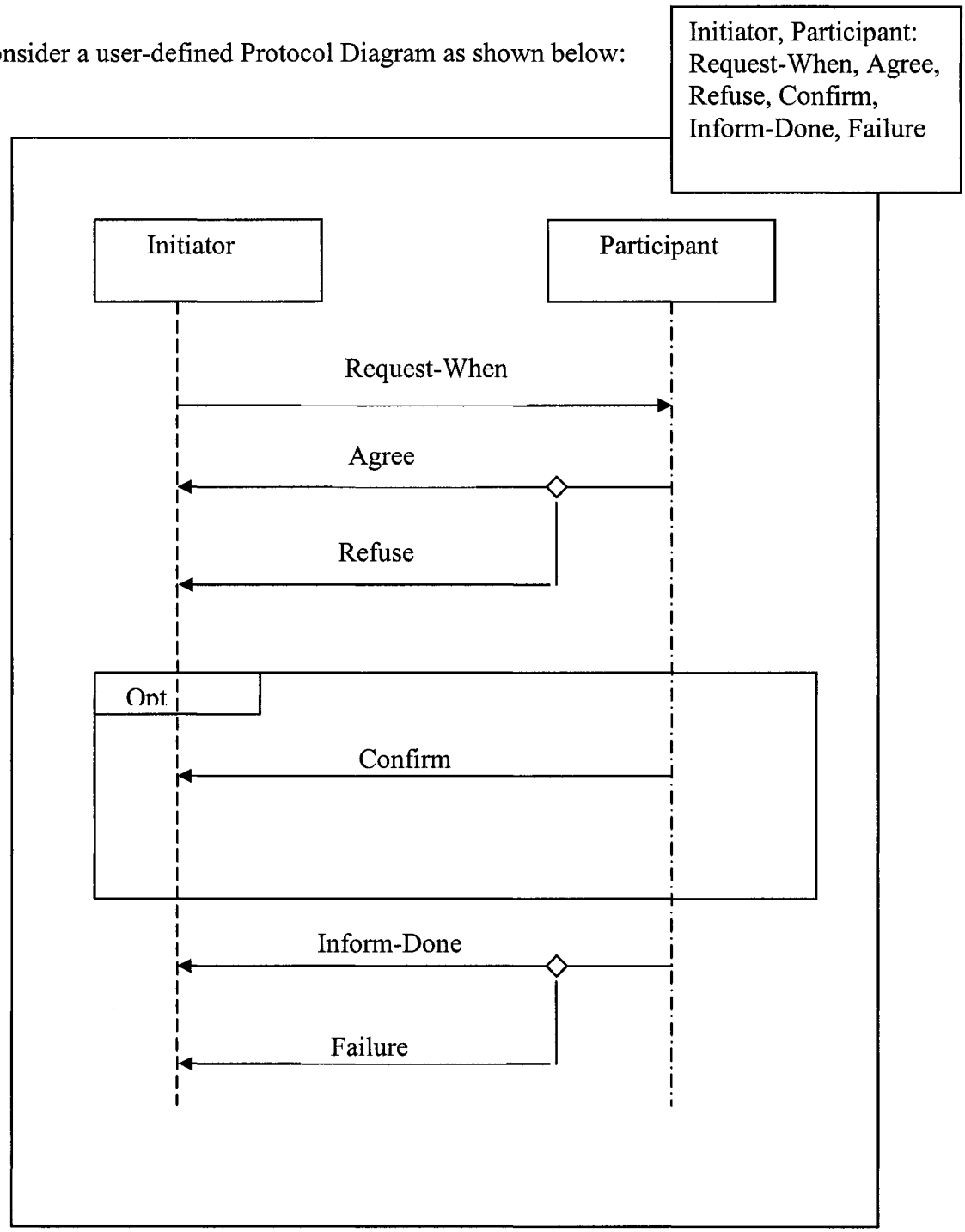
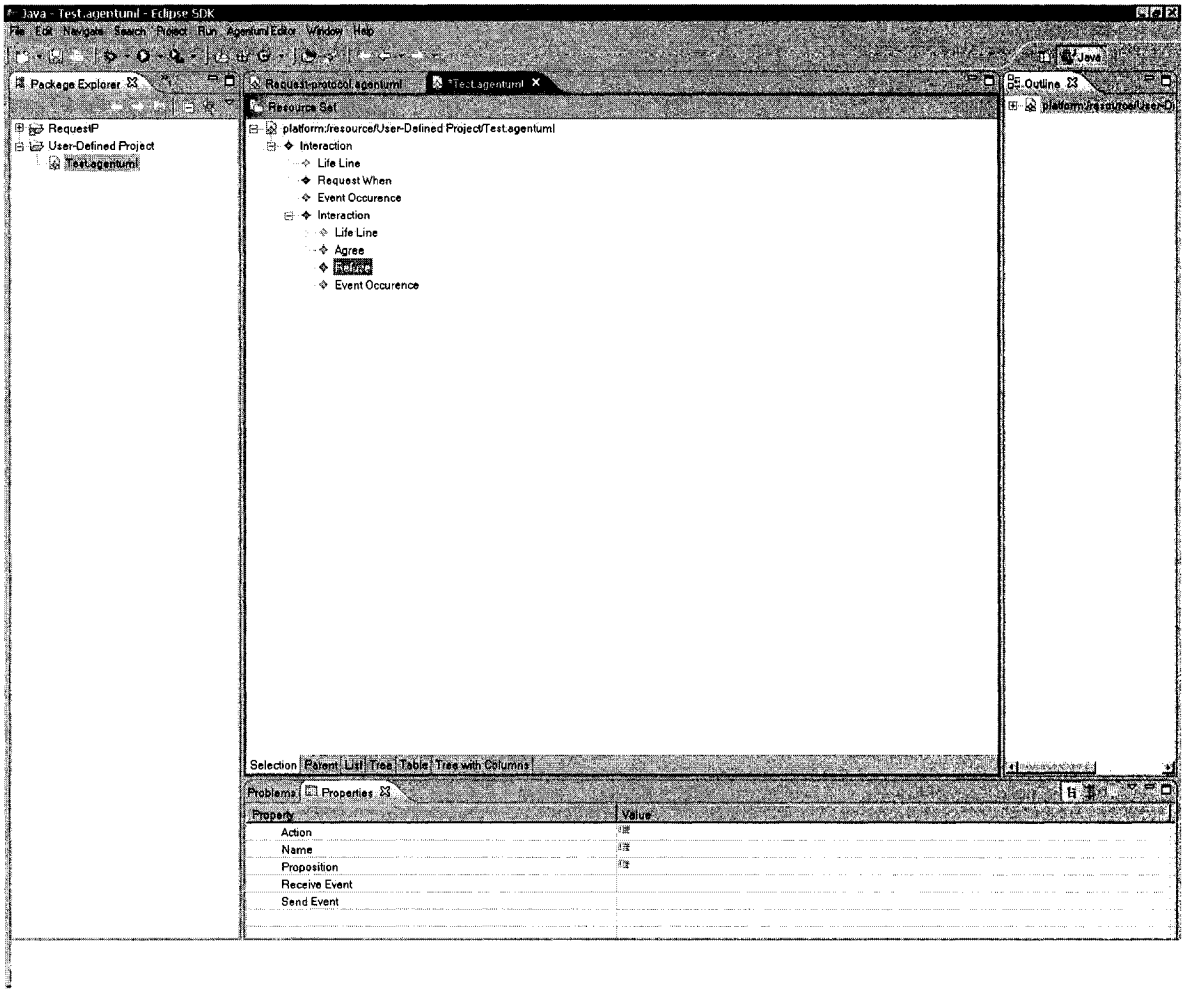


Fig 13: A user-defined Protocol diagram

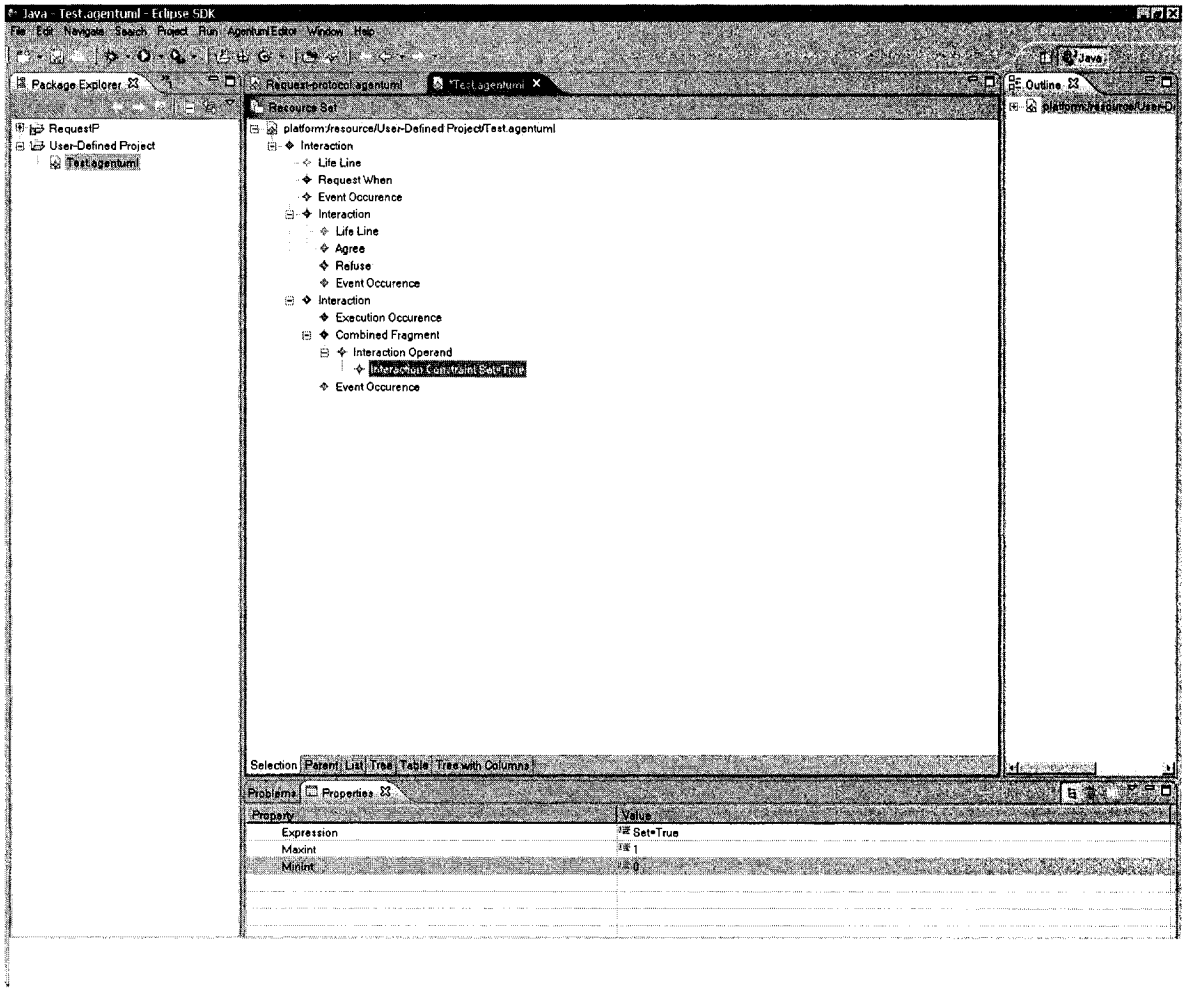
In this Protocol, the Initiator requests the Participant for some Information. The Participant can either Refuse to give the information to the Initiator it might Agree to the same. If it agrees, based upon whether a certain condition is true, the Participant might again confirm with the Initiator. Finally, the Participant might Inform the final result to the Initiator or might communicate a Failure.

This is represented using the meta-model as follows:

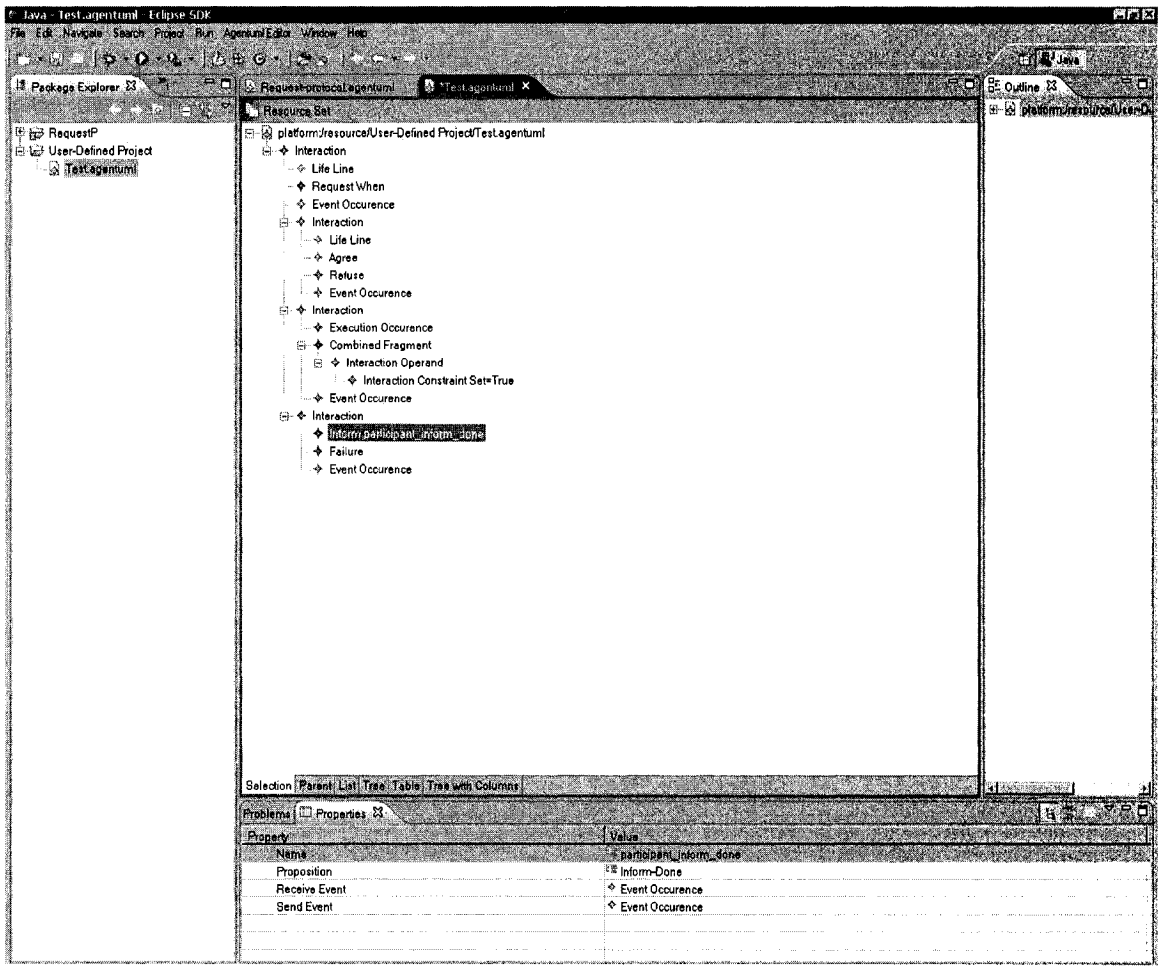
Step 1: Using 'Interaction' as the base model object, we build the communications between the two lifelines for the messages-Request-When, Agree and Refuse. The first communication sent from the Initiator Lifeline is modeled as an Execution Occurrence. Since the participant can respond back with only one of two different messages-Agree or Refuse, we associate only one Event Occurrence with the Participant Lifeline.



Step 2: Now we bring about the Option-Combined Fragment using the meta-model. We associate the 'Combined Fragment' with an ExecutionOccurrence. This Execution occurrence is associated with an Interaction Operand Opt through the Interaction Operator enumeration. Since any Combined Fragment is guarded, we set the guard in the meta-model to be a Boolean Condition (which is normally checked during run time). If the guard evaluates to true, the Combined Fragment is executed.



Step 3: Finally, the rest of the Protocol Diagram is built in by using the meta-model. The Participant Lifeline sends one of two messages as a response to the Initiator Lifeline – Inform Done or Failure depending upon its outcome. These messages are again associated with only Execution occurrence to signify that only one messages can be executed. The properties associated with the different model objects-such as cardinality, sending Event Occurrence, receiving Event Occurrence and so on can be set by the user in the meta-model at the time of building the Protocol.



Hence, the meta-model can be used to generate any user-defined protocol too. The features provided in the meta-model such as the graphical user-interface, drop down menus, property tabs and code generation modules aid in development of Protocol Diagrams without any prior knowledge of the same.

4.6 A thought on Composition

Composition refers to the process of enabling one single agent to run more than one role at the same time. In other words, it is the process of composing a single agent onto multiple roles. Though Composition has not been currently implemented as part of this dissertation and has been classified as future work to be carried on in this field, a small thought process on Composition is provided below.

The agent platform JADE provides inherent support for role composition. This can be explained as follows. JADE supports multiple behaviors. In other words, it can run multiple threads with one agent, wherein each thread can represent a different behavior (or if seen from an implementation point of view – a method of a behavior). This provides for inherent support for role composition. Each role can be viewed as a unique behavior. The scheduler can then schedule different roles such that the objective of composition can be achieved.

Composition implemented as co-routines.

This method involves user-input to a certain degree The agent platform JADE has an in-built scheduler that automatically schedules different agent behaviors in a round-robin fashion. It is nothing but one single agent running multiple threads for multiple behaviors and these behaviours are called in an orderly fashion one after the other, automatically by the scheduler application.

Each Execution Occurrence is generated as a method and hence, we have methods for different Execution Occurrences. This in turn is mainly nothing but a method generated for every sending of a message from one role to another. We would then need to build this program skeleton model into the JADE scheduler such that different roles can be scheduled onto one agent.

The answer to this question could be obtained by viewing each role as a behaviour. The user provides the different method names of each role (method name: role name) to the scheduler. So, at any moment, the scheduler can call each of these methods in a round robin fashion. But how ever, such a setting does not mimic a conversation. Hence, the application programmer interferes such that one method calls another. Hence when the scheduler implements these methods, the starting method (again given as input by the application programmer) is first executed, which in turn calls another method (possibly of another role) which is then executed. This recently executed method in turn calls the next method to be executed (again possibly of another role) and so on until the conversation ends. Hence composition could be achieved.

The main disadvantage of this method is that it is simpler and requires a significantly high amount of interference on the part of the application developer as the application developer manually feeds the method names of the Execution occurrences to the agent scheduler.

5. Conclusion

The work described in this thesis makes an effort to understand the semantics of Interaction Protocol Diagrams as described in AUML and generate program skeletons for the same using a validated, extensible model. The following was looked at in the course of work pursued:

- A meta-model for AUML was developed.
- There was an evaluation of tools that could be possibly used to develop the model.
- The generated model was validated.
- Provisions were built in for extensibility.
- A simple evaluation of the model was undertaken and a few sample program skeletons were generated.

This work mainly tries to generate synchronization skeletons for Interaction Protocol Diagrams. This in turn enables use of the agent ideology onto more complex applications across multiple platforms such that there can be seamless inter-agent communication. This also enables developers to input any Interaction Protocol into the system without being tied to a particular implementation language.

5.1 Future Work

'Research is always to be researched' as the saying goes. Although much has been done, a lot more needs to be achieved in this field. Future work would ideally comprise of:

- Agent Composition: Incorporating multiple roles onto a single agent. Although work on this front did take off, time constraints did not prevent full execution and implementation of Composition. This would give the AUML meta-model much more usability.
- Incorporation of the Object Constraint Language (OCL) to specify constraints in the meta-model. Although some work has been undertaken by the OCL group at Kent University in this regard, full implementation of an OCL toolkit needs to be undertaken.
- Finally, an editor that can take as input the different AUML primitives such as lifelines, messages and so on graphically and store them in a machine-understandable format would provide ease-of-use to even a novice who would want to work on Interaction Protocols.

6. References:

1. The website of the Eclipse Modeling Framework at www.eclipse.org/emf
2. M.P. Huget, and J. Odell,, "Representing Agent Interaction Protocols with Agent UML," *Proc. 5th Int'l Workshop on Agent-Oriented Software Eng.,* P. Giorgini, J. Müller, and J. Odell, eds., Springer-Verlag, 2004
3. S. Cranefield, and M. Pruns,, "UML as an Ontology Modeling Language," *Proc. Workshop on Intelligent Information Integration, 16th Int'l Joint Conf. Artificial Intelligence,* CEUR Publications, 1999
4. 12. Michael Wooldridge: Agent-based software engineering. IEE Proceedings - Software 144(1): 26-37 (1997)
5. M.-P. Huget,, "Representing Goals in Multiagent Systems," *Proc. 4th Int'l Symp. Agent Theory to Agent Implementation* (AT2AI–4), P. Petta and J. Mueller, eds., Austrian Soc. for Cybernetic Studies, 2004, pp. 588–593
6. B Burmeister, *Models and methodology for agent-oriented analysis and design*, in Working notes of workshop on Agent-Oriented Programming and Distributed Systems,(Fischer,Keds.) DFKI Doc. D96 –06
7. Nodine, Marian H., and Amy Unruh, "Facilitating Open Communication in Agent Systems: the InfoSleuth Infrastructure," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer, Berlin, 1998, pp. 281-296.
8. Kinny, David, and Michael Georgeff, "Modelling and Design of Multi-Agent Systems," Intelligent Agents III: Proceedings of the Third International workshop

on Agent Theories, Architectures, and Languages (ATAL'96), ed., Springer, Heidelberg, 1996.

9. Towards UML-based Analysis and Design of Multi-Agent Systems, Stephan Flake, Christian Geiger, Jochen M. Küster, Ibternational Symposium on Information Science Innovations in Engineering of Natural and Artificial Systems (ENAIS 2001), Dubai, March 2001.
10. A Knowledge Level Software Engineering Methodology for Agent Oriented Programming, Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos, pp. 648-655, *Proc. of Fifth International Conference of Autonomous Agents (Agents 2001)*, Montreal
11. Extended Modeling Languages for Interaction Protocol Design, Jean-Luc Koning, Marc-Philippe Huget, Jun Wei, and Xu Wang, pp. 93-100, *Proc. of Agent-Oriented Software Engineering (AOSE) 2001*, Agents 2001, Montreal
12. Michael Wooldridge: Agent-based software engineering. IEE Proceedings - Software 144(1): 26-37 (1997)
13. Iglesias, Carlos A., Mercedes Garijo, José C. González, and Juan R. Velasco, "Analysis and design of Multiagent Systems using MAS-CommonKADS," *Intelligent Agents IV: Agent Theories, Architectures, and Languages,* Munindar P. Singh *et al.* ed., Springer,Berlin, 1998, pp. 313-328.
14. Iglesias, Carlos A., Mercedes Garijo, and José C. González, ed., *A Survey of Agent-Oriented Methodologies* University Pierre et MarieCurie, Paris, FR, 1998.
15. The website of Computer Aided Software Engineering (CASE) at Carnegie Mellon at http://www.sei.cmu.edu/legacy/case/case_whatis.html

16. M.-P. Huget. *Agent UML class diagrams revisited.* In ernhard Bauer, I(. Fischer, J. Muller, and B. Rumpe, editors, Proceedings of Agent Technology and Software Engineering (AgeS), Erfurt, Germany, October 2002
17. Marc-Philippe Huget-Generating Code for Agent UML Sequence Diagrams
18. The home of Rational Rose at www-306.ibm.com/software/rational/
19. B. Bauer, J.P. Milllet, and J. Odell. *An extension of UML by protocols for multiagent interaction.* In International Conference on MultiAgent Systems (ICMAS'00), pages 207-214, Boston, Massachussetts, july, 10-12 2000.
20. M.-P. Huget. An application of agent uml to supply chain management. Technical Report ULCS-02-015, Department of Computer Science, University of Liverpool, 2002
21. M.-P. Huget. Model checking Agent UML protocol diagrams. Technical Report ULCS-02-012, Department of Computer Science, University of Liverpool, 2001
22. J.M.Bradshaw-Software Agents, Menlo Park, *CA: AAAI Press*, 1997
23. M.-P. Huget, J. Odell,, and B. Bauer,, "The AUML Approach," *Methodologies and Software Engineering for Agent Systems*
24. The specification for the meta-model of the UML 2.0 superstructure at the website of the Object Management Group that can be found at http://www.omg.org/docs/ptc/04-10-02.pdf
25. Barbara Hayes Roth An Integrated Architecture for Intelligent Agents. SIGART Bulletin 2(4): 79-81 (1991).
26. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115-152, 1995

27. Durfee, E.H., Lesser, V.R., and Corkill, Distributed Problem Solving. *The Encyclopedia of Artificial Intelligence, Second Edition*, John Wiley and Sons.. January 1991

28. Katia P. Sycara: Multiagent Systems. AI Magazine 19(2): 79-92 (1998)

29. Yannis Labrou and Tim Finin. Yahoo! as an ontology | using yahoo! *categories to describe documents*. In CIKM '99. Proceedings of the Eighth International Conference on Knowledge and Information Management, pages 180-187. ACM, 1999.

30. LEVIATHAN by Thomas Hobbes 1651

31. Finin,T. et al., "KQML as an Agent Communication Language", in Software Agents, J.Bradshaw ed., MIT Press, 1995

32. . Burkhardt, A., *Speech Acts, Meanings and Intentions. Critical Approaches to the Philosophy of John R. Searle*, Berlin/New York. de Gruyter (1990), 29-61

33. Tim Finin, UMBC, 1994 . KQML, A Language and Protocol for Knowledge and Information Sharing DAI Workshop July 1994

34. The website of the Foundation for Inter-Operable Agents at www.fipa.org

35. T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199-220, 1993

36. The website of the Ingenias meta-model research team at ingenias.sourceforge.net.

37. S. Cranefield, and M. Pruns,, "UML as an Ontology Modeling Language," *Proc. Workshop on Intelligent Information Integration, 16th Int'l Joint Conf. Artificial Intelligence,* CEUR Publications, 1999.

38. J. Odell, H. Van Dyke Parunak,, and B. Bauer,, "Extending UML for Agents," *Proc. Agent-Oriented Information Systems Workshop, 17th Nat'l Conf. Artificial Intelligence,* G.Wagner, Y. Lesperance, and E. Yu, eds., ICue Publishing, 2000

39. en.wikipedia.org

40. Conversation Policies (1999) Mark Greaves, Heather Holmback, Jeffrey Bradshaw Mathematics and Computing,Issues in Agent Communication

41. Nicholas R. Jennings: An agent-based approach for building complex software systems. Communications of the ACM_(4): 35-41 (2001)

42. Layered Approach for Modelling Agent Conversations (2001) Mariusz Nowostawski, Martin Purvis, Stephen Cranefield

43. K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997. ISBN: 3-540-60943-1

44. Agent UML: A Formalism for Specifying Multiagent Interaction Systems Bernhard Bauer, Jörg P. Müller, James Odell

45. . Using Colored Petri Nets for Conversation Modeling (1999) R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, Issues in Agent Communication

46. The specification of the Request Protocol on the FIPA website at www.fipa.org/specs/fipa00026/PC00026D.pdf

47. The NZDIS Project: an Agent-Based Distributed Information SystemsArchitecture (2000)Martin Purvis, Stephen Cranefield, Geoff Bush, et al,HICSS

48. B. Bauer. *Extending UML for the Specification of Interaction Protocols.* Submission for the 6th Call for Proposal of FIPA and revised version part of FIPA 99, 1999

49. The website of the Modeling Driven Architecture division of the Object Management Group (OMG) at http://www.omg.org/mda/

50. The Unified Modeling Language Reference Manual (Addison-Wesley Object Technology Series) by James Rumbaugh, Ivar Jacobson, Grady Booch

51. The Website of the Unified Modeling Language group at www.uml.org

52. Ontology definition languages for Multi-Agent Systems (2002) -L.Botelho, J.Bento and L.Mota

53. J. Odell, H. Van Dyke Parunak, and C. Bock. Representing agent interaction protocols in UML. In OMG Document ad/99-12-01. Intellicorp Inc., December 1999

54. Representing Social Structures in UML (2001) H. Van Dyke Parunak, James J. Odell. Proceedings of the Fifth International Conference on Autonomous Agents

55. B.Bauer. UML Class Diagrams Revisited in the Context of Agent-Based Systems. Lecture Notes In Computer Science; Vol. 2222. Revised Papers and Invited Contributions from the Second International Workshop on Agent-Oriented Software Engineering II , 101 - 118 , 2001

56. S. Cranefield and L.Ehler. Executing Agent UML Diagrams, Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2 (AAMAS'04)

57. Representing Agent Interaction Protocols in UML," James Odell, H. Van Dyke Parunak, Bernhard Bauer, *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge eds., Springer-Verlag, Berlin, pp.121–140, 2001.

58. FIPA Interaction Protocol Library Specification from www.fipa.org at http://www.fipa.org/specs/fipa00025/XC00025E.pdf

59. Michael J. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

60. Marie-Pierre Gervais & Florin Muscutariu, Laboratoire d'Informatique de Paris: A UML Profile for MASIF Compliant Mobile Agent Platform, OMG's second workshop on UML for Enterprise Applications, 2001

61. Modeling early requirements in Tropos: a transformation based approach, Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos, pp. 67-75, *Proc. of Agent-Oriented Software Engineering (AOSE) 2001*, Agents 2001, Montreal

62. Burmeister, B., ed., *Models and Methodology for Agent-Oriented Analysis and Design* 1996.

63. Wooldridge, Michael, Nicholas R. Jennings, and David Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," International Journal of Autonomous Agents and Multi-Agent Systems, 2000.

64. Maes, P. *Agents that reduce work and information overload.* Communications of the ACM, 37(7):30-40, 1994.

65. ROMAS: A role based modeling method for multi-agent systems: Qi Yan, Li-Jun Shan, Xin-Jun Mao and Zhi Chang Qui, National Univ. of Defense Technology, China.

66. Greaves, M., Holmback, H. and Bradshaw, J. (1999) What is a Conversation Policy? Proceedings of the Workshop on Specifying and Implementing Conversation Policies (Agents '99), Seattle, WA, pp. 1-9.

67. The FIPA Communicative Act Specification of the FIPA website at http://www.fipa.org/specs/fipa00037/XC00037H.html

Appendix

Code generated by the meta-model for the FIPA Request Protocol:

The meta-model generates two levels of code- an Ecore level of code and an Eimpl level of code. The Ecore code basically defines only the function names being called while the Eimpl code gives the implementation of these functions too. For purposes of simplicity, we have included only the Ecore code in the Appendix

```
package request_code;

import junit.framework.TestCase;

public class CodeType extends TestCase {

	public static void main(String[] args) {
		junit.awtui.TestRunner.run(CodeType.class);
	}

	public CodeType(String name) {
		super(name);
	}

	protected void setUp() throws Exception {
		super.setUp();
	}

	protected void tearDown() throws Exception {
		super.tearDown();
	}

}
package request_code;

public class Req {


	public Req() {
		super();
		// TODO Auto-generated constructor stub
```

```
	}

	/*param args*/

	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}
package fipa.agentuml;

package fipa.agentuml.messagetypes;

import fipa.agentuml.Message;

import org.eclipse.emf.common.util.EList;

import org.eclipse.emf.ecore.EObject;


public interface Interaction extends InteractionFragment {

	EList getLifeline(Initiator);

	Elist getMessage(Request);

	Elist getEvent(SendEventOccurrence);

	interface Initiator extends EObject {

		String getRoleName(Initiator);

		getCoveredBy(SendEventOccurrence,ReceiveEventOccurrence,finalEvent

		Occurrence)

		}

	interface Request extends message{

	string getAction();

	void string setAction();
```

```
}
public interface Interaction extends InteractionFragment {
        SendEventOccurrence extends InteractionFragment,MessageEnd{
        Lifeline getLifeline(Participant);
        Set StartExecutionOccurrence(name.EventOccurrence);
        switch (eDerivedStructuralFeatureID(Value)) {
                case AgentumlPackage.EXECUTION_OCCURENCE__NAME:
                return NAME_EDEFAULT == null ? name != null :
                !NAME_EDEFAULT.equals(name);
                case AgentumlPackage.EXECUTION_OCCURENCE__START:
                return start != null;
                case AgentumlPackage.EXECUTION_OCCURENCE__FINISH:
                return finish != null;
                }
        Message extends Eobject Message
        getReceiveEvent(participant_agree,participant_refuse,Not_Understood);
        protected Object doSwitch(int classifierID, EObject theEObject) {
                        switch ( ReceiveEvent) {
                                case MessagetypesPackage.agree:
                                case MessagetypesPackage refuse:
                                case MessagetypesPackage Not_Understood:
        }
        SetReceiveEventOccurrence(message_value);
```

```
}
public interface Interaction extends InteractionFragment {
        SendEventOccurrence extends InteractionFragment,MessageEnd{
        Lifeline getLifeline(Participant);
        Set StartExecutionOccurrence(value);
        Message extends Eobject Message getReceiveEvent(failure,inform-ref,inform)
                protected Object doSwitch(int classifierID, EObject theEObject) {
                                switch ( ReceiveEvent) {
                                        case MessagetypesPackage.failure:
                                        case MessagetypesPackage inform-ref:
                                        case MessagetypesPackage inform:
                }
public interface finalEventOccurence extends InteractionFragment, MessageEnd {
        Lifeline setLifeline(Initiator);
        Void StartExecutionOccurrence(ExecutionOccurrence value);
        void setFinishExec(ExecutionOccurence value);
        }
}
```

This is the core skeleton generated by the EMF model as part of the fipa.agentuml.request package. The implementation code is also generated as part of fipa.agentuml.request.impl package. This is the skeletal code generated by the meta-model for the FIPA Request Protocol.