The Hong Kong Polytechnic University
Department of Computing

Improve the Service Quality of Multi-Agent System – Ontology Management

Ng Kwun Tak

A thesis submitted in partial fulfillment of the requirements for the Degree of Master of Philosophy

November 2003

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written nor material which has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____________________________(Signed)

Ng Kwun Tak________________(Name of student)

I. Abstracts

Multi-agent model has become very popular in the distributed environment in recent years. In order to fully realize its potentials, many studies have focused on improving the service quality related issues such as security, standardization, open environment architecture, integration with existing system, and discovery of services.

In this thesis, we mainly investigate issues related improving quality in service discovery. Service discovery is the use of designated agent directory server for agents to locate their required agent services autonomously based on the nature of tasks required by agents. Currently, directory services only provide run-time binding for an agent to a predefined agent service of which the agent has complete knowledge before hand. However there is a lack of discovery service at development cycle. There is no sufficient mechanism for an agent service to declare or export its functionalities on-line through the Internet so as to facilitate the development of agent applications. On the other hand, the client agent application development may be limited to a closed proprietary environment where the developer follows the service-provider-defined specification to implement the application. In this work, we extend service discovery to the development cycle. By using the provision of ontology and ontology management, the specification of agent services are linked to ontology, enhancement to services are also kept track of through version controls in ontology management. In addition, agents can declare its required services through a language using a defined version the ontology. Thus making it possible for new service providers to serve exists clients as needed services are declared explicitly. The technology developed involves locating relevant ontology, accessing and storing the ontology, keeping track of ontology changes, declaration of service through ontology, mapping services to implementation, automatic and semi-automatic wrapping for different versions of services and finally, at run-time, the binding of agents to services and online directory service for agent mapping with respect to a service type.

II. Publications arising from the thesis

Ng, K.T., Lu, Qin and Le, Yu. *Ontology Management for Agent Development.* In *the Seventh International Conference on Knowledge-Based Intelligent Information & Engineering Systems (KES 2003).* 3-5, September, 2003. Oxford, United Kingdom.

Ng, K.T. and Lu, Qin. *Improve the Service Quality of Multi-agent System: Ontology Management.* In *the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003).* 14-18, July, 2003. Melbourne, Australia.

Lo, C.W., Ng, K.T. and Lu, Qin. *CJK Knowledge Management in Multi-agent m-Learning System.* In *the First International Conference on Machine Learning and Cybernetics 2002 (ICMLC 2002).* 4-5, November 2002. Beijing, China.

III. Acknowledegements

I would like to express sincere thanks to my supervisor, Dr. Lu Qin, who helped me a lot throughout this thesis. She gave me valuable advices on the thesis and provided a motivating atmosphere. Through her support and perseverance, I worked diligently and seriously on this thesis.

I would like to thank my classmates for their friendship, encouragement and the stimulating research discussions. Lastly, I would like to thank my parents Ng Chok Yan and Wong Mun Mui. Without their sacrifice and spiritual support this thesis is not possible.

IV. Table Of Content

I. ABSTRACTS I

II. PUBLICATIONS ARISING FROM THE THESIS III

III. ACKNOWLEDEGEMENTS IV

IV. TABLE OF CONTENT V

V. LIST OF FIGURES AND TABLES VII

LIST OF FIGURES VII

LIST OF TABLES VIII

1 INTRODUCTION 1

1.1 PROBLEM STATEMENT 2

1.2 OBJECTIVES 4

1.3 METHODOLOGY 5

2 BASIC CONCEPTS AND RELATED WORK 7

2.1 BASIC CONCEPTS 7

2.2 OVERVIEW OF QUALITY OF SERVICE IN MULTI-AGENT SYSTEM 9

2.3 STANDARD PLATFORMS OF MULTI-AGENT SYSTEM 11

2.3.1 Agent Management Reference Model *14*

2.3.2 Agent Communication Language *15*

2.4 EXISTING DEVELOPMENT PLATFORMS 17

2.5 ONTOLOGY 21

2.5.1 Ontology Construction *22*

2.5.2 Ontology Management System and Tools *24*

2.5.3 Ontology Versioning *26*

3 SYSTEM ARCHITECTURE AND DESIGN PRINCIPLES 28

4 ONTOLOGY VERSIONING 33

4.1 ONTOLOGY EDITOR IMPLEMENTATION 40

4.1.1 User Interface *40*

4.1.2 Project Manager *42*

4.1.3 Ontolog Editor Core ... *43*

4.1.4 History Manager ... *48*

4.1.5 Undo Manager ... *53*

4.2 COMPREHENSIVENESS ... 56

5 SERVICE TEMPLATE ... 57

5.1 SERVICE TEMPLATE ONTOLOGY ... 58

5.2 SERVICE PROVIDER TEMPLATE AND SERVICE REQUEST TEMPLATE ... 63

5.3 ONLINE SERVICE TEMPLATE REPOSITORY ... 65

5.4 AGENT CODE GENERATION ... 67

5.5 VERSIONING ISSUE ... 70

6 AGENT WRAPPER FRAMEWORK ... 72

6.1 ADAPTER GENERATOR DESIGN ... 72

6.2 IMPLEMENTATION OF ADAPTER GENERATOR ... 75

6.2.1 DOM History File Parser ... *76*

6.2.2 Message Filter ... *77*

6.2.3 Adapter Code Generator ... *80*

6.3 DESIGN OF ADAPTER AGENT SEEKER ... 82

6.4 IMPLEMENTATION OF THE ADAPTER AGENT SEEKER ... 85

6.5 THE AGENT WRAPPER FRAMEWORK SCENARIO ... 88

7 CONCLUSION AND FUTURE WORK ... 95

REFERENCE ... 97

APPENDIX ... 101

A1 ONTOLOGY EDITOR ... 101

A2 ADAPTER GENERATOR ... 103

A3 SERVICE TEMPLATE ONTOLOGY SPECIFICATION ... 105

V. List Of Figures And Tables

List of Figures

FIGURE 2-1 AGENT COLLABORATION DIAGRAM 8

FIGURE 2-2 FIPA ARCHITECTURE-ORIENTED SPECIFICATION STRUCTURE......................12

FIGURE 2-3 AGENT MANAGEMENT REFERENCE MODEL [9]...14

FIGURE 2-4 STRUCTURE OF A MESSAGE [3]...17

FIGURE 2-5 JADE AGENT PLATFORM DISTRIBUTED OVER SEVERAL CONTAINERS [12] 20

FIGURE 3-1 COLLABORATION DIAGRAM WITH ADAPTER AGENT......................................30

FIGURE 3-2 THE REFERENCE MODEL AND THE SYSTEM COMPONENTS............................31

FIGURE 4-1 SYSTEM FRAMEWORK OF ONTOLOGY EDITOR..36

FIGURE 4-2 THE CHANGE OF CONCEPT RELATIONSHIP AFTER DELETION........................45

FIGURE 4-3 TWO JAVA FUNCTIONS GENERATED BY THE ONTOLOGY EDITOR................48

FIGURE 4-4 HISTORY FILE CONTENT..51

FIGURE 4-5 EXAMPLE OF <ADD> TAG IN HISTORY FILE..53

FIGURE 4-6 CLASS DIAGRAM - UNDO SUPPORT CLASSES..54

FIGURE 5-1SERVICE TEMPLATE ONTOLOGY REPRESENTED IN UML CLASS DIAGRAM.59

FIGURE 5-2 COMPONENTS OF THE ONLINE SERVICE TEMPLATE REPOSITORY................65

FIGURE 5-3 MAJOR COMPONENTS OF THE AGENT CODE GENERATION.............................68

FIGURE 5-4 ONTOLOGY VERSIONING AND IMPLEMENTATION RELATIONSHIP................70

FIGURE 6-1 SYSTEM ARCHITECTURE OF ADAPTER GENERATOR 73

FIGURE 6-2 STRUCTURE OF HISTORY FILE 77

FIGURE 6-3 INTERACTION WITH THE ADAPTER AGENT...83

FIGURE 6-4 COMPONENTS OF THE ADAPTER AGENT SEEKER..85

FIGURE 6-5 MUSIC SHOP ONTOLOGY CONSTRUCTION AT THE ONTOLOGY EDITOR 89

FIGURE 6-6 ADD A NEW CONCEPT TO THE ONTOLOGY 90

FIGURE 6-7 SKELETON CODE OF THE SELLER AGENT...91

FIGURE 6-8 SELLER AGENT REGISTRATION CODE TO THE ADAPTER AGENT SEEKER...92

FIGURE 6-9 ADAPTER AGENT SKELETON CODE 93

FIGURE 6-10 MESSAGE FILTERS 94

FIGURE A-1 ONTOLOGY EDITOR USER INTERFACE 102

FIGURE A-2 BEAN GENERATOR FOR PROTÉGÉ 2000 103

List of Tables

TABLE 2-1 LEVEL OF MESSAGING ACTIVITIES 16

TABLE 4-1 ONTOLOGY CHANGE OPERATIONS 35

TABLE 4-2 ONTOLOGY CHANGE OPERATION SUMMARY 43

TABLE 4-3 CONVERSION RULES AND FUNCTIONS 47

TABLE 5-1 COMMON METHOD OF THE JAVA HELPER CLASS 69

TABLE 6-1 METHODS OVERLOADED FROM JADE.DOMAIN.DFSERVICE CLASS 86

1 Introduction

Due to the popularity of Internet, computers are no longer a stand-alone system. Computers are well connected in the network. The information system becomes more complex. The distributed software components are useful for solving complicated problems, by sharing computer resources on different host computers in a distributed environment.

An agent is a software program which can act autonomously depending on the environment changes and its own knowledge. It is "a program that assists people and acts on their behalf. Agents function by allowing people to delegate work to them" [26]. A Multi-agent system contains a group of agents which interact with each other to solve a problem or accomplish a task at run time environment on a the agent platform. Agent interactions are goal-oriented and task-oriented [42]. Agents can be stationary or mobile. A *stationary agent* always stays at its creation place, while a *mobile agent* can be migrated to other hosts. A mobile agent can navigate on an agent platform from one host to another with their code, state and data in order to accomplish a task. Multi-agent systems become important in distributed environments.

Multi-agent systems can be used in various applications such as distributed information retrieval, electronic commerce, network management and work flow scheduling [10]. Various multi-agent system platforms and standards are proposed by different companies. As the popularity of the multi-agent model increases, some multi-agent systems have already been launched in the market [11]. However, the multi-agent model is still in the early stage of its development. There is a limited number of commercial multi-agent systems available. In order to fully realize its potentials, many studies have focused on improving the service quality related issues such as security, standardization, open architecture, integration with existing system, and discovery of services.

1.1 Problem Statement

Researchers have raised the question on the homogeneity of agent systems [41]. In some high-level applications example, such as information retrieval, it seems that an agent has the intelligence to interact with the execution environment, to find the appropriate data source, and to use the agent service interface. In fact, these kinds of know-how knowledge were coded in the agent design phase. For the open environment with different access control mechanisms and functionalities provided by different agent services becoming available at different times, it is not reasonable to assume that agents have the prior knowledge to deal with all of them. There should be ways to expose the specific access control mechanism and functionality of a specific agent service which can then be used both at the agent's development cycle and also at run-time. This falls into the category of service discovery.

In this study, we mainly study issues related to service discovery. Due to an agent's autonomous characteristics, it needs to find available agent services without an user intervention when it navigates through various hosts, which we call *service discovery*. We use the term *agent service* to refer to a service provided by an agent server to be used by a client agent. While an *agent service developer* is responsible for developing various agent services, a *client agent developer* is responsible for developing agent application to use some agent services. One of the most important services on an agent platform is called *service discovery* which has some designated agent directory servers for agents to locate their required agent services autonomously based on the nature of tasks required by agents [11]. Currently, directory service only provides run-time binding for an agent to a predefined agent service of which the agent has a complete prior knowledge. However, there is a lack of discovery of services at the development cycle. There is no mechanism for an agent service to declare or export its functionalities on-line through the Internet so as to facilitate the development of agent applications. On the other hand, the client agent application development may be limited to a closed proprietary environment where the developer follows the service-provider-defined specification to implement the application.

1.2 Objectives

It is more desirable that client agent developers can access the features and functionalities of agent services when they are designing client agents. As a result, there should be ways to describe the functionality and properties of agent services. An agent management component is also needed to keep the stock of various agent service classes and provides an interface for the agents to look for the required classes [10].

It is also desirable that an agent can declare the kind of services it needs once the development is over. The declared service type, which we call *generic requested service,* provides a way for agent service providers to find out what are needed by client agents. Thus, a new agent service can be adapted to the existing kind of generic requested service, so that a client agent coded earlier can use agent services which are developed at a later time.

The objectives of this study is (1) to find a way for agent service providers to declare and export the types of services it can provide with interfaces and methods so as to facilitate the development of agent applications, (2) to find a way for client agent developer to declare and export a generic requested service its needs at the development stage. This way, a client agent can be developed independent of the service agent which may be developed at a later stage, (3) methods for new agent service providers to provide service to existing client agents which have declared generic requested service, and (4) find ways to bind client agent with appropriate service agents at run-time.

1.3 Methodology

Our approach is to investigate how to make use of ontology to facilitate the development of agent based systems. Ontology is used to model the agent world. It generally specifies the concepts, objects and relationship between concepts of a domain of interest. Methods of using ontology and its association with agent services have been explored recently since ontology reflects the agent capability. The idea is that if we can make the association of ontology with services, service providers can declare their services through a well defined language with semantic definitions. Change and enhancement of agent services can be kept track of through the management of ontology repository.

In this work, we will first study the architecture of multi-agent systems and the implementation of multi-agent system platforms. Then, we try to investigate agent service related problems and review current solutions. We will then explore how to make use of an ontology repository for agent development. This involves locating relevant ontology, accessing and storing ontology, keeping track of ontology changes, mapping ontology to implementation, automatic and sem-automatic generation of wrapping service so that new service providers can serve older client agents, and online directory service from agent binding with respect to required service types.

The rest of the thesis is organized as follows. Chapter 2 gives basic concepts and related work. It introduces two major agent system standards and agent platforms. The detail of ontology application on multi-agent systems is studied. Chapter 3 presents our methodology and introduces our proposed system architecture. Chapter 4 describes the implementation of the system component related to ontology versioning. Chapter 5 discusses the details of linking ontology with agent implementation. Chapter 6 presents the Agent Wrapper Framework. Lastly, Chapter 7 concludes this thesis and discusses future work.

2 Basic concepts and Related Work

2.1 Basic concepts

An agent is a software component which acts autonomously when there are environmental changes and coordinate with other agents [42]. A multi-agent system often runs at an agent platform in which provides necessary development and run-time environment. An agent platform provides online functions to facilitate agent execution and agent management with the support of various underlying technology. In order to facilitate agent communications, an agent platform provide higher level of agent communication functions on top of some basic communication facilities provided by an operating system such as Remote Procedure Call (RPC), Remote Method Invocation (RMI) or TCP/IP protocol.

Most of the agent systems are written in the Java programming language. Java's promise of "write once, run anywhere" makes its code portable on heterogeneous hardware and operating system platforms over a network. Java's Naming and Directory Interface (JNDI) is a standard extension to the Java platform, providing Java technology enabled applications with a unified interface to multiple naming and directory services. Agents can make use of JNDI service providers to find out the required services.

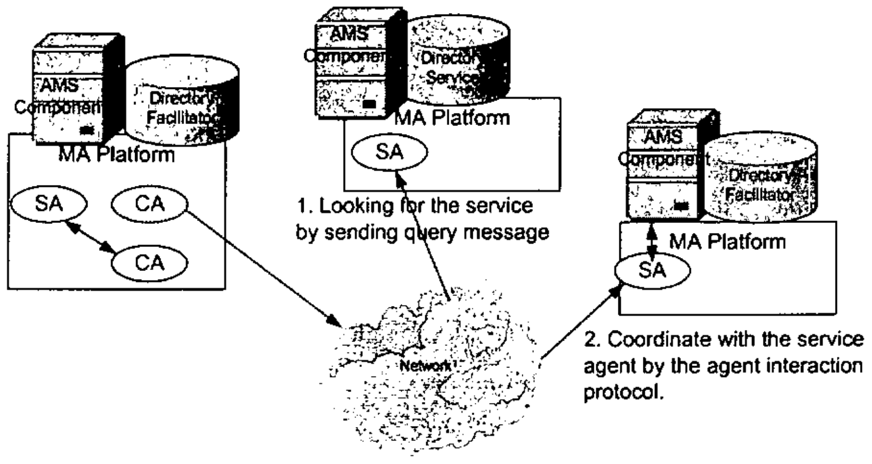


Figure 2-1 Agent Collaboration Diagram

Figure 2-1 shows a simplified multi-agent (MA) system model where agents are categorized into service agents and user agents. A *service agent (SA)* provides services for client agents. A *client agent (CA)* seeks services from server agents. Each host in a MA platform has a directory facilitator which maintains a list of agent in the current MA platform and an agent management system (AMS) component. There is also a directory service which is used to publish agent services providing at other MA platforms. Since an agent platform is not limited to a single host, intra-platform agent migration is possible. In a multi-agent system, a service agent can serve as a *wrapper* or sometimes called proxy through which a legacy application or an older version of a client agent can be serviced by new service agents.

What makes MA system different from other paradigms is the autonomous actions and interactions of agent to achieve a goal. While the constructed software components are assumed to have the same goal in most software engineering practices, an agent acts autonomously according to its self-interest [43]. Agents interact with each other by sending messages which are often written in the Agent Communication Language (ACL). The content of the message is written based on some agreed concepts defined in an ontology so that agents can do reasoning and inference processes on a common ground.

Generally speaking, an ontology is a semantic framework which can be used to model the agent world. An ontology specifies the concepts, objects, and the relationships between concepts in an area of interest [42]. It is similar to a database schema which describes concepts and relationship structures rather than storing the actual instance of data. In this thesis, *ontology* is defined based on the FIPA specification [2] that specifies concepts, proposition, rules, and actions.

2.2 Overview of Quality of Service in Multi-agent System

In this thesis, *quality of service* is often measured in terms of functional comprehensiveness and in this thesis it is viewed from the perspective of agent applications. Recent study on service quality of multi-agent system often includes security, agent platform standardization, integration with existing system and discovery of services. Although the development of multi-agent systems is still in its infant stage, these studies help to make it more comprehensive [11, 15, 30].

Security is one of the high profile issues currently because MA systems are designed for real applications over the internet or network, especially when agents are mobile and can navigate from one host to another. Security issues should consider access control, integrity, privacy and resource consumption [9]. For example, SOMA is one of the system framework studies security and interoperability [10].

As the popularity of the multi-agent systems grows, various platforms are implemented. However, inter-operation between heterogeneous agent platforms is often not possible. Standardization of agent platforms and the design of common architecture can provide a common ground for various applications. Currently, FIPA and MASIF are the two main standards developed for MA systems. Details of standards will be discussed in Section 2.3.

For existing legacy systems and applications, it is often too costly to completely re-develop them. Therefore, integration with the existing systems is one important topic in improving quality of service in a multi-agent system. Integration ensures that existing applications can get the benefits from newly developed multi-agent systems but will not involve a high re-development cost. Component based architecture [30] and the open agent architecture system [15] addresses the integration issue.

There is a number of works on run-time service discovery. Their main contributions are on discovering services and resources in an agent platform especially to address the problem of ad-hoc wireless network condition [27]. However, there is lack of work on service discovery in agent applications' development cycle, nor any study on the linkage between service development and the runtime. This binding by service types motivates our research to find solutions for providing a better development environment as well as smooth and autonomous binding at runtime.

2.3 Standard Platforms of Multi-agent system

There are many open agent system standards and agent-based software engineering methodologies studied before. There were already over 60 agent platform systems before the introduction of the first agent standardization. Standardization of agent systems is introduced to address the interoperability issue. There are two main agent standardization organizations working on these matters. The Object Management Group (OMG) developed the Mobile Agent System Interoperability Facility specification (MASIF) [33] and the Foundation for Intelligent Physical Agents (FIPA) developed the FIPA specification [3].

MASIF specification aims at providing a unified middleware for heterogeneous mobile agent system so that certain degree of interoperability can be achieved without large modification of existing systems. MASIF only addresses interoperability between platforms written in the same programming language. Interoperability in MASIF is standardized in terms of agent management, agent migration and navigation, agent naming, and agent types.

The FIPA specification was originally concentrated on intelligent co-operation in the context of multi-agent systems. Yet, the latest FIPA specification included agent mobility support. As FIPA is more comprehensive, it will be used to develop our system in this thesis. Thus, we will give a more thorough discussion FIPA.

The FIPA specification can be divided into two main areas, architecture-oriented and application-oriented. The FIPA Application specification defines the way to use agents on different application areas. As FIPA Application specification is informative and not considered as standard, we will not discuss it in details.

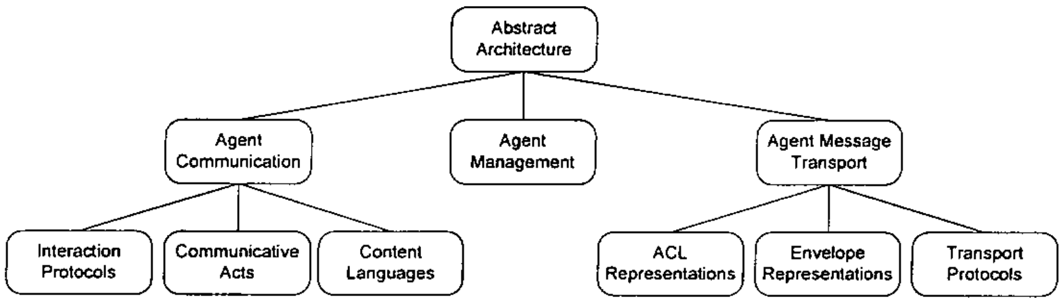


Figure 2-2 FIPA architecture-oriented specification structure

The FIPA architecture-oriented specification is concerned with building agent services and agent environment. It is normative and technology-oriented. Figure 2-2 shows the FIPA architecture-oriented specification structure. The architecture includes three parts: agent communication, agent management, and agent message transport.

Agent communication deals with interaction protocol, communicative acts and content languages [5]. In a FIPA compliant multi-agent system, agents co-ordinate and communicate with each other by exchanging messages written in the Agent Communication Language (ACL), such as FIPA-ACL and Semantic Language 0 (SL0). Agent Communication Language is based on speech act theory: messages are actions, or communicative acts, as they are intended to perform some action by virtue of being sent [25].

Agent Management deals with the control and management of agents within and across agent platforms [4]. It specifies the agent platform reference model for the creation, registration to the directory facilitator, location, communication, migration and retirement of agents. The details are further discussed in Section 2.3.1.1.

FIPA Agent Message Transport deals with the transport and representation of messages across different network transport protocols, including wireline and wireless environments [5]. These specifications include ACL Representation, Envelop Representation and Transport Protocols specification. Agent Communication Language specification defines different representational format to encode the ACL messages. The ACL messages can be encoded in XML, String and bit-efficient format. Envelop Representation specification defines how to use XML or bit-efficient to encode the envelope. Transport Protocol Specification concerns about the ways of transmitting ACL message using different network protocols, such as WAP, IIOP and HTTP.

In this study, we will concentrate on the directory facilitator since it is related to the online service registration and the agent service lookup in agent management.

2.3.1 *Agent Management Reference Model*

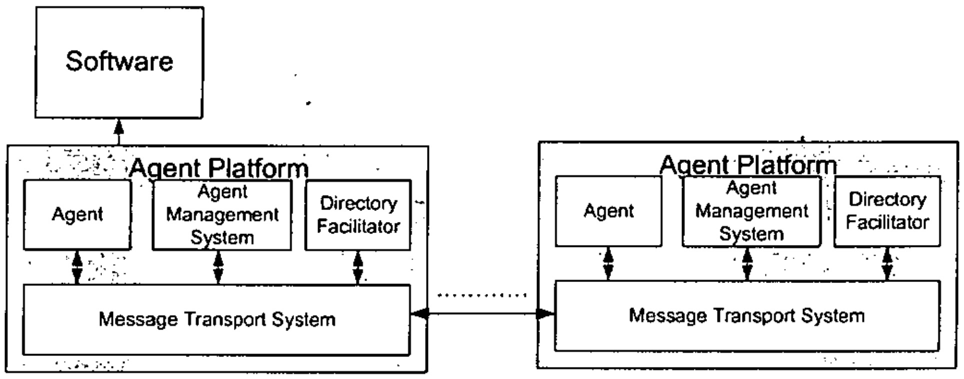


Figure 2-3 Agent Management Reference Model [9]

The reference model in FIPA is shown in Figure 2-3. An agent platform should have three service components: an Agent Management System, a Directory Facilitator and a Message Transport System.

Besides to agents, the agent management system maintains the life cycle of an agent and keeps track on it. It stores the transport addresses of the agent registered with the Agent Platform. The directory facilitator provides yellow pages service to other agents. It provides agent service registration and deregistration, service update, and answering inquiries. The message transport service provides the communication method between agents on different FIPA-compliant Agent Platforms.

2.3.2 Agent Communication Language

In FIPA compliant platforms, agents communicate with each other by message passing. Message passing is not new in distributed computing. However, the difference between ACL and the previous models such as RPC system is the semantic complexity and the object of discourse [25]. Since multi-agent system was originated from Artificial Intelligence subject, ACL expresses propositions, rules and actions to describe semantics of message content.

Every communicative act in ACL is described with both a narrative form and a formal semantics based on modal logic according to human communication theory. The messaging activities between agents can be decomposed into the four levels as summarized in Table 2-1.

Table 2-1 Level of messaging activities

Level	**Description**	**Real World Example**
Interaction Protocols	Social rules to conduct the conversation. Conversation means exchanging a set of agent messages.	Someone looks for an apple and he requests the fruit seller to sell. Fruit seller replies with the price and then sell it.
Agent Communication Language (ACL)	Specify the communicative act. FIFA aims at allowing heterogeneous agent communicate through FIPA-ACL.	"I want ...." It is a request.
Content Language	Specify the content of the communicative act	"I want you to sell an apple" apple is the content.
Ontology	Specify semantic meaning of the content. It is the primitive term using in the content language.	The meaning of apple and sell.

The highest level of agent communication is interaction protocol. Agents talk to each other follows a pattern defined in the interaction protocols. The communication language defines the format of an agent message and the communicative act. While the content language specifies the content of the communicative act, the content semantics is defined by ontology.

Figure 2-4 shows the structure of an agent message. The basic messaging information such as sender, receiver, content language, protocol and communicative act are expressed in Agent Communication Language. The content of a message refers to the domain specific component of the communicative act and the content expression may include propositions, actions or terms. The content is expressed in a content-language, such as SL (Semantic Language) a formal language used to define semantics. A content-language may reference an ontology defines the semantics of the content. Ontology will be further discussed in the section 2.5.

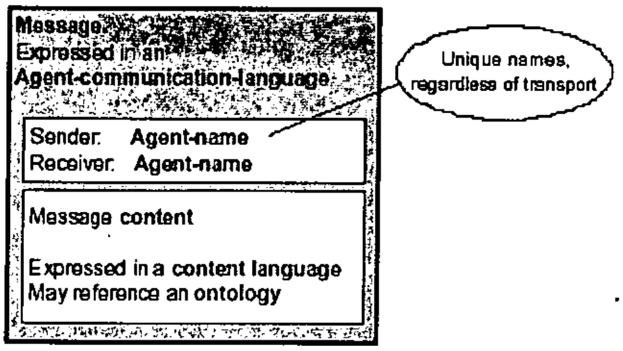

Message
Expressed in an
Agent-communication-language
Sender: Agent-name
Receiver: Agent-name
Unique names, regardless of transport
Message content
Expressed in a content language
May reference an ontology

Figure 2-4 Structure of a message [3]

2.4 Existing Development Platforms

FIPA is more comprehensive than MASIF. Not only does FIPA provide support on mobility, but also on the intelligent agent communication and co-ordination using agent messages. As we follow the FIPA Specification in this thesis, we will give an overview of current major FIPA-compliant agent development environments.

Agent Development Kit [8] is a mobile component-based and modular architecture development platform. It provides a commercial development kit for Java-based mobile agent applications and components. Compared with other open platforms, it has fewer third party or research institutes research involvement since it is not an open source platform.

Like Agent Development Kit, Grasshopper [14] is a commercial product. It has a mobile agent environment and provides plug-in add-ons for its core platform in order to support both the MASIF standard and FIPA standard. However, it is only compliant to FIPA-97 specification and provides limited functions to compose an ACL message.

Zeus [35] toolkit provides a library of software components and tools for the design, development and deployment of agent systems. It consists of three functional components: an agent component library, an agent building tools and a suite utility agent comprising name server, facilitator and visualiser agents. These components provide visual environment for the generic agent functionality development and support planning and scheduling of an agent actions. It is one of the early day agent toolkits. But, its recent development or activity seems sluggish.

Java Agent Services [1] is designed by major computer vendors such as Fujitsu, Sun, IBM, HP, Comtec, etc. Java Agent Services defines an industry standard specification and Java API for the implementation of the architectural features of the FIPA Abstract Architecture. It aims at creating commercial applications using FIPA specifications. However, it is still in the infant stage of development and no platform implementation is available yet.

FIPA-OS [38] is a component-oriented toolkit for the development of FIPA compliant agents. Agent is built from three types of components: mandatory components, components with switchable implementation and optional components. Major FIPA Experimental specifications are supported. It is an open source agent toolkit and is going to be improved by various developers from the agent development community.

JADE [12] is a software framework fully implemented in the Java language. It is compliant to the latest FIPA-2000 specification. It provides debugging tools and deployment tools for the agent development. Besides, there are third-party add-ons to enrich the functionality and features of the development kit. The agent platform can be distributed on different hosts as shown in Figure 2-5. Each host has a Java Virtual Machine, which serves as a container of agents to provide a runtime environment for agent execution. As shown in Figure 2-5, the JADE platform contains only one main container which contains Agent Management Service (AMS), Directory Facilitator (DF) and RMI registry. In addition, the main container provides a front-end GUI tools for agent debug and deployment. Containers on other hosts are also part of the platform. They connect to the main container and form a complete run-time environment for the execution of agent. Intra-platform agent migration is allowable. It is an open source agent platform with lots of development features. Due to the wide range of functionalities provided for JADE and the fact that it is provided free of charge and it is fully compliant to FIPA-2000, we choose JADE as our agent platform.

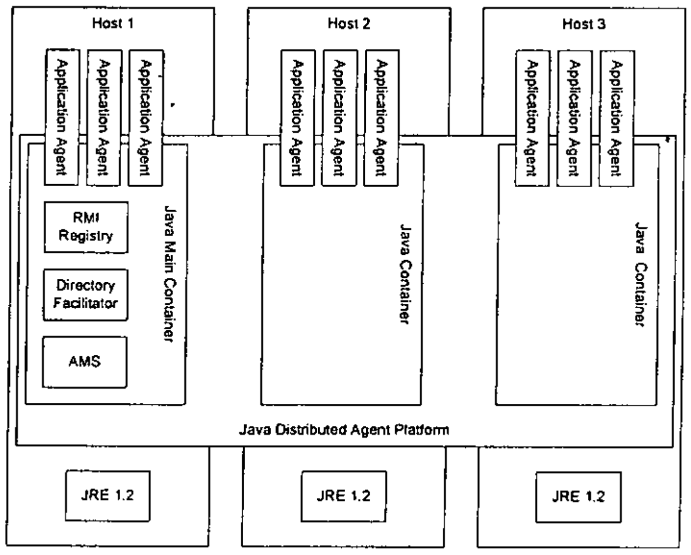


Figure 2-5 JADE agent Platform distributed over several containers [12]

2.5 Ontology

Before we going into details of ontology, we consider the following case. Suppose there is a CD Buyer agent and a CD Seller agent in a virtual market. A CD Buyer wanting to buy a CD must first specify the album name and the preferred price. In other words, both the CD Buyer agent and the CD Seller must already have reached the consensus on the semantic meaning of what "Album Name" and "Price". Ontology specifies the concepts and taxonomy used in agent communication.

People are often like to model the real world in order to build an application capable of handling real life applications. Since the real world is so complex that it is both hard and costly to represent all the details inside it, only a part of the real world or a specific focus are called "domain" is considered in developing an application. A *domain* is confined to a subject or an area of real world knowledge, such as transportation, mathematics, physics, medicine, etc. Different methodologies were developed to capture the domain knowledge. Ontology was one of the ways to represent domain concepts and their relationships. Ontology has become a hot topic in agent development society because of its suitability in agent communication.

"An ontology is an explicit specification of a conceptualization" according to Thomas R. Gruber [18]. Ontology is mainly used for explicit information representation. For human being, we use natural language to communicate and get the work done. Yet, it is not an easy task for computers to understand a natural language due to intrinsic ambiguity nature of natural languages. In order to let machines or software agents to understand the semantic meaning of a language, ontology specifies the concepts and their relationship in a formal ways [42]. This greatly simplifies the representation of semantics in the natural language. It bridges the semantic gap between syntactic representation and its concepts [28]. Other miscellaneous usages are global query model and verification of the integration description [40].

Agents can communicate and interact with each other according to their functionality because they use the same communication language with a common vocabulary, which contains keywords in a common application domain. The means of the keywords are defined in a shared ontology [13].

2.5.1 Ontology Construction

A new ontology can be constructed from the ground or reuse the existing ontology. Ontology reuse can be divided into two categories [36]: integration and merging. *Integration* means assembling, extending, specializing and adapting ontologies from different subjects or domains into a new ontology. The resulting ontology is often customized for a specific application. *Merging* combines different ontologies from the same domain or related domains into a common ontology. Since organizations and industry in the same domain may construct their ontology similarly, merging can update the definition of concepts to the industry standard and attain consensus ontology.

Recent research studies have shown attempts to automate the process of integration and merge [37]. However, a complete automatic solution is still not yet possible. It needs human to resolve the ambiguity of domain knowledge, so it may hard to attain this goal in the near future.

Both integration and merging finally result in some designated ontologies for applications in certain domain. Variants of these ontologies are developed as concepts are changed and expanded as time goes by. Consequently, there are two kinds of relationships when ontology changes: the relationship between different or similar domain ontologies and the relationships between ontology of different versions in the same domain, we refer to the second relationship as *ontology-variant relationships*. In this thesis, we only focus on the study of ontology-variant relationships and how to transform one version of an ontology to the other.

2.5.2 Ontology Management System and Tools

At writing of this thesis, the OntoWeb [17] is still in its early stage. OntoWeb provides a Service Description Language DAML-S and a DASD (DAML agents for Service Discovery) to binding web service with client agents. However, OntoWeb does not handle ontology change.

Semantic Network Ontology Base (SNOBASE) is designed by IBM [21]. It is a framework that allows the access of ontologies from a file system and the Internet and is used for locally creating, modifying, querying, and storing ontologies. Its major feature is to provide a programming interface to access the ontology. There is no direct linking between ontology and an application.

Chimaera is a software system that supports users in creating and maintaining distributed ontologies on the web [32]. Two major functions it supports are merging multiple ontologies together and diagnosing individual or multiple ontologies. It is a knowledge management tool to build ontology for experts in certain domains. Again, it is not directly linked to any implementation of services using the ontology.

Exteca is a collection of technologies which can form the foundation of knowledge models to support knowledge management system [16]. When an ontology needs to be transformed into another ontology, a set of filter rules can be defined to describe how the transformation should take place.

OntoManager [19] tries to formalize the ontologies and transform them into a more expressive representation, using RDF, DAML+OIL, OWL. It provides an interactive tool to semi-automate the detection and the resolution of various ontological mismatches in a workbench environment.

Protégé 2000 [34] is an ontology tool to help developers to construct domain ontology, such as defining concepts, predicates, slots and agent actions. Due to its well defined API structure, providing library access, supporting third party plug-in and large user communities, Protégé 2000 is chosen as our ontology editor environment. With appropriate plug-in [7], the tool can generate Java class to support the definition of ontology in JADE. However, Protégé 2000 does not keep track of ontology change.

In short, most of the current ontology management systems are mainly for ontology storage, knowledge management and semantic web. The links of ontology with agent development is not addressed. That is what motivates us to investigate methods to make use of ontology in agent development cycle so that service agents can have declared types of service which in turn can help client agents to be bound to the right type of service at run-time.

2.5.3 Ontology Versioning

When an ontology is used by a multi-agent system, the ontology engineering process does not end at there. In real life, same concepts become obsolete and new concepts are being created. This means that ontology is not static and it needs to be updated as times goes by generating newer version of ontology. Basically, if a service agent and the corresponding client agents are build based on certain version of ontology, the service agent would not be able to serve some client agents which "speak" a newer language involving new concepts. On the other hand, an older client agent would not enjoy a newer service agent which provides some service that the older client is not aware of nor understand.

Currently, there are a number of studies [20, 22, 29] on ontology versioning. *Versioning* is a method to keep track of the ontology changes as it evolves. It makes the relation of one version of a concept or relation to other version of that construct explicit [24]. However, they are mainly concerned with Semantic Web to annotate the web pages information. Ontology is linked to information in web pages so as to allow software agent or machine understanding the web page. However, the association of ontology with agent capabilities in the multi-agent system is not entertained. In addition to representing instance of concepts, predicates, actions and rules are required to be specified by ontology in the multi-agent system. We are going to work on this issue.

3 System Architecture and Design Principles

As mentioned in Chapter 2, if ontology evolves and the change is not properly managed, agents using different versions of ontology cannot communicate with each other. Existing agent implementation may need to be re-constructed which can be very costly for agent application users.

In this work, we aim to use ontology management to link ontologies with agent development. The linking includes three aspects. Firstly, each implementation, which we call a *declared service implementation*, is linked with a particular version of an ontology, thus making the "language" it speaks explicit. The service provided by this implementation is also made explicit with respect to the part of the ontology used in this implementation as an implementation does not need to implement every construct in a particular version of ontology. Secondly, an client agents can make a so called *service request declaration* where a client makes it explicit the concepts it understands, the "language" it speaks, the service it requires and the interface it uses for such a service. Even though a service request declaration can be identical to a declared service implementation, which should be the case in more instances, such declaration may be a declaration for a preferred service which can be implemented later by someone who finds such a service is worthwhile to provide. Thirdly, we use the mechanism of version control to keep track of ontology changes in a systematical way. This makes it possible to know exactly what the difference between two implementations linked to different versions of the ontology thus making wrapping service for different service agents possible.

With the provision of service request declarations and wrapping service, new agent services can be used by existing client agents. Here the term *wrapping* refers to the construction of a proxy agent which understands two versions of an ontology and act as the "translator" between a client agent and a service agent which is built on different versions of the ontology. A wrapping service generates a so called "adapter agent" which is an agent that converts ACL message from one version to the other and it serves as a translator for different versions of the same ontology. Detailed discussion will be given later in Chapter 6.

With the adapter agent as a proxy, a client agent can speak to a newer service implementation (or even a newer version) because the wrapping has taken care of the translation. In other words, the client built earlier can use the agent service developed later. Wrapping service helps to save time and reduce the cost of agent re-implementation. The service declaration allows a service agent to advertise their available service. It also allows a client agent to declare the service it needs.

Figure 3-1 shows the proposed collaboration model with the newly introduced adapter agent. When a client agent looks for a service agent by querying a directory facilitator (shown in Step 1), the directory facilitator (*DF*) tries to locate the best matched service agent (shown in Step 2). If the directory facilitator cannot find the exact match, it will try to look for a service with the appropriate adapter agent (*AA*) (shown in Step 2'). In this case, the adapter agent acts as the interface between the client agent and the service agent (*SA*') so that service of *SA*' can serve a *CA* even if *SA*' and *CA* do not speak the same "language".

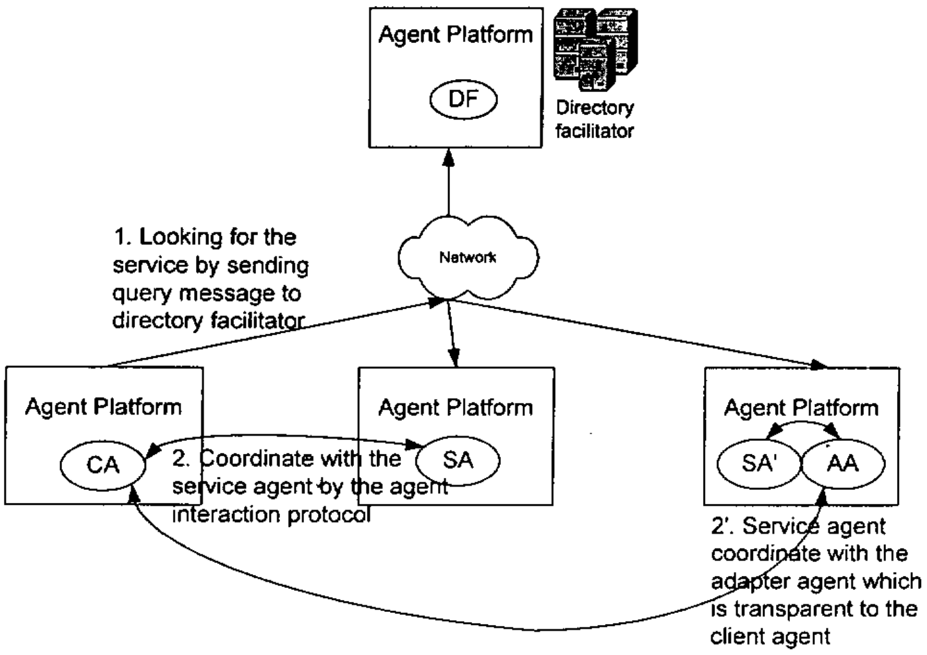

Agent Platform
DF
Directory facilitator
Network
1. Looking for the service by sending query message to directory facilitator
Agent Platform
CA
Agent Platform
SA
Agent Platform
SA'
AA
2. Coordinate with the service agent by the agent interaction protocol
2'. Service agent coordinate with the adapter agent which is transparent to the client agent

Figure 3-1 Collaboration diagram with adapter agent

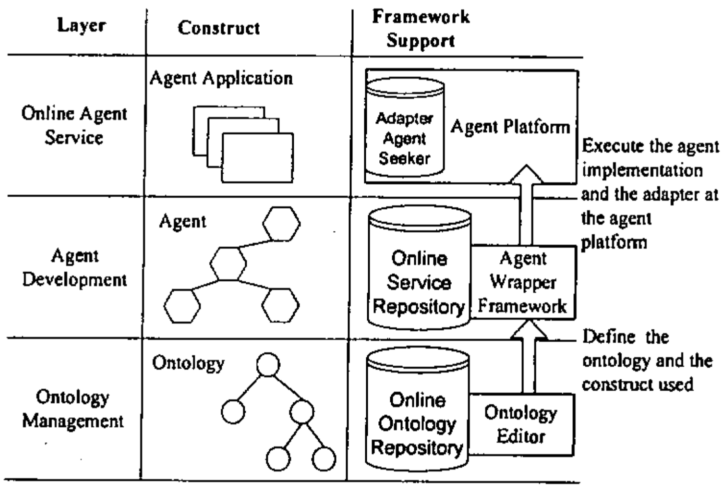


Figure 3-2 The reference model and the system components

The proposed ontology management platform, referred to as the **OntoWrap** in this thesis, provides three layers of services, namely **ontology management**, **agent development**, and **online agent service** as shown in Figure 3-2. At the ontology management layer, ontology is managed by one data component and a functional component, namely, the **Ontology Repository** and the **Ontology Editor**. The Ontology Repository keeps records of ontologies and ontology changes so that the ontology-variant relationships between different versions of ontology and their extensions are maintained. The Ontology Editor provides an interface to modify and create an ontology. At the agent development layer, service related declarations are also carried out by a data component and a functional component, namely, the **Service Repository** and the **Agent Wrapping Framework**. The Service Repository keeps records of service request declarations and declared service implementations as well as relationship data to link ontology with agent service specifications given in the ontology repository. The Agent Wrapper Framework provides the interface between service declaration and service implementation. It is also where the wrapping code is generated for adapter agents. Note that we used the term *Framework* in this component because we cannot guarantee automatic generation adapter agents. If some of the changes in an ontology are semantic in nature, wrapping service may still needs to be programmed by a person. In that case, the skeleton code will be generated for programmer to insert the code. At the online agent service layer, an extended directory service called the Adapter Agent Seeker is provided to help locating the right implementation or the right adapter agent for a requested service at run time as discussed in the collaboration model in Figure 3-1.

Before the development of any multi-agent systems, the problem domain must be analyzed first. Through conceptualization, the problem domain and the planned target objectives are realized in a defined ontology, which enables an agent to understand the message content during agent communication. Consequently, the produced ontology reflects and binds the capability of an agent. Various domains of ontologies are created and stored in the ontology repository. When there are ontology changes, its versioning function keeps track of these changes and distinguishes ontology variants.

4 Ontology Versioning

In this thesis, we use ontology versioning to keep track of ontology changes. It is one of the most important features of OntoWrap. Two types of ontology change operations are identified as the *syntactic change operations* and the *semantic change operations*. A syntactic change operation changes the ontology definition in its syntax representation only. A semantic change operation, however, changes the semantics meaning of the definition. In the following context, the term *ontology construct* refers to a tree node in the ontology structure, such as, a concept, an agent action or a predicate. Not all ontology change operations can be used for the generation of adapter agent. When a totally new concept is introduced, such as a Book concept added into a Music Shop Ontology, the automatic translation by adapter agent may not be possible. At least the adapter agent knows that an older client does not know what a Book is, so it will block any message pieces related to book service to client agent.

The Ontology Editor is designed to facilitate the development and the evolution of domain specific ontology. It keeps track of ontology changes during the editing of the ontology. Currently, most of the ontology editors [39] only provide basic functions to construct ontologies. The agent construction and more comprehensive ontology construction functionalities, such as undo function, are not available in these editors. According to [39], an ontology editor is suggested to have the following features:

- Supporting evolutionary changes
- Enabling users to resolve a change
- Providing control over evolution changes
- Declaring ways to undo previous change effects
- Managing ontology change history
- Providing support for continuous ontology improvement
- Finding inconsistencies and reasons for easy ontology management

The ontology management system, Protégé 2000, allows users to define generic classes and class hierarchy, slots and slot-value restrictions, relationships between classes and properties of these relationships. However, there is no class constraint for agent development. In the FIPA 2000 specification, ontology includes elements, such as concepts, actions, predicates and relationships. Currently, users can only create these basic elements as classes in Protégé 2000 and create subclasses that inherit these basic element classes. For example, the buy action and sell action are the subclasses of agent action class. This approach does not have sufficient control over ontology construction and declaration of ontology elements.

OntoWrap, makes use of the basic functionalities provided by Protégé 2000 so that we do not need to reinvent the wheels. In order to ensure the correctness and consistency of ontology construction, we extended Protégé 2000, by implement our extended service as a Protégé 2000 tab widget plug-in to allow ontology experts to select the ontology elements and define appropriate element properties. For instance, when the developer defines an agent action, an agent action property window dialog is displayed to let a user enter the required action arguments that are constrained to concepts.

Table 4-1 Ontology Change Operations

Change Operation		**Target Construct**
Syntactic Operation	Add	Concept, Agent Action, Predication, Slot, Argument
	Delete	Concept, Agent Action, Predication, Slot, Argument
	Rename	Concept, Agent Action, Predication, Slot, Argument
Semantic Operation	Replace	Slot and Argument

Table 4-1 lists the basic change operations such as *add*, *delete*, *rename* and *replace* and the ontology constructs these operations can be applied to. Moreover, the version information, that is the sequence of change operations performed, the version number and meta-information, can be saved during the ontology update process. Undo function is also implemented to facilitate ontology construction.

With reference to the Model-View-Controller paradigm [31], the Ontology Editor is mainly composed of two parts: the User Interface (UI) and the Ontology Editor Engine. This partition is to keep each component as much independent of the other as possible so that either one can be modified without affecting the other.

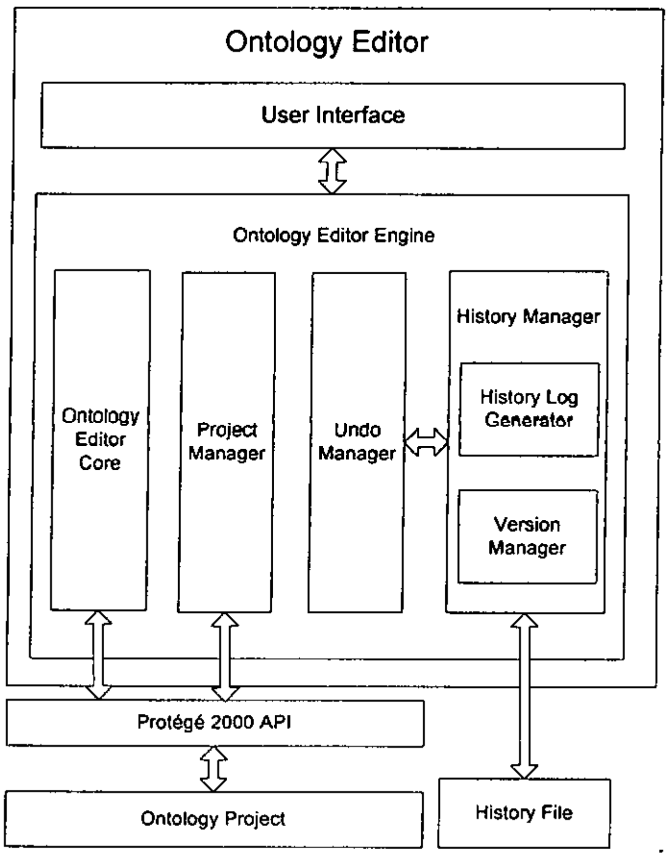


Figure 4-1 System Framework of Ontology Editor

The major components of the Ontology Editor are shown in Figure 4-1. The **User Interface** is a graphical user interface (GUI) which mainly interacts with the ontology designer. Ontology is represented in a tree structure on the graphical user interface. Through the manipulation of the ontology tree structure, an ontology designer can create new constructs and make changes to the ontology. Those changes are passed to the Ontology Editor Engine. After the engine has processed these change operations, change effects are immediately reflects on the GUI. The **Ontology Editor Engine** modifies the ontology according to the input. It also keeps a record of the changes performed as well as the version information. The Ontology Editor Engine is composed of four components: the **Ontology Editor Core**, the **Project Manager**, the **History Manager** and the **Undo Manager**. The Ontology Editor Core modifies the underlying ontology model. The Project Manager is responsible for opening and saving the project. The History Manager saves the sequence of changes and the versioning information to a history file in XML format. Undo and redo functions are provided by the Undo Manager as well. Note that the Ontology Editor Engine is built on top of Protégé 2000. Thus Protégé 2000 is supplied to OntoWrap as a development platform.

The main function of the Ontology Editor Core is to modify the underlying ontology model according to the request passed by the User Interface. It supports those basic change operations such as *add*, *delete*, *rename*, and *replace*. Sometimes an ontology developer may want to make use of a concept in an existing ontology in order to maintain the consistency of all the ontologies developed. Protégé 2000 provides the function to import a whole ontology project. It may not be helpful since not all the elements are needed. In OntoWrap, we have a function to copy and paste a partial ontology construct of an existing ontology into the working ontology. As shown in Table 4-1, the *replace* operation is applied to slot and arguments only. This operation keeps track of changes related to slots and arguments to handle complex changes that cannot be carried out easily by simple operations *add, delete and rename*. Changes can be further categorized as arithmetic change and custom change and they can be declared for each change operation. A *type change* specifies the primitive data type changes made to a concept slot or an action argument. It can be a syntactic change, such as changing from integer type to string type. We use the *arithmetic change* to record a slot or an argument value change in terms of arithmetic expression. For example, while a price slot value of a product changes from HK dollar to US dollar, the arithmetic change records currency exchange's calculations. A type change declaration and an arithmetic change declaration can help to fill the information for automatic generation of adapter agent code. Lastly, a *custom change* records the custom defined programming code to specify the change. Custom change declaration is useful for some complicated cases which cannot be recorded by type change and arithmetic change. For instance, the date of birth slot of a person can be changed to the age slot through manually inserted conversion code. Further details of these conversions will be described in Section 4.1.

The Project Manager is responsible for opening and saving the ontology projects. It makes uses of the existing functions and the same file format of Protégé 2000. An ontology developer can open an existing ontology project using the Project Manager. When the ontology is modified, the Project Manager saves the ontology project in the appropriate location based on the version information given by the History Manager.

The Undo Manager is responsible for the undo and redo functionality. It gets the ontology change operation history from the History Manager to restore a previous version of ontology. The undo function not only allows a user to correct his mistakes but also enables a user to try different ways of constructing an ontology. This satisfies the reversibility requirement which states that "an ontology editor has to allow undo changes at the user's request" [39].

The History Manager is responsible for saving a sequence of change transactions performed to the history file during ontology update. Each *change transaction* includes the change operation and its operands. The operand is composed of the original ontology construct and the modified ontology construct. The history file records the version information, which includes the change transaction, the identifier of the ontology, the location of the ontology, the name of the previous version of the ontology, the author and the description of the ontology.

The Protégé 2000 Application Programming Interface (API) is an interface that allows an external application to access the Protégé knowledge base. As shown in Figure 4-1, the Ontology Editor Core and the Project Manager make use of this API's underlying classes and methods to manipulate the ontology model and access the ontology project respectively. The ontology project is a set of files that stores the ontology constructs and their relationships.

4.1 Ontology Editor Implementation

The Ontology Editor is implemented as a tab widget of the Protégé 2000 using the JAVA programming language. Details of each component will be discussed in separate subsections.

4.1.1 User Interface

The graphic User Interface acts as a bridge between the ontology developer and the backend Ontology Editor engine. It is composed of several sub-components: the ontology panel, the versioning information panel, the history panel, the operation panel, the toolbar and the menu.

The ontology panel allows users to view details of an ontology. It displays the main elements of an ontology in a tree structure. The main ontology constructs includes concepts, predicates and agent actions according to the FIPA 2000 specification. These constructs are represented as JTree's node. As mentioned in chapter 2, while slots describe a concept's characteristics, arguments are input parameters of predicates and agent actions. Details of these construct are shown at the UI. Three types of *replace* operations, such as the type conversion, the arithmetic conversion and the custom conversion, can be applied on a concept slot or an action argument.

The versioning information panel shows the ontology meta information such as, the name of the ontology, the version number, the author name and the location of the previous version of the ontology. The version number is updated by the History Manager whenever the modified ontology is saved.

The operation panel provides functions to add concepts, predicates and agent actions from other ontology projects. User can browse the ontology repository and use the *import* function to use other ontology project's constructs. The import function is achieved by copying the entire selected ontology construct including its concept slots or action arguments from the imported external ontology project to the current working project.

The toolbar provides a set of basic actions to manipulate the ontology, such as, add, delete, rename, edit, import, browse, save and generate adapter agent. These functions invoke the Ontology Editor engine to modify or manipulate the underlying ontology model. The adapter agent generator can be called to generate an adapter agent.

While the *add* operation is used to add an ontology construct as a child to a concept, a predicates or a agent actions node, the *delete* operation provides two different ways to delete a construct: 1) the deleted construct's child nodes are attached to its parent; and 2) the deleted construct's child node is also deleted. The *rename* operation is used to rename concepts, agent actions and predicates. In short, *add*, *delete* and *rename* are the syntactic change operations which do not affect the original construct's semantic meaning.

4.1.2 Project Manager

The Project Manager is responsible for opening and saving the ontology projects. It uses the Protégé 2000 API to perform these functions. The Project Manager maps the ontology identifier to the path of the ontology file in order to locate the appropriate project. The version number is implicitly integrated with the location path in this format: `<working path>/<ontology project identifier>/<major version number>/<minor version number>`. For example, an ontology named "`MusicShopOntology`" with version number 1.2 would be saved in the file path `<working path>/MusicShopOntology/1/2/`.

The `<working path>` can either be a path in the local driver, or a URL specifying a location in the Internet. When a new version of the ontology is created, a new directory is made to store the ontology according to the ontology identifier.

4.1.3 Ontolog Editor Core

The Ontology Editor Core provides backend functions support for the ontology manipulation. As mentioned in Section 4.1.1, these functions are *add*, *delete*, *rename*, *replace* and *import* operations. When a user interacts with the UI, it invokes the Ontology Editor Core's corresponding function to make changes to the underlying ontology model. Table 4-2 summarizes the ontology change operation to ontology constructs, concept slots and action arguments.

Table 4-2 Ontology change operation summary

Type of operation	**Operation**	**Description**
Construct ontology operation	*Add*	Use to add new concept, predicate or agent action
	Delete	Use to delete old or obsolete concept, predicate or agent action
	Rename	Use to rename concept, predicate, agent action
Slot/argument operation	*Add*	Use to add new concept slot or action argument. After adding slot or argument, a default value can be assigned.
	Delete	Use to delete new concept slot or action argument.
	Rename	Use to rename concept slot or action argument name

	Replace	Use to convert or relate old concept slot to the new slot by three types of conversion: type, arithmetic and custom.

In order to help an adapter agent to determine the slot value, a default value can be assigned to the concept slot or action argument when adding or deleting a new slot. This default value is saved in the history file. The adapter agent helps to fill in a default concept slot or action argument value because the agents using the older version of ontology may not understand the slots or arguments added in the newer version of ontology. Only after filling in a default value and forming a valid message content, agents using the older version of ontology can use the new services developed later.

In ontology construct deletion, the descendent nodes are either all deleted as shown in case (a) of Figure 4-2 or attached to the parent node as shown in case (b) of Figure 4-2. It should be noted that Protégé 2000 supports only case (a) type of deletion. OntoWrap also supports case (b) type of deletion.

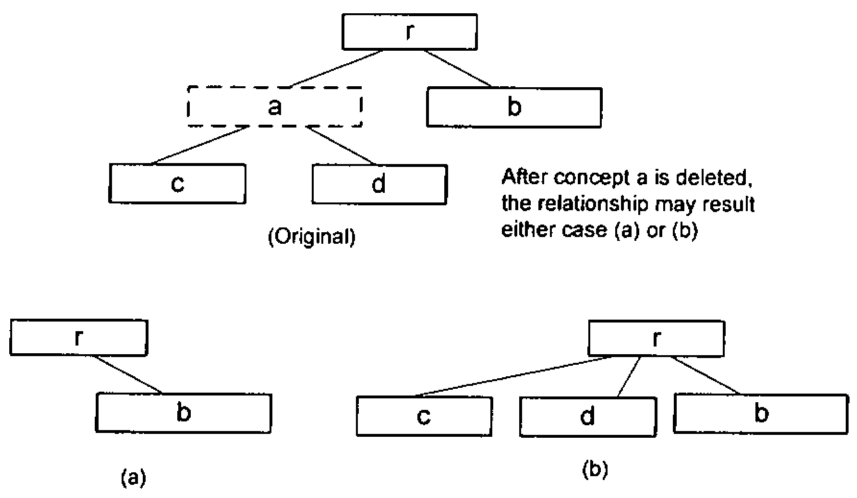

r
a
b
c
d
(Original)
After concept a is deleted, the relationship may result either case (a) or (b)
r
b
(a)
r
c
d
b
(b)

Figure 4-2 The change of concept relationship after deletion

No matter the *rename* operation is applied to the ontology construct operation, concept slot or action argument, it changes the name of the item syntactically. Since no two items in an ontology project is allowed to have the same name, the Editor Core Manager helps to validate the user input. If items with duplicate names are found, an integer is appended to the name of one item so that they will be different. For example, suppose an object named "CD" is already in the ontology. When the ontology developer names another item to "CD", the new "CD" will be modified to "CD_1". The number increases automatically. This mechanism is also used in the *add* operation to avoid two items with the same name in an ontology project.

We allow an imported construct to be added to a working project. Since ontology constructs may be related to each other, there is an option to specify whether to import relations as well. So, if a concept C_1 has a slot that refers to another concept C_2, when C_1 is imported, reference to C_2 can also be imported.

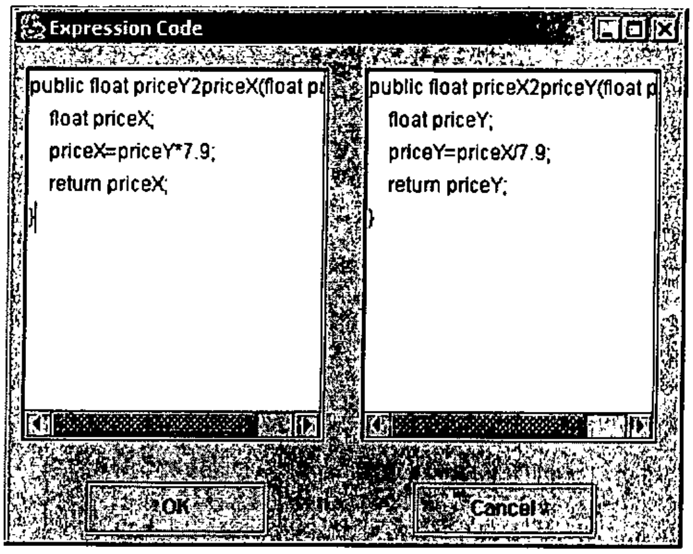
So far, all *rename* operations applied to the concept slot or the action argument are syntactic operations without changing the original meaning of the slot/argument. For example, after renaming a concept `CD` to `CompactDisc`, it still refers to the same concept semantically. However, there may be a case that the name change of a slot gives the item new meaning even if it may be related to the original one. For example, a customer record's slot `age` is changed to `theDateOfBirth` due to requirement changes. This kind of semantic change is handled by the *replace* operation. Again *replace* can be used for either: type conversion, arithmetic conversion, or custom conversion.

The type conversion is the conversion between the data types of a slot or an argument. For example, a slot named `serialID` of a concept `CD` can be converted from an integer type to a string type using replace operation. Since JAVA supports the conversion between these kinds of primitive types, the basic conversion functions is written before hand. In the ontology development phase, an ontology developer only needs to choose the type. The Ontology Editor Core selects the appropriate functions according to the source type and destination type. These functions are recorded in the history file as a set of rules. Conversion functions are implemented corresponding to each rule and is shown in the following table.

Table 4-3 Conversion Rules and Functions

Rule	**Functions**
Integer to String	`Public String Integer2String(Integer i)`
Integer to Float	`Public Float Integer2Float(Integer i)`
String to Integer	`Public Integer String2Integer(String s)`
String to Float	`Public Float String2Float(String s)`
Float to Integer	`Public Integer Float2Integer(Float f)`
Float to String	`Public String Float2String(Float f)`
Boolean to String	`Public String Boolean2String(Boolean b)`
String to Boolean	`Public Boolean String2Boolean(String s)`

The arithmetic conversion provides ways to specify arithmetic relations between slots. The arithmetic conversion provides two text fields for an ontology developer to fill in the arithmetic expression used to convert the concept slot or action argument. It requires two arithmetic expressions because translation of an adapter agent requires two ways communication. The two arithmetic expressions are parsed using a mathematical expression parser called JEP [6]. The expression is parsed and represented in a tree data structure in order to generate the slot value conversion code. For example, a slot named "price" of a concept "CD" may be converted from using the HK dollar to using the US dollar. The arithmetic expressions are `priceX=priceY*7.9` and `priceY=priceX/7.9`, in which `priceX` denotes the price in HK dollar and `priceY` denotes the price in US dollar. The resulting JAVA programs are shown is the following figure.



```
public float priceY2priceX(float p
  float priceX;
  priceX=priceY*7.9;
  return priceX;
}
```

```
public float priceX2priceY(float p
  float priceY;
  priceY=priceX/7.9;
  return priceY;
}
```

Figure 4-3 Two JAVA Functions Generated by the Ontology Editor

The custom conversion is used to provide customized conversions which may require coding. The Ontology Editor facilitates the ontology developers to write their own conversion code. In OntoWrap, only Java language is allowed since our agent platform is based on Java. The customized codes are saved in the history file by the History Manager.

4.1.4 History Manager

The History manager is composed of the History Log Generator and the Version Manager. The History Log Generator keeps the sequence of change operations during ontology editing. It then saves log and the version information of an ontology project in a history file in XML format. The Version Manager is also responsible for the version generation and management.

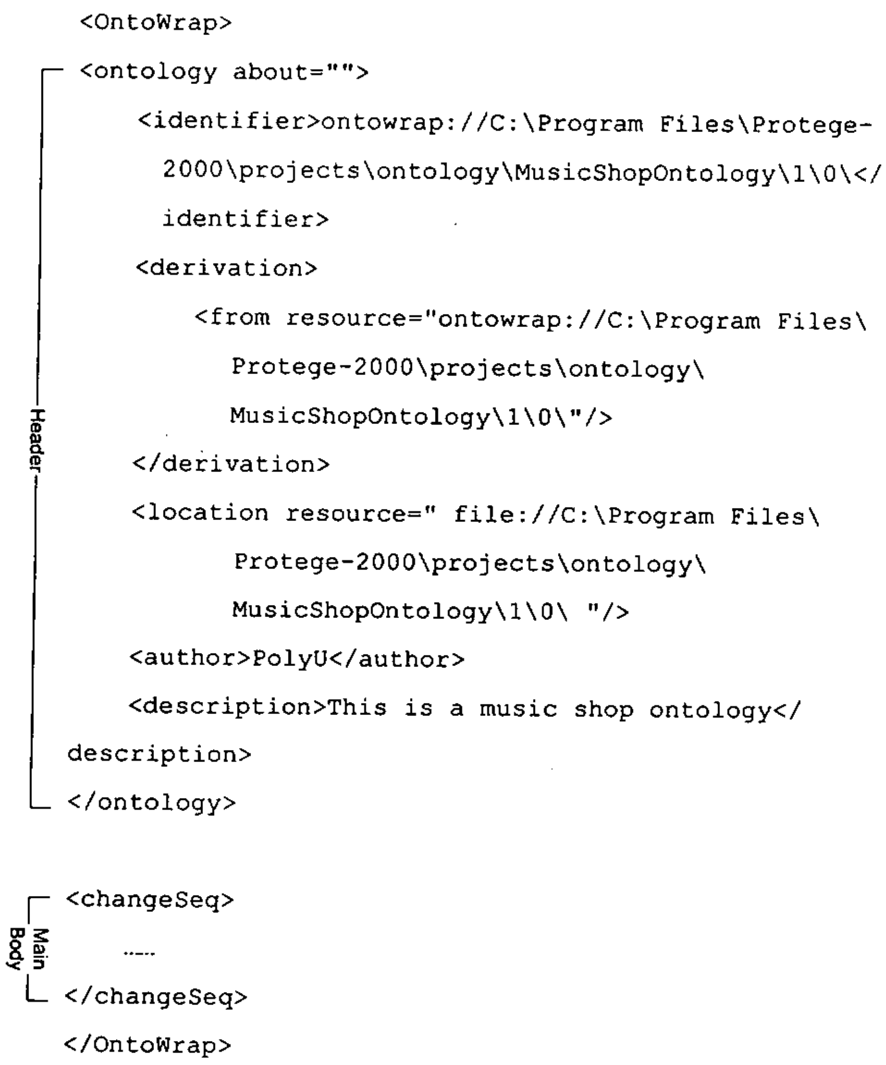
Version information includes: the identifier of the ontology, the identifier of any previous version of the ontology, the location of the ontology file, the author and the description of the ontology

According to [23], the ontology identifier uses the URI as the identification mechanism. According to the URI definition, a "generic URI" has the following format: <scheme>:// <authority><path>? <query>. In this project, we use the name "ontowrap" for the scheme, constitute the <authority> and <path> from the server name and path of a URL, and uses the major and minor number as the last two elements of the path. The location of the ontology is specified by the file identifier. The file identifier is a URL associated with the ontology identifier. Every time the ontology changes, a new ontology identifier and file identifier are specified. This has the advantage that file change and location changes can be isolated from ontological changes.

The ontology identifier uses a two level numbering scheme where the minor numbers are used for backward compatible modification and the major numbers are used for incompatible changes. In OntoWrap, it is assumed that the conceptualization will not change, only the minor number is increased by one when a new version of an ontology is created. It is the responsibility of the author to decide whether the change is compatible or not.

The above version information is generated automatically by the Version Manager. The author and the description of the ontology are passed from the User Interface to the Version Manager. Moreover, the Version Manager reads in the header of the history file and extracts the versioning information for the User Interface every time a new project is opened.

The History Log Generator saves the sequences of changes and version information in the history file. The history file is composed of two parts: the header and the body. The header contains the versioning information, i.e., the identifier of the ontology, the identifier of previous version of the ontology, the location of the ontology file, the author and the description of the ontology. The body contains the sequence of changes performed. Figure 4-4 shows a history file structure which contains header and the change sequence main body.



```
<OntoWrap>
<ontology about="">
    <identifier>ontowrap://C:\Program Files\Protege-
      2000\projects\ontology\MusicShopOntology\1\0\</
      identifier>
    <derivation>
        <from resource="ontowrap://C:\Program Files\
           Protege-2000\projects\ontology\
           MusicShopOntology\1\0\"/>
    </derivation>
    <location resource=" file://C:\Program Files\
           Protege-2000\projects\ontology\
           MusicShopOntology\1\0\ "/>
    <author>PolyU</author>
    <description>This is a music shop ontology</
description>
</ontology>

<changeSeq>
    ......
</changeSeq>
</OntoWrap>
```

Figure 4-4 History File Content

The sequence of changes is saved in an `ArrayList` during the processing of the ontology. The information needed includes the type of change, the operand and the resulting ontology item. A new class called `ChangeOp` is created to save this information. Several new classes are created for every type of elements in the ontology so that the operand and the resulting ontology item can be recorded. They are different from the classes used in the Protégé 2000. For example, the class `SlotXML` has the `Object dValue` as the attribute for the default value. The relation between slots and argument are saved in the class called ReplacedLogic. After the change is performed, the History Manager creates a new instance of ChangeOp, set the type of changes, and add the operands and resulting ontology element. Finally, the History Manager saves the two functions used to convert the slots and arguments in the ReplacedLogic if necessary. In the mean time, the History Manager passes the instance to the User Interface. The User Interface extracts the information and displays the changes in the history panel.

After the ontology evolution process is finished, the ArrayList of ChangeOp is passed to a DOM parser to form the XML document tag-by-tag and element-by-element. The parser reads in every ChangeOp in the sequence they are performed. The ChangeOp is classified according to the types of operations. The operation tag is formed in this step. Then the operand and resulting ontology element are passed to the functions to form the XML elements according to their types. Finally, the XML elements are appended as the children of the operation tag. The sequence of operation tags represents the sequences of changes performed. For example, an *add* operation that adds a concept named CD to the parent concept Item will be formed as shown in Figure 4-5. The instance of the ChangeOp is first classified into the *add* operation. An <add> tag is formed. The concept CD and its parent Item are passed to the function ConceptTag to form the XML elements for the concepts. The function also makes use of other functions such as `<Slot>` tag since the concept `CD` have slots such as `name`. Finally, the XML element `concept` is appended as the child of the `<add>` tag. The result is shown below:

```
<add>
   <concept parentConcept="Item">
       <name>CD</name>
           <slotList>
              <slot belongToConcept="CD">
                 <primitiveType type="string"/>
                 <name>DiscName</name>
              </slot>
          </slotList>
   </concept>
</add>
```

Figure 4-5 Example of <add> Tag in History File

4.1.5 *Undo Manager*

Undo Manager is designed for the undo mechanism. Generally speaking, it should allow the user to revert the last change operation they just performed. Each change operation is atomic which contains all the information to restore the pre-execution state. As shown in Figure 4-6, we make use of the Java Swing's undo support to create a set of Java classes to store all the essential information of each operation, such as `AddEdit`, `DeleteEdit`, `RenameEdit` and `ReplaceEdit`. This makes each operation is self-contained with data and effect.

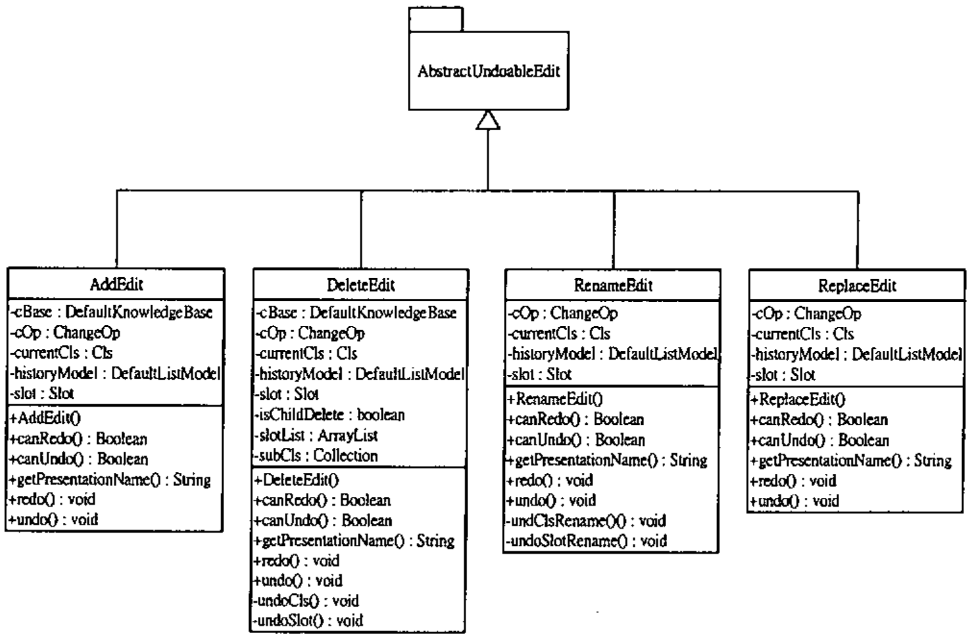


Figure 4-6 Class Diagram - Undo Support classes

Yet, it is not as simple as taking the inverse of an action. For example, there are two different ways to perform the *delete* operation, such as deleting all the child nodes or attach the child nodes to the parent node. It cannot add back the deleted node. In fact, the whole hierarchy is restored. Thus we have to keep all the operation information and the pre-execution state. In the following code shows the undo operation. It determines the appropriate undo effect to be executed and updates the editor's user interface.

```
import javax.swing.*;
import javax.swing.undo.*;
import javax.swing.event.*;
.....

public class DeleteEdit extends AbstractUndoableEdit {
  ......
  public void undo() throws CannotUndoException {
    if (slot == null){
      undoCls();
    }
    else{
      undoSlot();
    }
    historyModel.removeElementAt( historyModel.size() - 1 );
  }

// Undo Class deletion
  private void undoCls(){
    Cls parentCls = cBase.getCls(cOp.getOperand(0).getName());
    Collection parents = new ArrayList();
    parents.add( parentCls );
    currentCls = cBase.createCls(cOp.getOperand(1).getName(), parents);

// Add back deleted slot!
    if (slotList != null){
      Iterator it = slotList.iterator();
      while (it.hasNext()){
        Slot slot = (Slot) it.next();
        currentCls.addDirectTemplateSlot(slot);
      }
    }

    if (isChildDelete == false){
      // Lower one level down of the original direct subclass of
      //the current node from the parent
      Iterator it = subCls.iterator();
      while (it.hasNext()){
        Cls cls = (Cls) it.next();
        cls.addDirectSuperclass(currentCls);
        cls.removeDirectSuperclass(parentCls);
      }
    }
  }

  // Undo slot deletion
  private void undoSlot(){
    Cls parentCls = cBase.getCls(cOp.getOperand(0).getName());
    parentCls.addDirectTemplateSlot(slot);
  }
}
```

Swing provides `UndoableEditSupport` class to manage the listeners which listen to those edit events generated by each change operation. We registers our Undo Manager to the `UndoableEditListener` by `addUndableEditListener()` method. By invoking the `postEdit()` method, the edit event is sent to the listener. For example, this is shown in the `addOp()` method.

```
    public void addOp(ChangeOp op, Slot slot){
        UndoableEdit edit;
        switch(op.getActionType()){
            case ChangeOp.RENAME_ACTION:
                // Create EditUndo
                edit = new RenameEdit(historyModel, op, slot);
                undoSupport.postEdit( edit );
                break;
            case ChangeOp.DELETE_ACTION:
                // Create EditUndo
                edit = new DeleteEdit(historyModel, cBase, op, slot);
                undoSupport.postEdit( edit );
                break;
            default:
                System.err.println("addOp(): Cannot handle this
operation: " + op);
        }
        addOp(op);
    }
```

4.2 Comprehensiveness

As mentioned in Table 4-1, we have the basic ontology change operations *add*, *delete*, *rename* and *replace*. They are the *atomic change operations* which cannot be further decomposed by any simpler operations. A *complex change operation* is composed of a sequence of atomic change operations. For example, a move operation is often useful in building a slot from one specific concept to a generic concept or vice versa. At this time of implementation, OntoWrap only support a set of basic atomic change operations. A complex change operation is only carried by a sequence of atomic operations specified explicitly by users.

Having a basic set of atomic change operations, a set of complex change operations can also be defined as atomic operations in the future. Complex change operations can also be attached with additional semantic meanings and constraints. Even though some complex change operations maybe composed by the same sequence of atomic operations, they may have different semantic meanings. For example, we can distinguish a move of a slot from one class to the other class from a move of a slot along the class hierarchy. Copy-and-paste operation can also be defined as an atomic operation.

5 Service Template

We would like to make use of ontology to facilitate the development of agent based systems. We first need a platform where an agent service provider to declare and export the types of services it can provide with interfaces and methods so that such service are made public with well defined functions. We also need a platform where a client agent developer can declare and export the generic requested service. The system where the ontologies are maintained would naturally be the binding place for such declaration and exports for both service agents and client agents.

We use a so called *Service Template* construct to keep the declared service type or a declared request linked to a specific version only ontology. Each service agent implementation, must be associated with a declared service which is linked with a particular version of ontology. Then an agent build according to a declared service request, which is also linked to a certain version of ontology, would be very easy to locate a service agent as long as the service agents ontology version is made known at runtime.

A service template is associated with each declared service. It contains service information, such as, the service type, the service address, the used ontology (includes version), the service provider and the service parameters. The service parameters, which include preconditions, input and output, are directly related to the agent service invocation. Since a service template describes all the necessary information related to an agent service, it naturally binds a service with an ontology. In OntoWrap, service template is also used to generate agent implementation code skeleton. An upper level meta ontology, which we call **Service Template Ontology**, is defined as a service template description language. A service template is written using the Service Template Ontology.

A service template is further classified into two types called: the **Service Provider Template** and the **Service Request Template**. A Service Provider Template is used to describe a declared service implementation in details. The run time information of a service (carried out by a service agent), such as service address, agent identifier and service category, are contained in a service provider template. The Service Request Template is user for service request declarátion with only an abstract service description without implementation.

5.1 Service Template Ontology

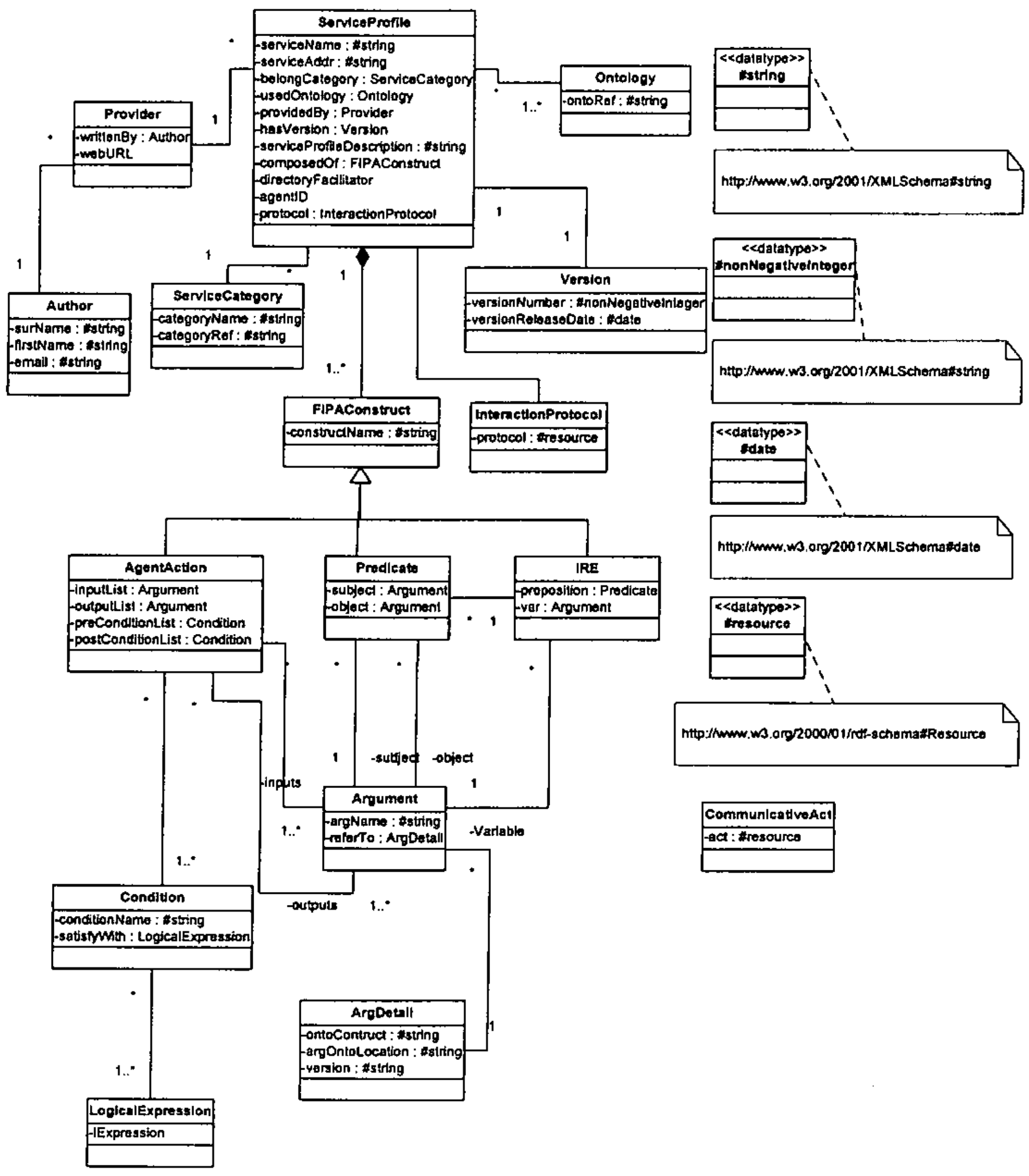


Figure 5-1Service Template Ontology Represented in UML Class Diagram

Any service template must be specified by the Service Template Ontology. The service template ontology is an upper level ontology describing an agent service. The Service Template Ontology is represented in a UML class diagram as shown in Figure 5-1 defined in the DARPA Agent Markup Language (DAML).

The `ServiceProfile` class is the main class describing all the necessary information concerning a service in the Service Template Ontology. The `ServiceProfile` class consists of service provider information, service category, basic FIPA construct, used ontology and version information. The service provider information is represented by the `Provider` class. It stores the provider name, contact e-mail and the service related web address. When there are any query concerning the agent service, users can contact the relevant provider based on this information.

The `ServiceCategory` class specifies the service category and its classification reference. A classification reference is identified by the classification system's URI. The category can be referred to the current classification system, such as, Open Directory Project. Assigning category to an agent service makes browsing more easily by client agent developers.

The `Version` class defines the service template's release date and the version number. The version number stores the major, the minor and the revision numbers. For example, the agent service and the client agent with different revision number, the existing client agent can still use the agent service without compatibility problem.

The `CommunicativeActs` class defines the acts of an agent message. As mentioned in Chapter 2, communicative acts are the basic blocks of a dialogue between agents. Although there are 22 communicative acts defined in the FIPA specification, they can be classified into *request*, *query* and *inform* acts. For example, in a shopping scenario, a buyer agent sends a query message to a seller agent for a product price. The buyer agent gets the price from the seller agent's inform message. Finally, the buyer sends a request message to buy a product. These are the basic interactions between agents.

The `FIPAConstruct` class defines the basic construct class of FIPA. It is the parent class of three main classes. They are `AgentAction` class, `Predicate` class and `IRE` class. These constructs have the corresponding communicative acts. An agent service contains a number of agent actions which are invoked during agent interaction. `AgentAction` class provides functional description such as agent action name, input argument, precondition and the output. These are important information when invoking an agent action in a service. For instance, in a virtual CD shop, a buyer agent as a client can invoke the sell action of a seller service agent. However, the buyer agent needs to provide the CD name and a valid credit card number. Then, the seller agent can return the order confirmation number to the buyer agent. `Predicate` class specifies a predicate which can be either true or false. Predicate is often used in the agent query message to check conditions. For example, if a precondition in the `Predicate` classes, the agent action can only be executed when those precondition's predicate are declared true. The `IRE` class specifies the Inferential Reference Expression. Like the `Predicate` class, it is used in the agent query message. But, it gets a reference variable value if the given condition is true. For example, a client can query price by sending a Price IRE expression by giving the CD name and the inferential variable x. When the seller agent matches the CD name (that is the given condition is true), the value of x will return to the buyer agent.

The `Ontology` class specifies the ontologies involved in the service template and stores its URI reference. For example, a CD selling agent service may involve E-Commerce ontology and the music shop ontology. These information will be recorded by the Ontology class.

The `Argument` class contains the argument name and the details of an argument, which is defined as `ArgDetail` Class. The `Argument` Class is used in the `AgentAction` class, `Predicate` class, `IRE` class since they have a set of arguments. For example, a `Predicate` class has two arguments, the subject and the object. The `ArgDetail` Class stores the reference of an ontology construct. It keeps the name of the construct, the location of the reference ontology construct and its version.

The `Condition` class contains the condition name and the logical expression. This class defines the agent action's pre-conditions and post-condition. The core of the `Condition` class is the logical expression, which is represented by the `LogicalExpression` class. The truth value of the logical expression is checked during the execution of an agent action.

The `InteractionProtocol` class defines a list of interaction protocols defined in the FIPA specification. An interaction protocol specifies how agents send the communicative acts and interact with each other. This `InteractionProtocol` class just stores the name reference. The details of interaction specification should refer to the FIPA specification during agent implementation.

5.2 Service Provider Template and Service Request Template

In a traditional client-server model, a service is first defined with an application program interface (API). The client implementation follows the service specification exactly. This is a service-provider-oriented approach. In other words, a third party client developer needs to strictly follow the service provider's service specification in order to use the corresponding service. When the service implementation is changed, the client program may need to be re-implemented. Indeed, some current system, such as CORBA and the web service, are still using this approach. A service provider constructs the service agent with the corresponding Service Provider Template.

OntoWrap does support the traditional service-provider-oriented approach, we also support the client-developer-oriented approach where client agent developers can declare its required service through the Service Request Templates to make known what clients want to service developers. Then, client agents can be implemented based on service declared by the Service Request Template. As a result, a client agent can be developed first even when the requested service is not yet implemented. In this approach, the client developer takes the active role. With the declaration of Generic Service, a service provider would know that such a service is needed. Consequently the service provider can try to either adapt their existing service to the client declared generic service request or to create a new concrete agent service implementation based on the Service Request Template specification. This also makes it possible for a new service provider to provide service for an older client agent using older version of an ontology.

Comparing with the Service Request Template, the Service Provider Template not only includes interface information and ontology construct information but also specifies information necessary for run-time identification and binding such as agent address, the registry address, the interaction protocol and the provider information. The Service Provider Template is registered to the directory facilitator so that client agents can look up the service at run-time.

5.3 Online Service Template Repository

The **Online Service Template Repository (OSTR)** maintains the declared service templates to allow users for look up using a web-based interface. OSTR is implemented as a file repository which stores the Service Template in DAML file. Users can browse the repository by category or search by keywords. The keywords are matched against the meta-description in the service templates and the basic constructs used in the template. This is similar to the Semantic Web's Universal Description, Discovery and Integration (UDDI) Directory approach.

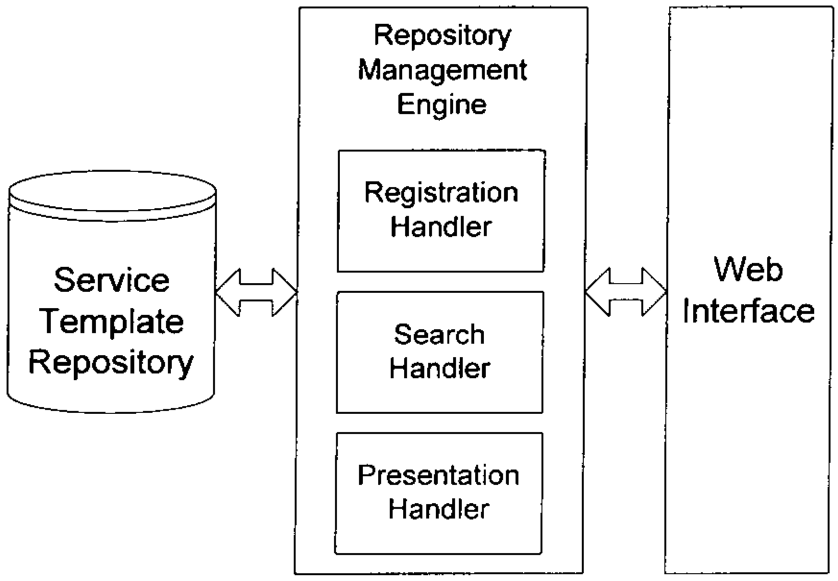


Figure 5-2 Components of the Online Service Template Repository

Figure 5-2 shows the major components of the Online Service Template Repository. It is a three-tier system with three major components: the **Service**

Template Repository, the **Repository Management Engine** and the **Web Interface**. The Service Template Repository is responsible for storing the Service Request Template and the Service Provider Template. The Repository Management Engine deals with the basic functionality of the repository such as service template registration, deregistration, and search. The Web Interface provides presentation service to online users for searching the repository. A user can access the repository by browsing the repository's web-based interface. A user can also register their service templates to the repository by uploading the template generated by the Service Template Generator. Details of Service Template Generator will be introduced in Chapter 6.

The Service Template Repository stores service templates organized service categories. We use the Open Directory Project's category schema. However, service category schemes are not limited. The repository is implemented as a simple file repository. The directory structure is indexed by the service category and the type of the service template.

The Repository Management Engine provides the basic functionalities of the repository. It consists of three sub-components: the **Registration Handler**, the **Search Handler** and the **Presentation Handler**. The Registration Handler is responsible for registering and deregistering a service template. For registration purpose, it first parses the submitted service template and gets all the necessary information in order to place the template in the repository. Then, it stores the service template in an appropriate category and creates an index entry. Deregistration of a service template requires both service template identifier and the service category. After getting this info, the Registration Handler finds a matching entry in the index table and then removes the template in the file space.

The Search Handler is responsible for searching the service template. Currently, it supports searching by keywords, service name, service category and service provider. After getting a search request from a user, it looks up the index table to match the search request. If there are matching templates, it will pass the details of the templates to the Presentation Handler to generate the result in web page.

The Presentation Handler is responsible for generating the web page. It sends the formatted web page to the Web Interface or web browser. It also provides browsing functions for the service template by reading the index table entry.

5.4 Agent Code Generation

Since each Service Provider Template is defined with service interface specification and linked ontology version. It is important that agent code generated conforms to such declarations. To ensure conformance, system provide an agent code generator to produce the skeleton code for agent implementers. The generated skeleton code ensures the external behavior of an agent implementation is understood for any agents who uses the declared version of the ontology

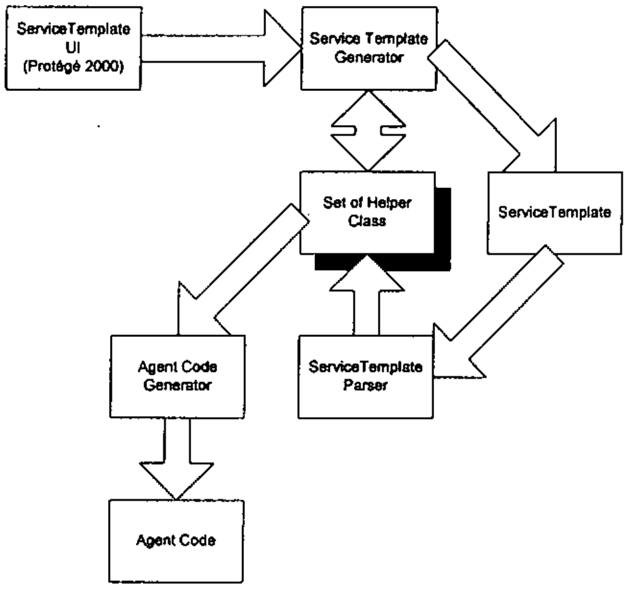


Figure 5-3 Major components of the agent code generation

Figure 5-3**Error! Reference source not found.** shows the major components involved in agent code skeleton generation. The major components are the **Service Template Generator,** a **set of helper class**, the **Service Template Parser** and the **Agent Code Generator**.

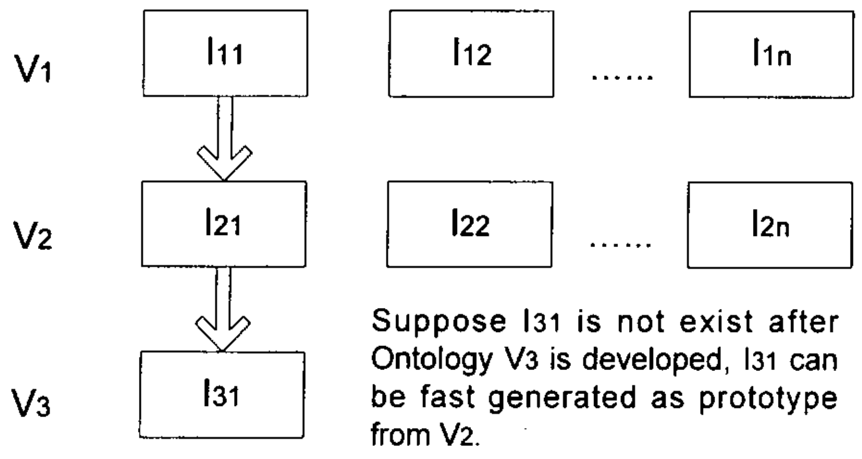
The Service Template Generator is used to export the Protégé 2000's ontology model to a service template. Since there is a gap between the ontology model and the DAML model, a set of helper classes are used to bridge this gap. The helper classes provide high level class manipulation. Its main functions are representing the Service Template model's basic elements in Java classes and creating Service Profile instance in DAML Model. These classes contains a set of core methods are summarized in Table 5-1**Error! Reference source not found.**. After the DAML model for the Service Template is constructed, the JENA API [31] is used to save the model in memory to a DAML file on hard disk.

Table 5-1 Common method of the Java Helper class

Helper class common methods	**Description**
`public DAMLJavaHandler getDAMLJavaHandler()`	Get a reference to DAML-Java Handler which helps to convert from and to DAML to JAVA class.
`Public createInstance()`	Create a DAML instance in the model.
`Public updateInstance()`	Update an existing DAML instance in the model.
`public ArrayList getPropertyValues()`	Get a set of property values from the DAML instance.

The Service Template Parser is used to parse and get the template elements from the service template's DAML file. After parsing, information such as, provider, agent actions, predicate, used ontologies and service name, are stored in the helper classes. The Code Generator generates the agent code skeleton based on the helper classes. Each agent action class and its arguments in the service profile are implemented as a Java functional method and the input arguments respectively. Since the preconditions are represented as predicates, their truth values are checked before execution of agent action.

5.5 Versioning Issue



V1
I11
I12
......
I1n
V2
I21
I22
......
I2n
V3
I31
Suppose I31 is not exist after Ontology V3 is developed, I31 can be fast generated as prototype from V2.

Figure 5-4 Ontology versioning and implementation relationship

If ontology changes, it will affect the current agent implementation since an agent may send messages to agent which use different ontology versions and thus they do not fully understand the message content. As mentioned before, ontology changes are managed by versioning which keeps track of changes. The connection between an ontology version to an implementation is linked by a declared service provider template.

A Service Template is associated with a set of specific versions of ontologies. When there is a new set of ontology version, it may have a corresponding set of agent service implementation available. As illustrated in Figure 5-4, where V represents a set of ontology versions and I represents a set of agent implementations, each ontology version may have a set of implementations. The notation V_1 means a set of ontologies of a specific version. Supposing there is another new ontology version V_2. A corresponding set of new agent implementation may be developed. Our wrapper framework tries to adapt the implementation from I_{11} to I_{21} by ontology management based on the ontologies changes. The adapter generation details will be discussed in Chapter 6. As the ontology V_3 comes out and there is no corresponding agent implementation I_{31}, it can still generate the adapter from I_{21} to serve for the future service request.

6 Agent Wrapper Framework

With the provision of linking ontology with development specification, we have designed an agent wrapper framework for automatic or semiautomatic wrapping service, to provide agent message translation service. Currently, there is some works [7] on using ontology to generate Java classes. The agent is still implemented by programmers.

Agents send messages to communicate with each other. Message contents are specified by an ontology. So that the semantic meaning of the message contents are well understood. Agents using different ontology versions may not understand either due to semantic change or syntactic change in the ontology definitions. As mentioned in Chapter 4, an ontology changes are kept track of, wrapping service can generate a so called "adapter agent" to convert ACL message from one version to the other acting as a translator for different versions of the same ontology.

6.1 Adapter Generator Design

As mentioned in chapter 3, adapter generator can generate the adapter agent that can converts the content of the ACL messages from one ontology to another provided that these two ontologies having some commonality or relationship. It operates in a semi-automatic way since not all the mapping issues can be resolved automatically.

The architecture of the Adapter Generator is shown in Figure 6-1. It is composed of three components: the DOM History File Parser, the Message Filter and the Adapter Code Generator. The DOM History File Parser parses the history file and constructs the sequence of changes performed. The sequence of changes is then passed to the Adapter Code Generator. The Adapter Code Generator generates the code of the adapter agent by filling in the parameters of the Message Filter based on the sequence of changes. The Message Filter is a set of filters that can convert the content of the ACL message.

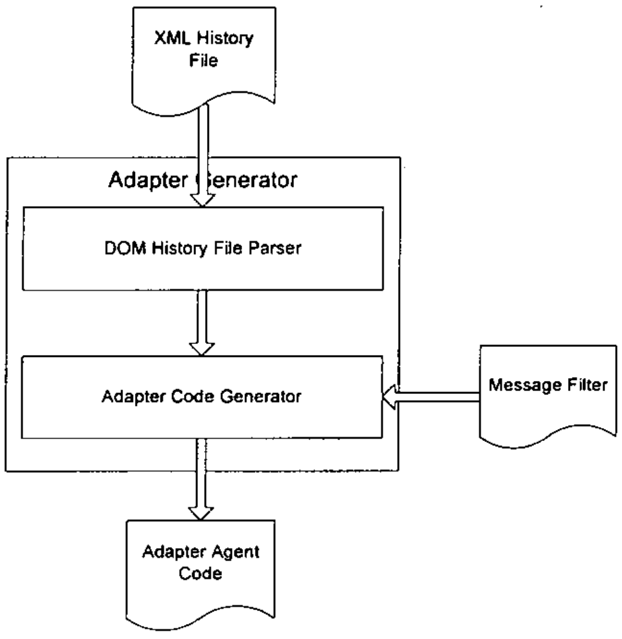


Figure 6-1 System Architecture of Adapter Generator

A message filter is a pre-defined implementation code that converts the content of the ACL message according to the changes performed to the ontology. A set of message filters are defined with reference to the ontology change operation mentioned in the chapter 4. They are *add filter*, *delete filter*, *rename filter* and *replace filter*. According to the sequence of changes performed, the corresponding operation message filter executes in the same sequence. These filters are independent of each other. Their input is the content of the ACL messages before the conversion and the output is the content of the ACL message after the conversion. Their functions can be extendedly easily without affecting each other.

Add filter is used for the slot and argument add operation. It does not support add concept, predicate or agent actions since these change the conceptualization and make the ontologies incompatible. The *add filter* simply adds the concept slot or action argument to their appropriate parent with the default value.

Delete filter performs the reverse function of the *add filter*. It removes the concept slot or action argument. Since the communication is a two way process, the delete filter and add filter are used as a pair.

Rename filter deals with the *rename* operation. In OntoWrap, the *rename* operation is considered as syntactic changes only. It does not affect the semantics of the ontology. In an ACL message, the name of the ontology item is simply a string. Therefore, the *rename filter* can replaces the name of the ontology item with only string manipulation.

Replace filter deals with the conversions of slots and arguments. This conversion is done through the functions recorded in the history file or the pre-defined functions for the type conversions.

The Adapter Code Generator generates the code of the adapter agent using the Message Filter and the skeletons of the adapter agent. It reads in the list of operations passed by the DOM History File Parser. The information is then used to fill in the parameters of the message filters.

6.2 Implementation of Adapter Generator

The Adapter Generator generates the adapter agent based on the information saved in the history file. In OntoWrap, we choose JADE as the multi-agent platform, since it is compliant with the FIPA 2000 specification. As mentioned in chapter 2, JADE is implemented in the JAVA programming language. Therefore, an adapter agent is also implemented in the JAVA programming language. This makes the adapter agent more flexible. It can run on any platform provided that it equips with JAVA virtual machine.

6.2.1 DOM History File Parser

The DOM History File Parser is responsible for the construction of a sequence of changes performed and passing the information to the Adapter Code Generator. It is implemented based on the existing XML parser Document Object Model (DOM). The parser reads in the content from the XML history file. The information is then represented by a tree structure called DOM tree. As is shown in Figure 6-2, the root of the tree is the `ontowrap` element. It has two children: `ontology` element for the header part and `changeSeq` element for the body part. The child nodes of the `changeSeq` elements represent the sequences of changes performed. The parser then traverses the sub-tree of the `changeSeq` element. The information of every change performed is extracted and saved in an instance of the `ChangeOp` class. After the traversing of the whole sub-tree is finished, the sequence of changes is saved in a `JList` instance. The `JList` instance is then passed to the Adapter Code Generator to generate the code of the adapter agent.

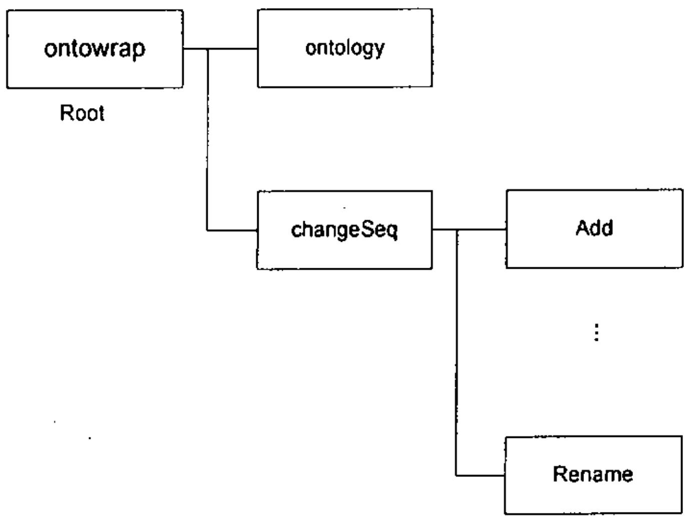


Figure 6-2 Structure of History File

6.2.2 Message Filter

The message filter is a set of filters that can convert the content of ACL messages according to changes performed to the ontology. By default, the ACL messages are encoded via string format defined by FIPA. However, an agent platform can be configured to add additional codec that can be used by agents on that platform. In this project, the XML based implementation of the codec is used. It provides extra functions and easy approach to facilitate the implementation of the message filters.

The message filters are implemented based on the Xerces2 Java Parser 2.3. The Xerces2 Java Parser 2.3 supports the APIs of both Document Object Model (DOM) and Simple API for XML (SAX) 2.0. The messages filters make use of the SAX parser to extract the content of the ACL messages. The DOM parser is then used to form the DOM tree of the message content for easy of manipulation. SAX is not used since it provides only sequential access to the XML documents and does not allow random access. After the conversion is finished, the DOM parser is used to reformat the message content and the re-formatted message is passed to the adapter agent.

The *add filter* is used to add slots and arguments to their appropriate parents. The *add filter* first calls the `createElement` function to construct the node of the concept slot or action argument. In ACL messages, the attributes of the node is represented in the sub-nodes. Therefore, a function is written to construct the sub-nodes and fill in the attributes. It uses the name of the parent to find the corresponding node in the DOM tree of the message content. The function "appendChild" is then called to actually append the node.

The *delete filter* performs the reverse function of the add filter. It removes the node of the concept slot or action argument from its parent. The delete filter finds the parent node in the DOM tree using the name. In the sub-tree of the parent node, the delete filter uses the name to find the node of concept slot or action argument. Finally, the `removeChild` function is called to remove the node.

The *rename filter* replaces the name of the ontology item according to the parameters. The parameters should include: the name of the ontology item before the replacement and the name after the replacement. In the ACL message, all the names of the ontology items are strings in text nodes. Therefore, the replacement can be done easily using the SAX parser. The SAX parser provides a callback function "characters" to handle the text nodes in the XML document. The replacement is done within the callback function with simple string manipulation.

The *replace filter* converts the value of the concept slot or the action argument. There are three types of conversions supported: primitive type conversion, arithmetic conversion and customized conversion. As mentioned in Chapter 4, the conversion functions and classes are predefined for primitive type conversion. The conversion functions of the last two types are obtained from user input. As a result, a reflection technique is used to accommodate this situation. According to the JAVA specification, reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the reflected fields, methods, and constructors are used to operate on their underlying objects within security restrictions. The replace filter takes the name of a concept slot or action argument to find the node in the DOM tree. If the conversion falls into the last two types, the conversion functions from the user input are retrieved from the history file and saved in another class with the name in a predefined format. Finally, the delete filter creates an instance of the conversion class, passes the value and type as the argument, and invokes the appropriate method to get the result. The value and the type of concept slot or action argument is replaced with the result accordingly.

6.2.3 Adapter Code Generator

The Adapter Code Generator generates the code of the adapter agent using message filters. The adapter is separated into two parts: the main part and the messing processing part. The main part deals with the basic agent operation while the message processing part converts the content of the ACL messages using the message filters. Both parts are implemented in two JAVA programs.

The main part of the adapter agent, that is `adapter.java,` contains the main body of the adapter agent. It registers the codec and ontologies used by the adapter agent. Upon receiving the request of generating the adapter agent, the Adapter Code Generator reads in the skeleton of `adapter.java` line by line and fills up the `adapter.java` according to information provided by the DOM History File Parser. The Adapter Code Generator also leaves the instructions in the `adapter.java` for the developer to fill in the blanks.

The message processing part contains two JAVA programs: `processMsg1.java` and `processMsg2.java`. They are generated simultaneously by the Adapter Code Generator from the skeletons. The skeletons contain only the definitions of classes. Which are then used by the Adapter Code Generator in the message filters according to the sequence of changes.

The Adapter Code Generator obtains the information on the sequence of changes from the `JList` of `ChangeOps` in the DOM History File Parser. These `ChangeOp` is then passed to different methods to extract their operands and resulting ontology elements according the type of the change. For example, the `ChangeOp` with the type of `ChangeOp.ADD_ACTION` is passed to `handleAdd` method. The method checks whether the change is semantic or not. As mentioned in Chapter 3, the changes in conceptualization cannot be adapted. These changes include: adding concept, agent action, and predicate as well as deleting concept, agent action and predicate. The *rename* operation is considered a syntactic change. Those *rename* operations that changes the semantics of the ontology element is not considered. For example, the concept "CD" is renamed to "DVD". Although it is a *rename* operation, the semantic meaning of the concept is changed. If the adapter simply renames the "CD" to "DVD", the buyer that wants to buy a DVD may get a CD as a result. Therefore, the adapter service will not be provided.

If the change is only syntactic change, the methods extracts the operand and resulting ontology element from the `ChangeOp`. The information is then used to fill in the parameters of the message filters. The *replace* operation needs conversion functions. As mentioned in chapter 4, these functions are either predefined or saved in the `ReplacedLogic` within the `ChangeOp`. Before filling in the parameters of the replace filter, these functions are saved in another file in the predefined format. For example, the conversion class for the primitive type `Integer` to `Float` is named `IntegerFloatConversion`.

After all the message filters are added, the message processing part is finished. The developer only needs to write the code in the "adapter.java" to register the ontologies used by the adapter agent. Finally, the adapter agent is generated.

6.3 Design of Adapter Agent Seeker

The Adapter Agent Seeker is a run time directory facilitator. When the client agents looks up a service at this directory facilitator, service agents advertise their service using the Service Provider Template here. In addition to this basic directory functionality, it tries to find a service adapter if the exact service implementation is not found. Figure 6-3 shows the interaction between client agent, service agent and the adapter agent while the client agent and the service agent use different versions of the ontology.

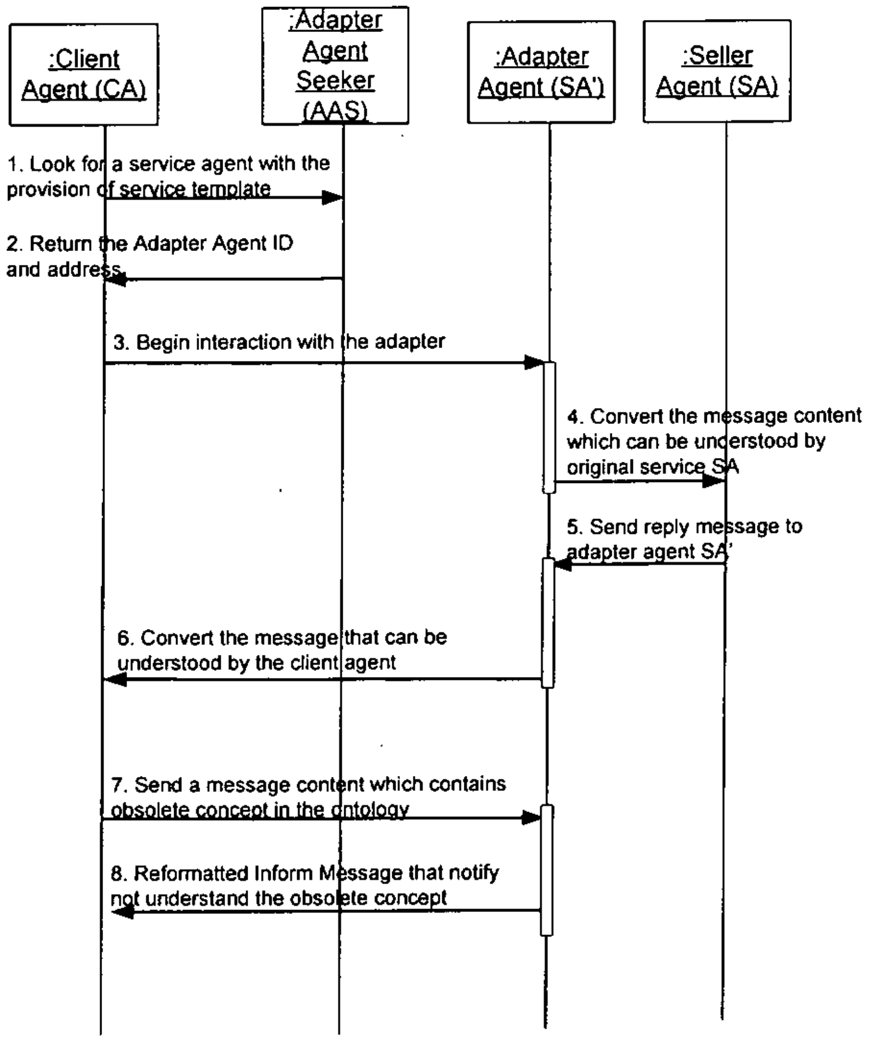

:Client Agent (CA)
:Adapter Agent Seeker (AAS)
:Adapter Agent (SA')
:Seller Agent (SA)
1. Look for a service agent with the provision of service template
2. Return the Adapter Agent ID and address
3. Begin interaction with the adapter
4. Convert the message content which can be understood by original service SA
5. Send reply message to adapter agent SA'
6. Convert the message that can be understood by the client agent
7. Send a message content which contains obsolete concept in the ontology
8. Reformatted Inform Message that notify not understand the obsolete concept

Figure 6-3 Interaction with the adapter agent

In Figure 6-3, we assume that a client agent (CA) uses Ontology O_1 and the service agent (SA) uses Ontology O_2. O_2 is a changed version derived from O_1. The client agent searches for the required agent service at the directory facilitator as shown in Step 1. The Adapter Agent Seeker tries to match the submitted the service request template from the client agent with the service provider templates registered by the service agent. If no service provider template is found, it tries to match against the service agent which has the adapter agent provided. The Adapter Agent Seeker returns the address of the adapter agent ID if the search is successful as shown in

Step 2. The client agent can then interact with the adapter agent as shown in Step 3. From a client agent's perspective, an adapter agent is simply a service agent. The translation service is transparent to the client agent. The adapter agent converts the message content using O_1 to the content using O_2 as shown in Step 4. The service agent processes the request from the client after getting the converted agent message from the client agent as shown in Step 5. CA' returns the message reply written in O_2. The adapter agent then converts the message from O_2 to O_1, so that it can be understood by the client agent as shown in Step 6.

There are some cases when an adapter agent cannot convert agent messages. For example, an obsolete concept in O_1 deleted from O_2 cannot be handled by the adapter agent. If a client agent composes a message with an obsolete concept as shown in Step 7. The adapter can only return a not-understood message to the client agent as shown in Step 8. Further processing or response of the not-understood message is up to the client implementation.

6.4 Implementation of the Adapter Agent Seeker

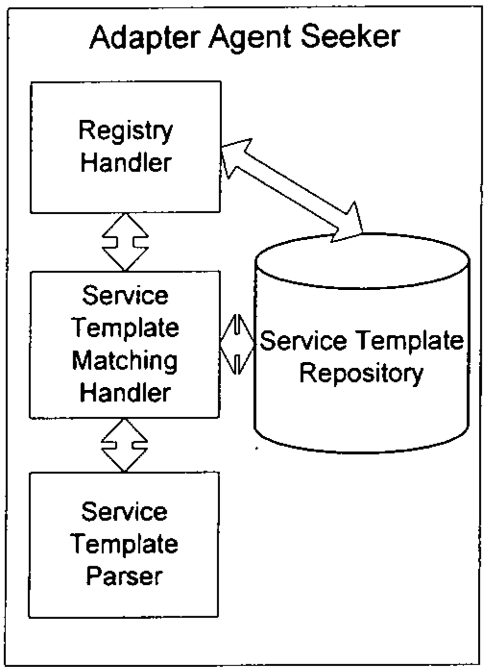

Adapter Agent Seeker
Registry Handler
Service Template Matching Handler
Service Template Repository
Service Template Parser

Figure 6-4 Components of the Adapter Agent Seeker

The Adapter Agent Seeker is composed of the **Registry Handler**, the **Service Template Matching Handler**, the **Service Template Parser** and the **Service Template Repository**, as shown in **Error! Reference source not found.**. The Adapter Agent Seeker is an extended on the JADE's Directory Facilitator's implementation.

Main functionalities of the Registry Handler are getting the service request from the client agent, receiving the service provider template registration and managing the Service Template Repository. It interacts with service agents, client agents and the Adapter Agent Seeker. The Adapter Agent Seeker extends on `jade.domain.DFService` class so that it can still provides the same service and interface to existing agents. A set of registration and searching methods are overloaded with different input arguments. Table 6-1 summarizes the overloaded methods.

Table 6-1 Methods overloaded from jade.domain.DFService class

Methods	**Description**
`public static void deregister( Agent a, AID dfName, ServiceTemplate template)`	Deregister the ServiceTemplate from an agent in a specific directory facilitator.
`public static void register( Agent a, AID dfName, ServiceTemplate template)`	Register the ServiceTemplate with an agent to a specific directory facilitator
`public static void register( Agent a, ServiceTemplate template)`	Register the ServiceTemplate with an agent.
`public static ServiceTemplate[] search( Agent a, AID dfName, ServiceTemplate template, SearchConstraints constraints)`	Search a service agent with the provision of the service template given search constraints.

The Service Template Matching Handler is the core of the Agent Adapter Seeker which tries to find a service provider template of a service agent matching the request from a client agent. The algorithm is shown as follows.

```
Search Algorithm(ServiceTemplate t){
  // Get the Service Template Category requested by the client agent
  c = get_ServiceTemplateCategory(t);
  // Get the Service Template Ontology requested by the client agent
  oList[] = get_ServiceTemplateOntology();
  // Get the Service Template Ontology Actions requested by the client agent
  actionList[] = get_ServiceTemplateAgentAction();
```

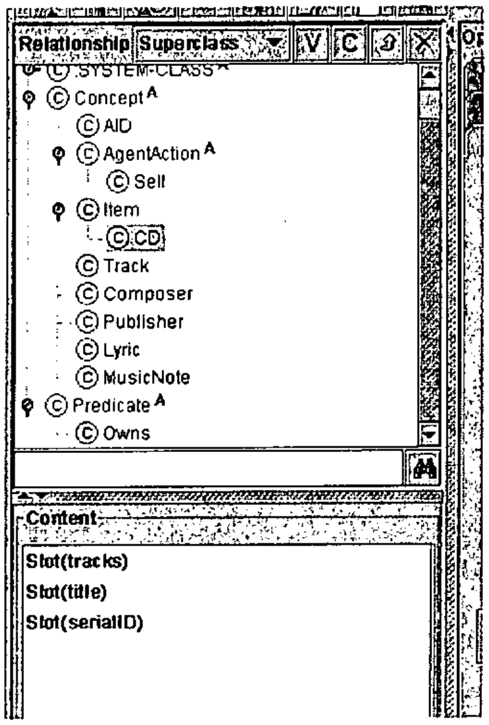
```
 //Retrieve the list of Service Provider Template from the Service Template
Repository
 serviceTmp[] = get_ConcreteServiceTemplateFromRepository(c);


 // Iterate all the shortlisted service provider template
 for (i = 0; i < serviceTmp.size; ++i){
   s = match(c, oList, actionList, serviceTmp);
   if (s is not empty){
     //return the address of the service address
     addr = get_ServiceAddress(s);
     return;
   }
 }


 // Tries to match if any adapted service available
 serviceTmp[] = get_AdaptedConcreteServiceTemplateFromRepository(c);
 // Iterate all the shortlisted concrete service template
 for (i = 0; i < serviceTmp.size; ++i){
   s = match(c, oList, actionList, serviceTmp);
   if (s is not empty){
     //return the address of the adapter agent address
     addr = get_ServiceAddress(s);
     return addr;
   }
 }
}
```

The Service Template Parser is based on the JENA API [31]. The Service Template Parser gets the service template from the Service Template Repository. Then, it creates an instance of `DAMLModel` from the JENA API. The `DAMLModel` stores the parsed content of the service template. The Service Template Parser provides some utilities functions to access those often used information, such as, service address, service agent ID, service category, directory facilitator address.

6.5 The Agent Wrapper Framework Scenario

In short, the Agent Wrapper Framework tries to automate the wrapping service as much as possible. It links ontology with the agent implementation and generates the adapter agent for the existing agent service reuse purpose.

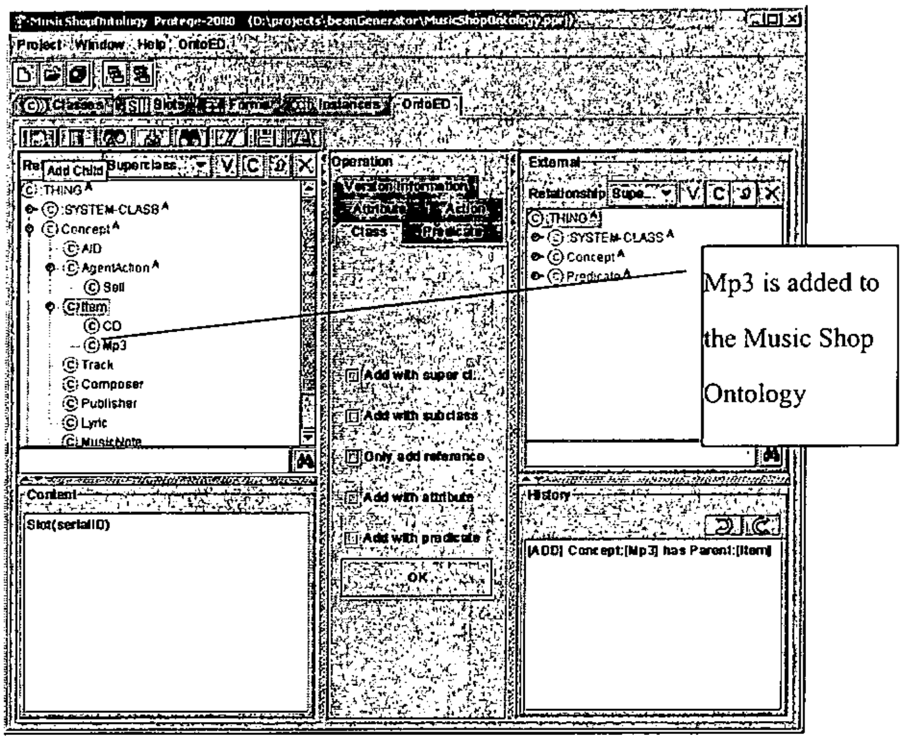
To sum up, we present a simple music shop to illustrate the features of OntoWrap. Firstly, the ontology of music shop as a service needs to be constructed using the Ontology Editor. Suppose an expert constructs the music shop by importing the music ontology and the e-commerce ontology which are already available in the ontology repository. The resulting ontology is shown in Figure 6-5.



Slot(tracks)
Slot(title)
Slot(serialID)

Figure 6-5 Music Shop Ontology construction at the Ontology Editor

Through the Ontology Editor *add* operation, they can add new concepts, such as new music media type MP3 as shown in Figure 6-6. The *delete* operation helps to remove irrelevant concepts and slots in the imported ontologies, such as music note concept. After these steps, a designated ontology is constructed for the online music shop service.



[ADD] Concept:[Mp3] has Parent:[Item]

Figure 6-6 Add a new concept to the ontology

Then, the music shop seller agent developer uses the Service Template Generator to build a service provider template which can then be exported to implementation code by using the Agent Code Generator. Since it is only a code skeleton as shown in Figure 6-7, the implementation details are written by the service developer. For example, the programmer need to write the agent interaction logic in a class called `HandleSellRequestBehaviour` to handle the sell action as shown in Figure 6-7. The template is then registered to the Online Service Template Repository.

```
public class SellService_Server extends Agent {
    private ContentManager manager = ( ContentManager )
getContentManager();

    // This agent "speaks" the SL language
    private Codec codec = new RDFCodec();

    // This agent "knows" the Music-Shop ontology
    //User need to add the ontology here
    private Ontology musicShopOntology0 =
MusicShopOntology0.getInstance();
    private Ontology musicShopOntology1 =
MusicShopOntology1.getInstance();

    //User need to specify the AIDs of the two agents here
    private String AID1 = " ";
    private String AID2 = " ";

    protected void setup() {
        manager.registerLanguage( codec );
        //user need to register the ontology here
        manager.registerOntology( musicShopOntology0 );
        manager.registerOntology( musicShopOntology1 );
        //User implement the Sell Action Behavior
        addBehaviour( new HandleSellRequestBehaviour( this )
);
    }

    protected void takeDown() {
        System.out.println( getName() + " exiting ..." );
    }
}
```

Figure 6-7 Skeleton code of the seller agent

The client agent developer who develops the buyer agent first looks for the music shop's service provider template in the Online Service Template Repository. The developer exports the service template into client agent skeleton code and fills the implementation details. The communication and binding between a client agent and a service provider agent or its adapter agent through a directory facilitator have been described in the coordination model in Chapter 3 Figure 3-1. Thus it will not be repeated.

The service provider agent and the corresponding adapter agent must first register their service with the director facilitator to make this coordination model work. The registration codes of the service agent and the adapter agent are generated automatically as shown in Figure 6-8 and Figure 6-9 respectively. When they are starting up at a agent platform by the method `setup()`, they first get the address of the Adapter Agent Seeker by the method `getAAD()` and then register their service description by the method `register()`.

```
public class SellerAgent extends GuiAgent {
  private ContentManager manager = (ContentManager)
      getContentManager();
  .......
  // This agent "knows" the Music-Shop ontology
  private Ontology ontology = MusicShopOntology.getInstance();

  protected void setup() {
    // Adapter Agent Seeker Agent ID
    AID aafAID = new AID();

    // Look for Adapter Agent Seeker
    aafAID = getAAF();
    ......

   // Add behavior to handle different kind of message
    addBehaviour(new HandleQueryBehaviour(this));
    ......
  }

  private AID getAAF(){
    AID aafAID = new AID();
    // Create Adapter Agent Seeker description and use it to look for AAF
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    .......
      while (true) {
        // Search for the Adapter Agent Seeker here
        DFAgentDescription[] result = DFService.search(this,dfd,c);
        if ((result != null) && (result.length > 0)) {
          dfd = result[0];
          break;
        }
      .......
    return aafAID;
  }

  // Register to the Agent Adapter Seeker
  private void register(DFAgentDescription desc, AID aaf) throws Exception{
    DFService.register(this, aaf, desc);
  }
}
```

Figure 6-8 Seller Agent registration code to the Adapter Agent Seeker

```
public class adapter extends GuiAgent {
    private ContentManager manager = ( ContentManager )
getContentManager();
    .......
    protected void setup(){
      //Adapter Agent Seeker Agent ID
     AID aafAID;
      // Looking for Adapter Agent Seeker address
      aafAID = getAAF();
      .....
      // Add the behavior to handle the message from buyer agent and
seller agent
      addBehaviour(new HandleMsgBehaviour(this));
    }
      .....
}
  class HandleMsgBehaviour extends CyclicBehaviour {
        public void action() {
            ACLMessage msg = receive();
            .....
            // Convert the message by the message filters
            handlemsg( msg );
    }

  //Register to Agent Adapter Seeker
  private void register(DFAgentDescription desc, AID aaf) throws
     Exception{
    DFService.register(this, aaf, desc);
  }
}
```

Figure 6-9 Adapter agent skeleton code

Message filters are also generated for the adapter agent in order to convert different version of ontology as shown in Figure 6-10. These message filters are corresponding to the change operation to an ontology. Since it is two way communications, two sets of filter are generated for messages arriving from the client agent and the service agent.

```
//Filters used to convert message from buyer to seller

DeleteFilter df = new DeleteFilter(content, "LD");

AddFilter af = new AddFilter(content, "MD");

RenameFilter renamefilter = new RenameFilter(content, "CD", "COMPACTDISK");

ReplaceFilter rf = new ReplaceFilter(temp, "duration", new

   FloatIntegerConversion(), Integer.class, "Integer", "Float");


//Filters used to convert message from seller to buyer

AddFilter af = new AddFilter(content, "LD");
```

```
DeleteFilter df = new DeleteFilter(content, "MD");
RenameFilter renamefilter = new RenameFilter(content, "COMPACTDISK", "CD");
ReplaceFilter rf = new ReplaceFilter(temp, "duration", new
   FloatIntegerConversion(), Float.class,   "Float", "Integer");
```
Figure 6-10 Message filters

Currently, the registration of adapter agent is done separately from a provider agent. It is possible that an adapter agent is running but there is no corresponding provider available at runtime. The next step is to automatically associate an adapter agent with a provider agent so that they can be registered in a coordinated way.

7 Conclusion and Future Work

In this thesis, we investigated issues related improving quality in service discovery. Service discovery is the use of designated agent directory server for agents to locate their required agent services autonomously based on the nature of tasks required by agents. Currently, directory services only provide run-time binding for an agent to a predefined agent service of which the agent has complete knowledge before hand. By the provision of ontology and ontology management, the specification of agent services are linked to ontology. Enhancement to services are also kept track of through version controls in ontology management. In addition, agents can declare its required services through a language using a defined version the ontology. Thus making it possible for new service providers to serve existing clients as needed services are declared explicitly. The technology developed involves locating relevant ontology, accessing and storing the ontology, keeping track of ontology changes, declaration of service through ontology, mapping services to implementation, automatic and semi-automatic wrapping for different versions of services and finally, at run-time, the binding of agents to services and online directory service for agent mapping with respect to a service type.

Currently, the Ontology Editor only supports basic change operations. However, the granularity of these basic changes is at the lowest level and may not always be appropriate. As filter is based on these basic changes and a filter is built for each change operation, the resulting filter sequence can be repetitive and inefficient. Composite changes as atomic operation for coarse-grained operations can be added to make update ontology more semantically sound meaningful to avoid going through every single step of a sequence of basic changes. The auditing information can also be implemented as part of the version information saved in the history file.

There are limitations on the adapter agent ability to convert agent messages from one version to another. For example, when an obsolete concept is deleted, the service agent which uses a new ontology cannot understand the obsolete concept. The adapter agent currently only returns the not-understood message to the client agent. In the future, this issue can be further investigated so that a much more intelligent response can be produced for client agents. A knowledge base can be created to allow adapter agent to infer any semantically close substitution rather than simply giving a fail response.

Reference

1. Fujitsu Limited, Hewlett-Packard, IBM, InterX PLC, Spydell, Andy, Suguri, Hiroki, Sun Microsystems, Inc., Tolety, Siva, Perraju, University of West Florida, Institute of Human-Machine Cognition, *Java Agent Services Initiative*, [http://java.sun.com/aboutJava/communityprocess/jsr/jsr_087_jas.html]

2. Foundation for Intelligent Physical Agent, *FIPA 2000 Specification*, 2000, [http://www.fipa.org/specifications/index.html]

3. Foundation for Intelligent Physical Agents, *XC00001J FIPA Abstract Architecture Specification*, 2001,

4. Foundation for Intelligent Physical Agents, *XC00023 FIPA Agent Management Specification*, 2001,

5. Foundation for Intelligent Physical Agents, *XC00061 FIPA ACL Message Structure Specification*, 2001,

6. *JAVA Mathematical Expression Parser*. 2002, Singular System.

7. *Ontology Bean Generator for Jade 2.5*. 2002, IBROW Project, Universiteit van Amsterdam.

8. *Tryllian Agent Development Kit*. 2002, Tryllian.

9. Alberto Rodrigues da Silva, A.R., Dwight Deugo, Miguel Mira da Silva, *Towards a Reference Model for Surveying Mobile Agent Systems*. Autonomous Agents and Multi-Agent Systems, 2001. **4**(3): p. 187-231.

10. Bellavista, P., Corradi, A., and Stefanelli, C. *A secure and open mobile agent programming environment*. in *Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems (ISADS '99)*. 1999. Tokyo, Japan.

11. Bellavista, P., Corradi, A., Stefanelli, C., *Mobile agent middleware for mobile computing*, in *Computer*. 2001. p. 73-81.

12. Bellifemine, F., Poggi, A., Rimassa, G. *JADE - A FIPA-compliant agent framework*. in *Practical Application of Intelligent Agents and Multi-Agents*. 1999. London.

13. Brugali, D. and Sycara, K., *Towards agent oriented application frameworks*. ACM Computing Surveys (CSUR), 2000. **32**(1es): p. 21.

14. Bumer, C., et al. *Grasshopper - A universal agent platform based on OMG MASIF and FIPA standards*. 1999.

15. Cheyer, A. and Martin, D., *The Open Agent Architecture*. Journal of Autonomous Agents and Multi-Agent Systems, March 2001. 4(1): p. 143-148.

16. *Exteca*, 2003, [http://exteca.sourceforge.net/]

17. Graves, A., Lalmas, M., and Stutt, A., *OntoWeb Deliverable 9.2.1*. 2003.

18. Gruber, T.R., *Towards Principles for the Design of Ontologies Used for Knowledge Sharing*, in *Formal Ontology in Conceptual Analysis and Knowledge Representation*. 1993, Kluwer Academic Publishers.

19. Hameed, A., Sleeman, D., and Preece, A. *OntoManager:A Workbench Environment to facilitate Ontology Management and Interoperability*. in *Workshop on Evaluation of Ontology-based Tools at the 13th International Conference on Knowledge Engineering and Knowledge Management*. October, 2002. Siguenza, Spain.

20. Heflin, J. and Hendler, J. *Searching the Web with SHOE*. in *Artificial Intelligence for Web Search Workshop (AAAI Workshop)*. 2000. Menlo Park, CA: AAAI Press.

21. *Semantic Network Ontology Base*, 2003, [http://www.alphaworks.ibm.com/tech/snobase]

22. Klein, M. *Supporting evolving ontologies on the internet*. in *Proceedings of the EDBT 2002 PhD Workshop*. 2002. Prague, Czech Republic.

23. Klein, M. and Fensel, D. *Ontology versioning for the semantic web*. in *Proceedings of the International Semantic Web Working Symposium (SWWS)*. 2001. Stanford University, California, USA.

24. Klein, M., et al. *Ontology versioning and change detection on the web*. in *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*. 2002. Siguenza, Spain.

25. Labrou, Y.F., T. Yun Peng, *Agent Communication Languages: the Current Landscape*. Intelligent Systems, IEEE, 1999. 14(2): p. 45-52.

26. Lange, D. and Isguna, M., *Programming and Deploying Java Mobile Agents with Aglets*. 1998: Addison-Wesley.

27. Ludwig, S.A. and van Santen, P. *A Grid Service Discovery Matchmaker Based on Ontology Description*. in *EuroWeb 2002 Conference*. December, 2002. Oxford, UK.

28. Maedche, A., *Ontology learning for the semantic web*. 2002: Kluwer Academic Publisher.

29. Maedche, A., et al. *Managing multiple ontologies and ontology evolution in Ontologging*. in *Proceedings of the Conference on Intelligent Information Processing*. 2002. Montreal, Canada: Kluwer Academic Publishers.

30. Marques, P., et al. *Providing applications with mobile agent technology*. in *Open Architectures and Network Programming Proceedings*. 2001: IEEE.

31. McBride, B. *Jena: Implementing the RDF Model and Syntax Specification*. in *Semantic Web Workshop (WWW2001)*. 2001.

32. McGuinness, D.L., et al. *The Chimaera Ontology Environment*. in *the Seventeenth National Conference on Artificial Intelligence Proceedings*. 2000. Austin, Texas.

33. Milojicic, D., et al. *MASIF The OMG Mobile Agent System Interoperability Facility*. in *Proceedings of the International Worshop on Mobile Agents (MA'98)*. 1998. Stuttgart.

34. Noy, N.F., Fergerson, R.W., and Musen, M.A. *The knowledge model of Protege-2000: combining interoperability and flexibility*. in *12th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*. 2000. Juan-les-Pins, France.

35. Nwana, H.S., Ndumu, D.T., and Lee, L.C. *ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems*. in *Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems (PAAM'98)*. 1998. London, UK.

36. Pinto, H.S., Gomez-Perez, A., and Martins, J.P. *Some Issues on Ontology Integration*. in *Proceedings of the IJCAI-99's Workshop on Ontologies and Problem-Solving Methods (KRR5)*. 1999. Stockholm, Sweden.

37. Pinto, H.S. and Martins, J.P., *A methodology for ontology integration*, in *Proceedings of the international conference on Knowledge capture*. 2001, ACM Press. p. 131-138.

38. S.Poslad, Buckle, P., and Hadingham, R. *The FIPA-OS agent platform: Open Source for Open Standards*. in *Proceedings of the 5th International*

Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents. 2000. Manchester, UK.

39. Stojanovic, L. and Motik, B. *Ontology evolution within ontology editors*. in *Proceedings of the OntoWeb-SIG3 Workshop at the 13th International Conference on Knowledge Engineering and Knowledge Management*. 2002.

40. Wache, H., et al. *Ontology-based integration of information - a survey of existing approaches*. in *IJCAI Proceedings of the Workshop Ontologies and Information Sharing*. 2001. Seattle, USA.

41. Waldo, J., *Mobile code, distributed computing, and agents*. Intelligent Systems, IEEE, 2001. **16**(2): p. 10-12.

42. Weiss, G., *Multiagent systems: a modern approach to distributed artificial intelligence*. 1999: MIT Press.

43. Wooldridge, M., *An Introduction to Multiagent Systems*. 2002: John Wiley & Sons.

Appendix

A1 Ontology Editor

Installation

1. Download the Protégé 2000 from http://protege.stanford.edu/download/release/index.html

2. Install the Protégé 2000

3. Download the Bean Generator plug-in from http://www.swi.psy.uva.nl/usr/aart/beangenerator/ This plug-in is used to convert the ontology to JAVA beans to be used by the agent

4. Install the Bean Generator plug-in according to the instructions in the homepage.

5. Download and install the Xerces2 JAVA parser for XML from http://xml.apache.org/#xerces. This parser provides APIs of both SAX and DOM implementation.

6. Download and install the Java Mathematical Expression Parser from http://www.singularsys.com/jep/. This parser is used to parse the mathematical expression input by the user.

7. Put the JAR file "tabnew.jar" in the directory "plugins" under the home directory of the Protégé 2000.

User Guide

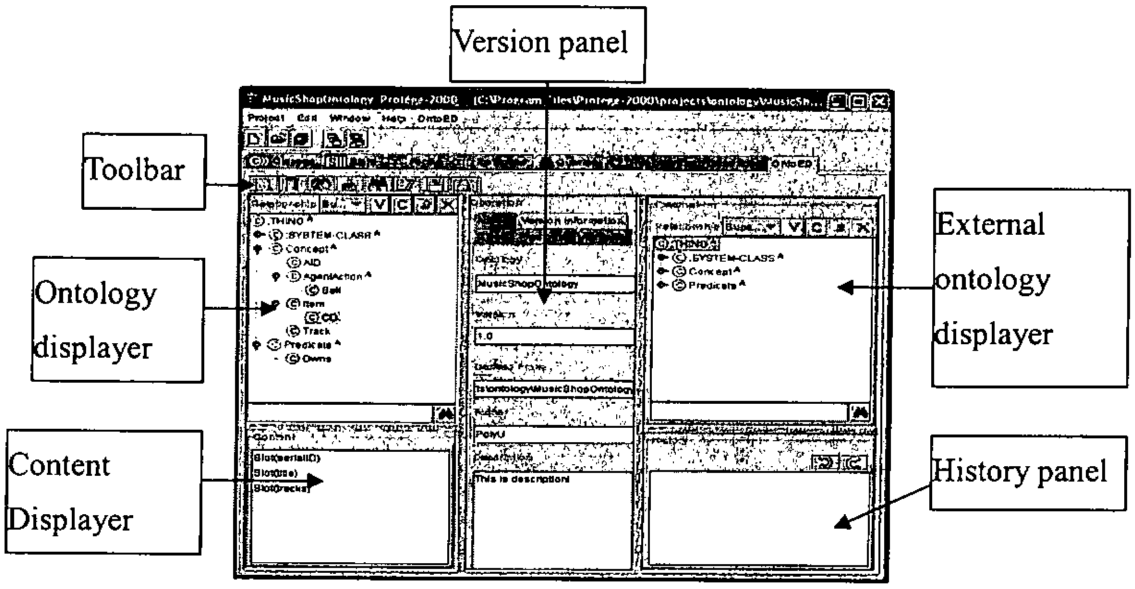


Figure A-1 Ontology Editor User Interface

1. The toolbar provides a set of button for the elementary changes and save, generate adapter operation.

2. The ontology displayer displays the main ontology element such as concepts, predicates and agent actions.

3. The content displayer displays the slots or arguments of the selected ontology element.

4. The version panel displays the versioning information such as version number, author, location and description.

5. The external ontology displayer displays the content of the ontology. The user can import ontology elements from this panel.

6. The history panel displays the change sequence performed.

A2 Adapter Generator

User Guide

1. After the ontology evolution process is finished, user needs to save the ontology first.

2. The new version of ontology will be saved according to name of the ontology and the version number. The path is displayed in the version panel.

3. Use the bean generator to generate the JAVA beans for both versions of ontologies.

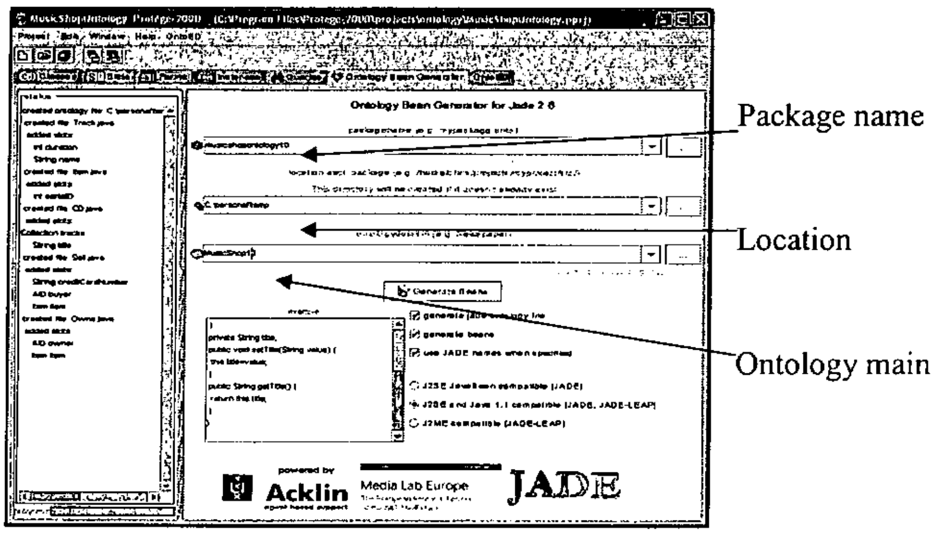


Figure A-2 Bean Generator for Protégé 2000

4. In package name and ontology main, fill in the name of the ontology with the version number. Choose the appropriate location. The version number is used since in adapter agent, both version of the ontology with the same name may be used. The version number is appended to make a distinction.

5. Press the "Generate" button. The source code of the JAVA beans is put in the location specified.

6. Press the adapter generator button in the toolbar of the ontology editor

7. The source code of "adapter.java", "processMsg1.java" and "processMsg2.java" are saved in the same directory as the new version of ontology.

8. Edit the file "adapter.java". Under the line "//User need to add the ontology here", fill in the two names used in the ontology main in the bean generator. Under "//User need to specify the AIDs of the two agents here", fill in the two agent IDs.

9. Compile the source code of the ontology and adapter agent.

10. The adapter agent can be executed on the agent platform JADE.

A3 Service Template Ontology Specification

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:xsd="http://www.w3.org/2001XMLSchema#"
xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
xmlns:ss="http://www.comp.polyu.edu.hk/~csktng/daml/ServiceProfile"
xml:base="http://www.comp.polyu.edu.hk/~csktng/daml/ServiceProfile">
    <daml:Ontology>
        <daml:versionInfo>ServiceProfile.daml v0.1</daml:versionInfo>
        <daml:comment>Service Profile Schema</daml:comment>
        <daml:imports
rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns"/>
        <daml:imports
rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
    </daml:Ontology>
    <daml:Class rdf:ID="ServiceProfile">
        <rdfs:label>Service Profile</rdfs:label>
    </daml:Class>
    <daml:DatatypeProperty rdf:ID="ServiceName">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
    <daml:ObjectProperty rdf:ID="serviceAddr">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="belongCategory">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range rdf:resource="#ServiceCategory"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="usedOntology">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range rdf:resource="#Ontology"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="providedBy">
```

```
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range rdf:resource="#Provider"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="hasVersion">
        <rdfs:label>hasVersion</rdfs:label>
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range rdf:resource="#Version"/>
    </daml:ObjectProperty>
    <daml:DatatypeProperty rdf:ID="ServiceProfileDescription">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
    <daml:ObjectProperty rdf:ID="composedOf">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range rdf:resource="#FIPAConstruct"/>
    </daml:ObjectProperty>
    <daml:DatatypeProperty rdf:ID="directoryFacilitator">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
        <daml:DatatypeProperty rdf:ID="directoryFacilitator">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
    <daml:DatatypeProperty rdf:ID="agentID">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
    <daml:ObjectProperty rdf:ID="protocol">
        <daml:domain rdf:resource="#ServiceProfile"/>
        <daml:range rdf:resource="#InteractionProtocol"/>
    </daml:ObjectProperty>
    <!--
 Author
```

```
  Organization
 //-->
     <daml:Class rdf:ID="Provider">
   </daml:Class>
     <daml:Class rdf:ID="Author">
         <rdfs:label>Author</rdfs:label>
     </daml:Class>
     <rdf:ObjectProperty rdf:ID="writtenBy">
         <daml:domain rdf:resource="#Provider"/>
         <daml:range rdf:resource="#Author"/>
     </rdf:ObjectProperty>
     <daml:DatatypeProperty rdf:ID="surname">
         <daml:domain rdf:resource="#Author"/>
         <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <daml:DatatypeProperty rdf:ID="firstname">
         <daml:domain rdf:resource="#Author"/>
         <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <daml:DatatypeProperty rdf:ID="email">
         <daml:domain rdf:resource="#Author"/>
         <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <daml:DatatypeProperty rdf:ID="webURL">
         <daml:domain rdf:resource="#Provider"/>
         <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <!--
Versioning Class
//-->
     <daml:Class rdf:ID="Version">
         <rdfs:label>Version</rdfs:label>
     </daml:Class>
     <daml:DatatypeProperty rdf:ID="versionNumber">
```

```
          <daml:domain rdf:resource="#Version"/>
          <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <daml:DatatypeProperty rdf:ID="versionReleaseDate">
          <daml:domain rdf:resource="#Version"/>
          <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#date"/>
     </daml:DatatypeProperty>
     <!--
Service Category Class
//-->
     <daml:Class rdf:ID="ServiceCategory">
      </daml:Class>
     <daml:DatatypeProperty rdf:ID="categoryName">
          <daml:domain rdf:resource="#ServiceCategory"/>
          <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <daml:DatatypeProperty rdf:ID="categoryRef">
          <daml:domain rdf:resource="#ServiceCategory"/>
          <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <!--
FIPA Construct
//-->
     <daml:Class rdf:ID="FIPAConstruct"/>
     <!--
  AgentAction
   - Arguments
   - Conditions
//-->
     <daml:Class rdf:ID="AgentAction">
          <rdfs:subClassOf rdf:resource="#FIPAConstruct"/>
     </daml:Class>
     <daml:DatatypeProperty rdf:ID="constructName">
          <daml:domain rdf:resource="#FIPAConstruct"/>
```

```
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
    <daml:ObjectProperty rdf:ID="inputList">
        <daml:domain rdf:resource="#AgentAction"/>
        <daml:range rdf:resource="#Argument"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="outputList">
        <daml:domain rdf:resource="#AgentAction"/>
        <daml:range rdf:resource="#Argument"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="preConditionList">
        <daml:domain rdf:resource="#AgentAction"/>
        <daml:range rdf:resource="#Condition"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="postConditionList">
        <daml:domain rdf:resource="#AgentAction"/>
        <daml:range rdf:resource="#Condition"/>
    </daml:ObjectProperty>
    <!--
   Predicate (IOTA)
      - Subject
      - Object
 //-->
    <daml:Class rdf:ID="Predicate">
        <rdfs:subClassOf rdf:resource="#FIPAConstruct"/>
    </daml:Class>
    <daml:ObjectProperty rdf:ID="subject">
        <daml:domain rdf:resource="#Predicate"/>
        <daml:range rdf:resource="#Argument"/>
    </daml:ObjectProperty>
    <daml:ObjectProperty rdf:ID="object">
        <daml:domain rdf:resource="#Predicate"/>
        <daml:range rdf:resource="#Argument"/>
    </daml:ObjectProperty>
    <!--
   Identifying Referential Expression class
     - proposition
```

```
      - variable
 //-->
     <daml:Class rdf:ID="IRE">
         <rdfs:subClassOf rdf:resource="#FIPAConstruct"/>
     </daml:Class>
     <daml:ObjectProperty rdf:ID="proposition">
         <daml:domain rdf:resource="#IRE"/>
         <daml:range rdf:resource="#Predicate"/>
     </daml:ObjectProperty>
     <daml:ObjectProperty rdf:ID="var">
         <daml:domain rdf:resource="#IRE"/>
         <daml:range rdf:resource="#Argument"/>
     </daml:ObjectProperty>
     <!--
   Argument
   An argument can be used for input or output and is associated with values
 //-->
     <daml:Class rdf:ID="Argument"/>
     <daml:DatatypeProperty rdf:ID="argName">
         <daml:domain rdf:resource="#Argument"/>
         <daml:range
 rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <daml:ObjectProperty rdf:ID="referTo">
         <daml:domain rdf:resource="#Argument"/>
         <daml:range rdf:resource="#ArgDetail"/>
     </daml:ObjectProperty>
     <daml:Class rdf:ID="ArgDetail"/>
     <!-- Construct Identifier //-->
     <daml:DatatypeProperty rdf:ID="ontoConstruct">
         <daml:domain rdf:resource="#ArgDetail"/>
         <daml:range
 rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     </daml:DatatypeProperty>
     <!-- Ontology Location //-->
     <daml:ObjectProperty rdf:ID="argOntologyLocation">
         <daml:domain rdf:resource="#ArgDetail"/>
         <daml:range
```

```
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
    </daml:ObjectProperty>
    <!-- Version: Should have a complex type //-->
    <!-- Simplified it here //-->
    <daml:ObjectProperty rdf:ID="version">
        <daml:domain rdf:resource="#ArgDetail"/>
        <daml:range
rdf:resource="http://www.w3.org/2000/01/rdf-schema#string"/>
    </daml:ObjectProperty>
    <!-- Condition should have a set of predefined states //-->
    <daml:Class rdf:ID="Condition">
  </daml:Class>
    <daml:DatatypeProperty rdf:ID="conditionName">
        <daml:domain rdf:resource="#Condition"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
    <daml:ObjectProperty rdf:ID="satisfyWith">
        <daml:domain rdf:resource="#Condition"/>
        <daml:range rdf:resource="#LogicalExpression"/>
    </daml:ObjectProperty>
    <!--
  Logical Expression
//-->
    <daml:Class rdf:ID="LogicalExpression">
  </daml:Class>
    <daml:DatatypeProperty rdf:ID="lExpression">
        <daml:domain rdf:resource="#LogicalExpression"/>
        <daml:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
    </daml:DatatypeProperty>
    <!--
  InteractionProtocol
    Define a list of protocol: reference JADE!
//-->
    <daml:Class rdf:ID="InteractionProtocol">
      </daml:Class>
    <!--
  CommunicativeAct complies with FIPA
 //-->
  <daml:Class rdf:ID="CommunicativeAct">
    <rdfs:comment>Communicative Act is defined in FIPA</rdfs:comment>
    <daml:oneOf rdf:parseType="daml:collection">
      <daml:Thing rdf:ID="Accept_Proposal"/>
      <daml:Thing rdf:ID="Agree"/>
      <daml:Thing rdf:ID="Cancel"/>
      <daml:Thing rdf:ID="CFP"/>
      <daml:Thing rdf:ID="Confirm"/>
      <daml:Thing rdf:ID="Disconfirm"/>
      <daml:Thing rdf:ID="Failure"/>
      <daml:Thing rdf:ID="Inform"/>
      <daml:Thing rdf:ID="InformIf"/>
      <daml:Thing rdf:ID="InformREF"/>
      <daml:Thing rdf:ID="NotUnderstood"/>
      <daml:Thing rdf:ID="Propagate"/>
      <daml:Thing rdf:ID="Propose"/>
      <daml:Thing rdf:ID="Proxy"/>
      <daml:Thing rdf:ID="QueryIf"/>
      <daml:Thing rdf:ID="QueryREF"/>
      <daml:Thing rdf:ID="Refuse"/>
      <daml:Thing rdf:ID="RejectProposal"/>

      <daml:Thing rdf:ID="Request"/>
      <daml:Thing rdf:ID="RequestWhen"/>
      <daml:Thing rdf:ID="RequestWhenever"/>
      <daml:Thing rdf:ID="Subscribe"/>
      <daml:Thing rdf:ID="Unknown"/>
    </daml:oneOf>
  </daml:Class>
    <!--
  Ontology
  Note that more than one ontologies can be used in a service template.
//-->
    <daml:Class rdf:ID="Ontology">
  </daml:Class>
    <daml:DatatypeProperty rdf:ID="ontoRef">
```

```
        <daml:domain rdf:resource="#Ontology"/>
        <daml:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    </daml:DatatypeProperty>
</rdf:RDF>
```