MULTI-AGENT SYSTEMS AND THE SEMANTIC WEB
The SemanticCore Agent-based Abstraction Layer

Marcelo Blois Ribeiro
PUC-RS, Informatics Faculty,
Av. Ipiranga 6681, Partenon, Porto Alegre, Rio Grande do Sul, Brazil

Carlos J. P. de Lucena
PUC-Rio, Computer Science Department,
Rua Marquês de São Vicente 225, Gávea, Rio de Janeiro, Brazil

Keywords: Semantic Web, Agents, Multi-agent Systems

Abstract: In the Web first years, it was claimed that it would revolutionize the way people work and learn by creating a rich information environment where everybody would cooperate through content publish and recovering. This promising model showed its limitations with the information explosive growth. Many initiatives were taken to address this problem, but none of them gained such attention as the Semantic Web proposal. The combination of machine understandable content with human oriented content can avoid information overload and create a new set of possibilities in terms of software development and integration. Although the Semantic Web is on its very beginning, some proposals already address some requirements for the Semantic Web creation. This paper presents the SemantiCore agent-based abstraction layer for the Semantic Web. The SemantiCore uses high level agent-based abstractions to create applications for the semantic web. SemantiCore uses the middleware concept to allow the integration with well known technologies such as the FIPAOS platform and the Web Services standards.

1 INTRODUCION

Over the recent years there has been a growth in the number of commercial and academic initiatives to capture and fulfill the Semantic Web requirements. Content annotation tools, ontology editors and inference engines where created to enable the development of applications that can benefit from the Semantic Web characteristics. None of these initiatives concentrate efforts on the provision of a complete set of abstractions to enable the fast application development combining all the efforts previously done.

The exploration of the Internet power to provide a network of interconnected services motivated the creation of standards and technologies such as RMI, J2EE (Kurniawan, 2002), CORBA (OMG, 2002) and Web Services (Champion et al, 2003). All these mechanisms offer an abstraction level for developers, hiding unnecessary distribution details and allowing them to concentrate in the application business logic.

It is possible to find common points among these initiatives. They usually present capabilities such as authentication, service description, service search, communication among peers and message exchange protocols. Service distribution is essential to the Semantic Web, since distributed applications or agents can communicate with each other and exchange machine-understandable content. Software agents are used in the Semantic Web as an entity capable of automatically consume published content. Thus, the Semantic Web can be thought as a global multi-agent system formed by the relation of a large number of agent societies.

Considering the Semantic Web a neighborhood of agent societies, it is possible to benefit from the multi-agent development proposals (Ferber, 1999) to form a consistent infrastructure for Semantic Web application development.

Blois Ribeiro M. and J. P. de Lucena C. (2004).
MULTI-AGENT SYSTEMS AND THE SEMANTIC WEB - The SemanticCore Agent-based Abstraction Layer.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 263-270
DOI: 10.5220/0002635102630270

The evolution of agent-based systems with automatic decision capabilities guided the efforts to build methods and tools to deal with the complex details of agents (Zambonelli, Jennings & Omicini, 2000). It is necessary to establish a software development process, a modeling language and an implementation mechanism suitable to work with the agent abstraction as the basic building block of a system. It is easy to find works related to these aspects of agent-based system development (Nwana et al, 1999) as the FIPA reference model (FIPA, 2002), the AUML (Odell, Parunak & Bauer, 2002) and ANote (Noya, 2002) modeling languages and the Gaia (Wooldridge, Jennings & Kinny, 2000) development process.

This paper outlines an agent-based abstraction layer for Semantic Web application development. The SemantiCore is structured as a framework to hide platform specific bindings and to provide the basic services primitives and the main agent internal definition for multi-agent systems developers.

To explain what services are abstracted by SemantiCore, Section 2 presents a brief comparison among different computation distribution platforms. Section 3 starts the SemantiCore presentation. The agent abstraction is shown in the first subsection with an example to illustrate the agent creation process. This section also discusses the semantic domain abstraction, which is responsible for society definition and objects' modeling. The final section concludes the presentation discussing the SemantiCore use and contributions and defining the future work topics.

2 SERVICE DISTRIBUTION AND THE SEMANTIC WEB

Distributed systems are rapidly gaining attention by software vendors as a way to scale system capabilities and to achieve better profits by negotiating service units encapsulated in software components. Although the application logic distribution is highly attractive, the complexity involved in the development of such systems are proportionally high. If we take into account the Semantic Web requirements this complexity grows even higher.

To enable complex distributed application development it is necessary to offer high-level abstractions. The current distribution architectures and standards share services as security management, service registration and transaction control. It is possible to provide abstractions to system developers that use these common services in an independent manner. Although these architectures and standards provide new distributed building blocks, some problems arise from their use. To illustrate these problems lets consider the Web Services standard.

The Web Services standard provides the protocols necessary for the implementation of distributed applications through service composition. It uses XML-based messages for service request and response and the HTTP protocol as the transmission medium. Web Services intend to glue applications on the Web by providing a common message transport mechanism independent of the implementation language used to build the application. Many vendors are offering APIs compatible with Web Services standards to enable application development using different languages, such as .Net C# API and the Java Web Services Development Pack.

The Java Web Services Development Pack is formed by support applications and packages to enable the development of Web Services using the Java language. It is easy to notice by the use of this pack that implementing a Web Service is a very demanding activity.

In fact, although the implementation model itself is simplified if compared with the XML message processing necessary for the application operation, the application configuration and deployment is extremely painful. Thus, part of the complexity involved in the service development is situated in the creation of the necessary XML configuration files and in the proper combination of different configuration applications. Figure 1 illustrates the Java code necessary to implement a service using the JAXRPC API.

It is possible to notice that building a service is quite the same as building a regular Java class. This particular true in terms of server code but the client necessary for service invocation has to perform the stub recovery in order to send messages to the service. By doing this, the developer has to know what is the structure generated by the configuration applications in terms of stubs (an application distribution detail).

To configure this simple example it is necessary to create 3 XML files and to run the ANT application for configuration files execution, WSDL (Chinnici et al, 2003) service description generation, stubs and ties development, war package creation and deployment. The simple model applied to the service development turns the configuration needed for its use into a complex task. Other distribution architectures also suffer from the configuration intensive problem. This is partially because they do not change paradigm and try do adapt no distributed environment and abstractions to distribution.

```
import java.rmi.*;
import java.rmi.server.*;

public interface HelloIF extends Remote {
   public String sayHello(String s) throws RemoteException;
}

public class HelloImpl extends UnicastRemoteObject
implements HelloIF {

   public String message = "Hello ";

   public String sayHello(String s) {
      return message + s;
   }
}
```

Figure 1: Hello service example using JAXRPC.

The existent computation distribution architectures are not adapted to the Semantic Web as well, since they do not cover aspects as ontology processing, inference mechanisms, knowledge sharing, learning, and adaptation. It could be useful to provide abstractions to Semantic Web application developers that could be organized into a layer that runs over a distribution layer. This is the main objective of SemantiCore.

3 THE SEMANTICORE

A Web composed by agents and machine-understandable content was envisioned in (Berners-Less, Hendler & Lassila, 2001) as the Semantic Web. The Semantic Web is the mixture of human understandable contents and machine (agents) annotated contents, with formal semantics describing ontologies for agent to operate on. Software agents would be able to "understand" the Web content and interact with each other and with their users in order to achieve certain goals.

The infrastructure necessary to develop the Semantic Web involves a common language to represent the semantics of a domain. The main initiative to provide such language is currently being standardized as the OWL W3C initiative (Smith, Welty & McGuinness, 2003), which is a XML-based language for ontology representation, i. e., specify concepts and the relation between them.

The SemantiCore is a framework that provides an abstraction layer over service distribution architectures in order to offer high-level artifacts for Semantic Web application development. The SemantiCore framework abstractions are an extension of the work developed in the Web Life architecture project (Ribeiro, 2002) to facilitate Web-based multi-agent systems creation.

The main goal of SemantiCore is to allow the development of agents' internals and multi-agent environments, considering the Semantic Web populated by semantic domains where agents "live". SemantiCore abstractions are built over computation distribution platforms, hiding distributed application development details from the developers. Throughout this section the abstractions will be represented using text boldface.

Another SemantiCore goal is to abstract the underlying software platform and communication protocol, enabling the application developers to send and receive messages using different public available standards like Web Services SOAP (Gudgin et al, 2002) or FIPA ACL (FIPA, 2000). Figure 2 shows the SemantiCore layer-based architecture.

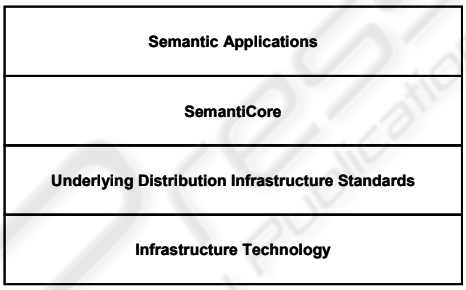

Figure 2: SemantiCore architecture.

The SemantiCore uses as the main abstraction the **semantic agent** construction. A semantic agent is essentially an autonomous execution context populated with logic processing and decision making capabilities which autonomously capture environment events and messages and perform interaction driven computations. These agents are organized into **semantic domains**. By this definition it is clear that each semantic domain can be thought as a multi-agent system. Each domain is connected to each other through the current Internet routing infrastructure. A semantic domain can be spread over multiple traditional Web domains. All the traditional web domain services and data are available to the semantic domain agents through **semantic objects**.

Semantic objects encapsulate attributes, operations and ontologies to provide the objects use contexts. Different objects have different meanings depending on the context they are used. The SemantiCore uses knowledge representation languages such as OWL to annotate ontological information and encapsulate attributes and operations within a semantic boundary. These annotations can be modified by software agents during objects use, creating a sequence of related ontologies that are transmitted with the object. Using this mechanism the object is constantly evolving its use contexts in the semantic web enabling its better use in the future by other agents.

To help developers build applications the SemantiCore framework is divided into two models: the semantic agent model, responsible for the agent internal definition, and the semantic domain model, responsible for defining the domain composition and administrative entities.

The two models provide flexible points (hot-spots) where the developers can plug-in different standards, protocols and technologies for specific task execution. These models are shown in a simplified view without all their characteristics in this paper.

3.1 The Semantic Agent Model

The semantic agent model defines all the elements needed to build a SemantiCore agent. An agent is defined in (Weiss, 1999) as a computer system situated in some environment and capable of autonomous action in order to meet the design objectives. By this definition the notion of autonomy indicates that an agent must have its own thread of control (Bellwood et al, 2002). Another important notion is the environment in which the agent's act. SemantiCore uses the Semantic Web as the agent interaction environment.

The semantic agents are composed of six components. Each component is responsible for specialized tasks and some of them have their own thread of control. They may be thought as a group of collaborative threads. The semantic agent component structure helps the developers to focus in agent parts. This modularization offers benefits in terms of maintenance and code organization.

The Factory design pattern (Gamma et al, 1994) is used to encapsulate the semantic agent drivers and listeners instantiation. This feature enables the semantic agent to be used with different underlying architecture such as FIPAOS or Java Web Services. Each infrastructure must have its own *PeerAgentDriver* to handle the operation the agent needs to initiate. Drivers are responsible for transmitting the agents' commands to the underlying architecture. The agent can listen to the infrastructure events through the *PeerAgentListener*. These abstractions (hotspots) must be implemented be layer configuration managers and have to include message transportation and representation mechanisms to allow the agent understand different message types as ACL or SOAP.

The first necessary capability of a semantic agent is to receive resources from the environment. The Sensorial Component centralizes the elements that enable the reception of semantic objects through the environment. The Sensorial Component has a sensor pool responsible for storing the **sensors** defined by the developers and for verifying if these sensors were activated by a semantic object received from the environment. If one or some sensors were activated, a **history** object is generated and passed to other components for processing. One important issue is the abstraction in sensor instantiation guaranteed by the *SensorFactory* use (hotspot). Developers can define different sensor types to capture different type of objects in the environment. The RDF Sensor is a special type of sensor that captures RDF objects in the environment. RDF (Larissa & Swich, 1999) is a representation language used in the Semantic Web initiative to model domain concepts using a triple (subject, predicate, object).

Every agent has implied objects to access its components, a similar mechanism to the one used in JSP pages to access the generated servlets objects. The **sensorial** object is the implied object which enables the access to the sensorial component interface and is used to install the sensors defined. The developer can override the sensor methods if it is necessary to do operations before the generation and transmission of the activated sensors history to other components.

After receiving the semantic object, the agent sends its content and the activated sensors identifications through the history object to his decision component. The decision component encapsulates the decision making method used by the agent. There are two factory types in the decision component structure: one to provide dynamic language representations in which the rules and facts are coded and the other to abstract the decision engine used by the agent (hotspots). These flexible points must be configured when using the SemantiCore abstraction layer.

The **decision** component receives the history object and processes the semantic objects by translating them into **facts** and **rules** in a given representation language. These facts and rules are sent to the decision engine as input and the decision component waits for the engine's outputs. For an output to be understood it must have an **action** instance. Actions map all the possible commands a semantic agent must understand to work properly. Actions can be applied to internal agent elements or to the semantic domain elements.

Some actions are predefined in the SemantiCore actions library. This library provides the basic

actions over the agent internal elements and over the semantic domain elements. The developers can define their own actions through the extension of the basic framework Action class (hotspot). The class file created has to be stored in the class library repository in the SemantiCore configuration directory. This procedure enables the architecture to dynamically instantiate the user defined class when necessary.

Predefined classes such as the *ExecuteProcessAction* will signal to the **execution** flow component to perform operations. The execution flow component is responsible for the agent participation in work processes which involves the collaboration with other agents. These processes are called **society processes** since they rule the agent society actions to produce a useful work or achieve a common **goal**.

The execution flow component abstracts the workflow engine (WFMC, 1995) used to control **activity** transitions in the workflow process (hotspot). It is important to notice that each agent has its own image of the society process and may have some activities assigned to itself while others actions may be assigned to other agents in the environment (including humans). Some process activities may publish semantic objects in the environment for other agents to consume.

Every semantic object publication in the environment requires an appropriate **effector** in the agent. The effectors are controlled by the agent effector component. This component receives data from other components and encapsulates these data in a semantic object to be transmitted in the environment. As the sensorial mechanism, the effector abstracts the resource representation, allowing developers to use different resource representation languages (hotspot).

The semantic objects transmitted will be encapsulated in the underlying message format. The sender and the receiver of a message will be identified by the underlying infrastructure, i. e., if we are using the FIPA model the messages will be encoded in an ACL envelope which has the sender and the receiver attributes to properly route the message.

Society processes often require semantic object exchange inside their activities. When an object is received during the process execution, its arrival is signaled to the execution flow component and it is stored in the agent **memory**. The component responsible for the agent memory management is the memory component. This component has a persistence manager which encapsulates the persistence technique used.

It is possible to store memory objects using a relational database or an object collection maintained in a file. When the agent is created, the developers have to choose the persistence method for the memory component. It is possible to associate events in the persistence manager to queries in a relational database. The basic store and recover memory mechanism is sufficient to enable the communication among agent's components through the consumed and generated objects.

To enable the evolution of the agent decision making and the knowledge exchange among agents, it is necessary a mechanism capable of controlling and classifying the ontologies used for agent deliberation. The **knowledge** component is responsible for knowledge representation, search and recovery in a semantic agent. This component controls the ontology an agent uses when deciding what to do in the decision component and relates this knowledge with other agent internal elements such as rules, sensors and processes.

The **knowledge object** abstraction encapsulates all the items related to a certain goal achievement. A knowledge object can contain ontologies, rules, sensors, effectors, society processes, facts, actions and activities. It is possible to exchange knowledge objects in the environment. Every semantic agent registers its knowledge in the Environment Manager administrative entity that will be discussed later.

This simple mechanism enables other agents to search for knowledge in the Environment Manager and possibly to get and install (decomposition of object elements and automatic configuration of agent components to operate with the new elements) a certain knowledge object. These search and acquisition features can be started by the *GetKnowledgeAction* class as a result of an inference or automatically activated when an element is referenced in the agent operation and its implementation is not available.

The agent identifies the basic SemantiCore actions because they are all defined in the SemantiCore base ontology. If an action is executed and its implementation class is not found in the class library, the agent automatically searches for its implementation in the target namespace Environment Manager entity.

This section presented the SemantiCore agent elements. A simplified presentation was used to show only the main characteristics of the agent internals. A closer look in the abstraction elements

must be done by the developers who want to create applications using SemantiCore.

3.2 The Domain Model

An agent must operate within an environment. The SemantiCore abstraction layer defines the place where an agent executes as a semantic domain. The domain concept gives the SemantiCore a very smooth transition mechanism from the traditional sense of a Web domain to an enhanced notion of a domain composed by agents and objects. Semantic domains require the Web domains to operate. A semantic domain can be thought as a region in a Web domain in which the agents live.

A domain is a relation of **agent societies** that are organized to achieve a goal through a society process. The semantic domain also abstracts the platform where it is running over (hotspot). This allows the creation of a domain as an extension of a FIPAOS platform. A *Platform* class wraps the command invocations to the platform, translating them to the proper administrative primitives. A *PlatformListener* class is responsible for capturing the platform events and for translating them to SemantiCore internal objects.

Each semantic domain is composed by three main administrative entities: the Resource Manager, the Domain Manager and the Service Manager. The Domain Manager is the agent responsible for registering the other agents in the environment. For a domain to exist, it is necessary a Domain Manager. This agent is also responsible for security features and for the reception of mobile agents from other domains. As any other agent in a domain the Domain Manager also is a SemantiCore agent with sensors, processes and etc configured to provide its basic authentication services.

The Service Manager is the agent responsible for the SemantiCore yellow pages system. The Service Manager links agents to **services** and enables other agents to search for a service. This agent also offers the API for an agent to request other agent services, similar to the Registry server used in the Web Services standard. Every agent that runs on a domain and provides services to other agents must register in the Service Manager.

The Resource Manager is the bridge between semantic and conventional domains. This agent receives messages and translates them to the proper underlying representation to send them in the semantic domain environment. The Resource Manager is also responsible to register the agent knowledge and to enable the search for a knowledge object in the environment. This agent manages the available public semantic objects. If a web page for example is considered a public semantic object, the Resource Manager is responsible for providing the page content to a requester. So, this agent has the characteristics of a Web Server and a SemantiCore agent at the same time.

The administrative entities provide the services necessary for agents to send and receive knowledge objects, adapt their behavior and migrate from one semantic domain to the other. Since these entities are semantic agents, it is possible to extend the administrative services by defining other agents among the semantic domain elements and to configure the main agent element to access these new services.

3.3 A Brief Usage Illustration

To better understand the concepts involved in the agent development, an illustration is necessary. It is based on the system integration idea. The Semantic Web can enable the development of semantic agents that can expand the current enterprise systems functionality while integrating different system components not previously designed to operate together. This integration can be better achieved in companies that have their business processes controlled over an Intranet. It is possible to integrate Intranet-based systems, Internet-based systems and non-distributed systems to leverage the client relationship with the company. For the sake of brevity the example concentrate in the agent society involved in the integration of key business areas.

A business area agent (BAA) encapsulates legacy systems logic offering their services to other BAAs in the company agent society. Consider the BAA agent used to integrate the ordering system which controls the client order processing over the Web to the company production system. The agent is responsible to receive an order and to decompose this order for production processing and controlling.

It is possible to think about different levels of production automation: agents can control each production task through the used of an automated production plant or can only coordinate human-based production activities. The multi-agent integration system can leverage the level of precision and reduce production lost rate, while accelerating the final product delivery to the client by the integration with the Web-based selling system.

To build a semantic agent to control the orders, the developer should extend the *SemanticAgent* class and configure the agent internals through the implicit objects that represent the components. First of all, it is necessary to install agent sensors to capture order objects in the environment. The sensor

must be defined using a *Sensor* or *RDFSensor* class extension. *RDFSensors* are used to capture semantic objects represented in RDF (object representation language hotspot).

After creating and installing the sensor, the developer must configure the decision component creating rules and facts that will be translated to a decision making technique plugged in the framework. In our example, we used the forward chaining method of the Java Theorem Prover (JTP) (Fikes, Jenkins, & Gleb, 2003) inference engine. It is also necessary to use a certain object representation language to represent facts and rules. We used DAML+OIL as the ontology representation language, which was converted to JTP facts and rules. This conversion was internally done by SemantiCore. The selling agent has rules to activate a selling action plan based on the reception of an order object.

The order object received is also stored in the agent memory for further use. The memory storage method used was a relational database. The objects are stored as a table and queried for their attributes. When the order arrive the JTP engine dynamically instantiates an *ExecutePlanAction* class. This action signals to the execution component that a selling plan must be instantiated. The execution component instantiates and controls the plan through its activities. The first activity is to decompose the order into items. For each item a second activity is executed to verify the storage availability. A third activity is executed if there are enough items to be delivered. This activity creates a delivery order semantic object that is transmitted through an effector to the delivery control agent.

It is necessary to configure an effector for the previous action plan capable of sending a delivery order in the environment using an object representation language (in our case the RDF markup language). It is possible to think about a priority selling taking into account VIP clients. To adapt the selling plan to VIP clients the selling agent must acquire a knowledge object in the semantic domain Environment Manager. This is done by transmitting a *RequestKnowledgeObject* object to the Environment Manager. It will return the knowledge object appropriate and the selling agent knowledge component will install the object elements in the corresponding components.

A final remark must be done about the semantic domain configuration. The agents execute over the Web Services standard using SOAP as the message format envelop and HTTP as the transmission protocol. This example presented some extension points and limitations in the SemantiCore current structure. Some of them are discussed in the next section.

4 SEMANTICORE EXTENSIONS AND FUTURE WORK

SemantiCore is a framework that abstracts the common points found in the current distributed technologies to provide high-level agent-based abstractions for those interested in Semantic Web application development. SemantiCore uses a component-based internal agent model to modularize agent functions. These components work together for the agent operation and are synchronized by the agent class lifecycle mechanism.

A SemantiCore agent must execute within an environment called a semantic domain. Each domain has administrative entities responsible for semantic and knowledge objects storage, object and knowledge search capabilities, authentication and security features, and service registration and discovery.

This paper provided an overview of the SemantiCore main elements showing a usage example based on an enterprise business integration semantic application. The system uses agents to integrate company business processes and related systems in the corporate Intranet. Some benefits from the SemantiCore usage in the enterprise business automation can be already seen.

The company has information islands that difficult its operation since the different systems do not communicate with each other. The customer order and the production planning and controlling are done in different moments and databases. It was a human task to plan the production based on a customer order. This task, although not completely straightforward, took into account a few parameters that could be coded in the agent inference system. The immediate translation from the customer order to the production plan enables a faster production and consequently order fulfillment. Other advantage of the use of a multi-agent system structure is the plug and play capability in terms of production resources and the possibility to extend the business areas, since another copy of a business area agent can join the society and may parallelize overused agents.

SemantiCore certainly requires extensions to become a fast middleware for agent construction. It is necessary to reduce the representation language translations among components by centralizing this feature in one translation component. This must enable a faster message exchange and decision making mechanism.

Another important improvement must be done in the SemantiCore agent authentication and security features, providing cryptography-based services, especially for mobile agents' security (Tschudin,

1998). It is also possible to extend the knowledge exchange mechanism to enable behavioral knowledge as the relation among agents. An agent can avoid another agent or can be very collaborative to other agents depending on the behavioral knowledge. These relationships could be defined in terms of SemantiCore operations.

Finally, SemantiCore can evolve to a virtual machine that could be programmed with an agent-based scripting language. The link between the SemantiCore and the underlying platform can continue to be dynamic but the system definition may be done using a specific language instead of Java-based packages. SemantiCore can play an important role for turning the Semantic Web into reality by providing high level abstractions to work with all the Semantic Web related low-level technology.

REFERENCES

Bellwood, T. et al. (2002). UDDI version3.0 . Retrieved October 25, 2003, from http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

Berners-Less, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 1(29), 35-43.

Champion, M., Ferris, C., Newcomer, E., and Orchard, D. (2003) Web Services Architecture, working draft. Retrieved October 25, 2003, from http://www.w3.org/TR/2003/WD-ws-arch-20030514/.

Chinnici, R., Gudgin, M., Moreau, J., and Weerawarana, S. (2003). WSDL Version 1.2, (2003). Retrieved October 25, 2003, from http://www.w3.org/TR/2003/WD-wsdl12-20030611/.

Ferber, J. (1999) Multi-Agent Systems – An Introduction to Distributed Artificial Intellingence. Addison-Wesley. 509p.

Fikes, R., Jenkins, J., and Gleb, F. (2003). JTP: A System Architecture and Component Library for Hybrid Reasoning. [Electronic Version]. *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*. Orlando, Florida, USA.

Foundation for Intelligent Physical Agents (FIPA). (2002). FIPA Agent Management Specification. Retrieved October 25, 2003, from http://www.fipa.org/specs/fipa00023/SC00023J.html.

Foundation for Intelligent Physical Agents (FIPA). (2000). FIPA ACL Message Structure Specification. Retrieved October 25, 2003, from http://www.fipa.org/specs/fipa00061.

Gamma, E. et al. (1994). Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley. 395p.

Gudgin, M., Hadley, M., Mandelsohn, N., Moreau, J., and Nielsen, H. (2002). SOAP Version 1.2, Retrieved October 25, 2003, from http://www.w3c.org/2000/xp/Group/2/06/LC/soap12-part1.html.

Kurniawan, B. (2002). Java for the web with servlets, JSP, and EJB. New Riders. 953p.

Larissa, O and Swich, R. R. (1999). Resource Description Framework (RDF) Model and Syntax Specification. Retrieved October 25, 2003, from http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

Noya, R. C. A modeling language for agent based systems. PhD. Thesis, Computer Science Department, PUC-Rio, 2002.

Nwana, H., Ndumu, D. Lee, L., and Collis, J. (1999). ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. [Electronic Version]. *Applied Artifical Intelligence Journal*, v. 13 (1), p129-186.

Object Management Group (OMG). (2002). Common Object Request Broker Architecture: Core Specification. Retrieved October 25, 2003, from http://www.omg.org/technology/documents/formal/corba_iiop.htm.

Odell, J., Parunak, H., Bauer., B. (2000). Extending UML for Agents. *In Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial* Intelligence, 3-17.

Ribeiro, M. B. (2002). Web Life – A multi-agent systems implementation architecture for the Web. PhD. Thesis, Computer Science Department, PUC-Rio.

Smith, M. K., Welty, C., and McGuinness, D. (2003). OWL Web Ontology Language, working draft. Retrieved October 25, 2003, from http://www.w3.org/TR/owl-guide/.

Tschudin, C. F. (1998). Mobile Agent Security. In: Klusch, M. (Ed.). Intellingent Information Agents. Springer-Verlag, 431-445.

Weiss, G. (1999). Multiagent systems: a modern approach to distributed artificial intelligence. The MIT Press. 619p.

Wooldridge, M.; Jennings, N. R.; Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 285-312.

Workflow Management Coalition (WFMC). (1995). The Workflow Reference Model. Retrieved October 25, 2003, from http://www.wfmc.org/standards/docs.htm.

Zambonelli, F., Jennings, N.R., Omicini, A., and Wooldridge, M. (2000). Agent-oriented software engineering for internet applications. In: Omicini, A. et al. (eds.). Coordination of internet agents: models, technologies, and applications. Springer-Verlag. 326-346.