KOJAC: Implementing KQML with Jini to Support Agent-Based Communication in Emarkets

M. Brian Blake

Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030
mblake@gmu.edu

Abstract

The Java programming language and technologies have been used extensively in the construction of web-based components and applications specifically in the context of electronic markets (emarkets) development. Java-based components can fulfill a variety of atomic roles in an emarket scenario from graphical user interfaces to complex database functions. However, in distributed settings, a great deal of monolithic code must be wrapped into the core functionality offered by Java components to allow for communication or message passing. Java introduces the Jini specification and associated functions to handle this distributed collaboration. Unfortunately, even Jini adds unnecessary overhead to the core functionality of these atomic components. Consequently, agents can be deployed in conjunction with these Java technologies to assist in communication as well as other nonfunctional concerns. The Knowledge Query and Manipulation Language (KQML) is a protocol that has been used in implementing agent communication. This paper suggests the use of agents to broker communication between atomic components. KOJAC (KQML over Jini for Agent Communication) is an approach to agent-based communication implemented with Jini services, specifically JavaSpace. KOJAC defines methods using Jini services that conform to the KQML protocol and incorporate an object-oriented ontology.

Introduction

Emarkets are becoming increasingly popular currently with even higher expectations in the future. One particular domain underlying emarkets is the implementation of workflow. A workflow can be defined as the realization of some business process. Some Internet implementations of workflow are the on-line ordering process, the on-line stock purchasing process, etc. In each step of these processes, a component or actor plays a role in fulfilling the overall goal. In the case of the (Figure 1.1) on-line stock purchasing process some characteristic roles may be a customer interface, broker, or trader roles. Each of these role players can be realized by a Java-based component, while some component-based service or operation may fulfill their functionality or task. Finally, a centralized controller component or scheduler can be deployed to invoke each operation in accordance with a workflow policy.

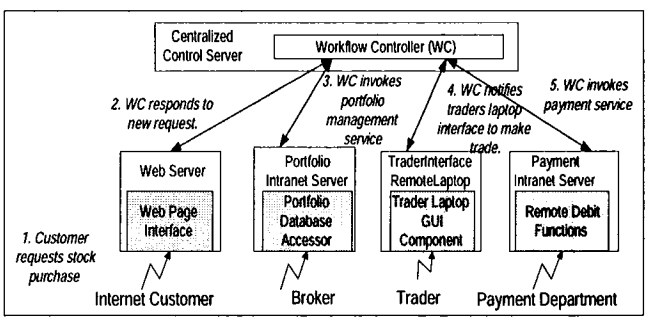

Figure 1.1 On-line Stock Purchasing Domain

Workflow Automation through Agent-Base Reflective Processes, WARP (Blake, 2000b), is an approach that uses an agent-base middleware layer to coordinate internet-based workflow. This work has successfully used agents to mediate communication among distributed services. A high-level architecture in context of the on-line stock-purchasing domain is shown in Figure 1.2

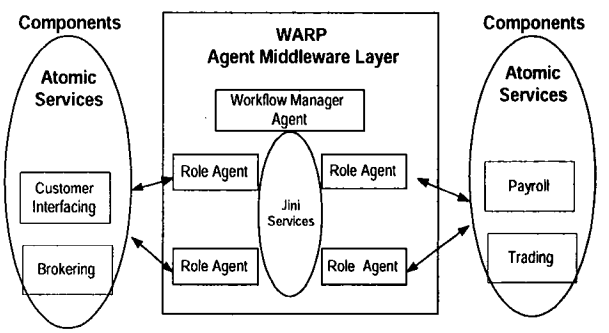

Figure 1.2 WARP Architecture for the On-line Stock Purchasing Domain

This paper proceeds in the next section with an overview of all the pertinent technologies. Next, the workflow-based ontology is defined. Later, the agent-based communication and tools in KOJAC is detailed in context of specific KQML semantics. Finally, this work is summarized in the in context of related work.

Overview of Technologies

In order to set the context for further discussion, this section gives background of each of the underlying technologies (i.e. workflow, Jini services, and KQML).

Workflow Terminology

The workflow language here follows workflow terminology used presently by researchers. The terminology closely follows the concepts in (Lei and Singh, 1997). In order to set the nomenclature for further discussion, the following set of definitions are adhered to throughout this paper.

- A *task* is the atomic work item that is a part of a process.
- A task can be implemented with a *service.*
- An *actor* or resource is a person or machine that performs a task by fulfilling a service.
- A *role* abstracts a set of tasks into a logical grouping of activities.
- A *process* is a customer-defined business process represented as a list of tasks.
- A *workflow* (instance) is a process that is bound to particular resources that fulfill the process.

Jini Services: JavaSpaces

Jini technology is a suite of services developed by Sun Microsystems that provide a simple substrate for distributed computing (Edward, 1999). Jini supports most common principles surrounding distributed coordination (i.e. remote objects, leasing, transactions, and distributed events). It is not in the scope of this paper to give an in-depth description of Jini but to describe those services that are used for agent communication, specifically JavaSpaces (Freeman, Hupfer, and Arnold, 1999). JavaSpace technology is based on "Tuple Spaces" (Gelernter, 1985). Tuple spaces, first introduced in the context of the "Linda" project in 1982, allows distributed software processes to communicate autonomously. The tuple space emulates a data storage server. The server receives entries from independent components and stores them for retrieval. Exterior components can be notified when an entry of a certain pattern or tuple is entered. Components can also read and take matching entries based on a tuple-based pattern they submit. Though JavaSpace technology was motivated by tuple space, it is slightly different. JavaSpaces are "object" storing service. It supports read, write, take, and notify on actual software objects. Sample JavaSpace interactions are illustrated in Figure 2.1.

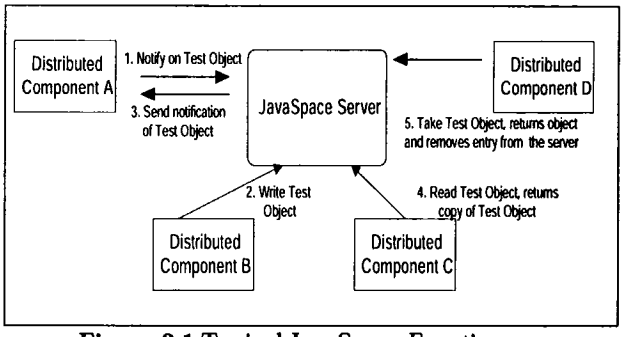

Figure 2.1 Typical JavaSpace Functions

KQML

The motivation for KQML (Labrou and Finin, 1994), (Labrou, Finin, and Peng, 1999) was to formalize a method by which agents can communicate effectively and efficiently. The message format supplies the agent with knowledge of which agent it is communicating to, a protocol for establishing dialogue, the language by which agents are communicating, terms by which other agents will interpret expressions, and exception handling. It is not my intend to cover the KQML specification in entirety but to introduce the portions of the protocol that may assist later interpretations

KQML is separated into 3 layers, content, message and communication layers. The content layer allows agents to communicate which language is going to be used in a particular message. The message layer contains the message to be communicated in the form of content messages and declaration messages. The final layer is the communication layer, which exchanges packages to specify communication attributes. The message layer is of importance to us. The message layer, more specifically in content messages, is what KOJAC emulates

As all layers, the message layer format is in Common Lisp keyword argument format. Some possible keyword arguments are TYPE, QUALIFIERS, CONTENT-LANGUAGE, or CONTENT. The following depiction illustrates an example message.

(MSG
:TYPE <Type of message (e.g. query, assert>
:QUALIFIERS <list of qualifiers for message>
:CONTENT-LANGUAGE <name of language used>
:CONTENT-TOPIC <topic of knowledge>
:CONTENT <Actual message in content language>)

The idea of message types is important to the functionality of this protocol. A specific message may have the functionality of asking a question or responding with an answer. Performatives are specialized KQML message types. The specification of a performative can increase system-wide transactions and functionality. The following example is a sample performative where an agent *joe* queries a stock server agent about the price of a share of

IBM stock.

(ask-one:
:sender joe
:content (PRICE IBM ? price)
:receiver stock server
:reply-with ibm-stock
:language LPROLOG
:ontology NYSE-TICKS)

In later sections, I will show how KOJAC can be used to implement a subset of the common reserved performatives (Labrou and Finin, 1994) as in Table 2.1.

Category	*Name*
Basic query	ask-one, ask-all
Generic Informational	tell-one, tell-all
Capability-definition	advertise, subscribe, monitor
Networking	register

Table 2.1 A Subset of Reserved Performatives

Workflow-based Ontology

KOJAC uses an object-oriented ontology as a shared knowledge base among agents. This solution is practical in the context of object-oriented domain analysis (Gomaa and Kerschberg, 1991) since agents reason about a particular domain when they communicate. Emarket designers can use traditional object-oriented analysis and design techniques to construct a domain model using object-oriented structural diagrams (Booch, 1999). This domain model later translates into a physical set of classes. Objects from these domain classes are later specialized to particular types of JavaSpace entry objects. This is discussed in greater detail in the operational semantics of KOJAC.

The first implementation of KOJAC is for the WARP agents. WARP agents communicate based on a domain that considers workflow policy, roles, services and data flow. This business process-based ontology is reusable

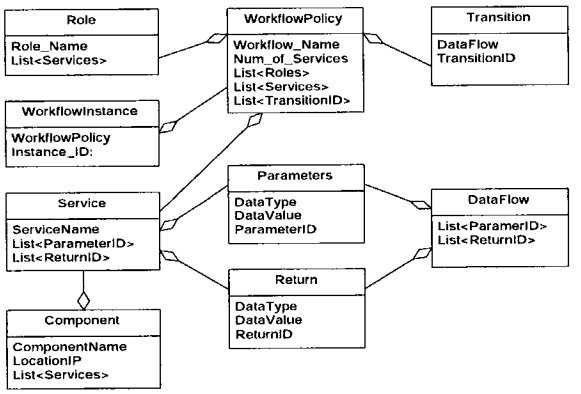

Figure 3.1 Workflow-based Object-Oriented Ontology

across all Emarket domains that implement a workflow of distributed components. The structural view of the workflow-based ontology is illustrated in Figure 3.1. The workflow policy is the heart of this ontology. Agents that coordinate component-based services first need to know the workflow policy. Each step in the workflow policy correlates to a role and the completion of a specific service. Each service has one or more parameters or return values. The workflow policy further defines the subset of parameter and returns that are populated between each individual step as a dataflow. The reason for defining data flow is because one service may return more information than the subsequent service requires. Also, multiple concurrent services may proceed a single service. In this case, a combination of returns from multiple services would precede the subsequent service.

Setting the Foundation for KOJAC

In order to understand the operational semantics of KOJAC, we use the generic workflow-based ontology defined in the previous section. KOJAC can be implemented for any ontology following an object-oriented analysis of the domain and of the system. The corresponding object-oriented design and software would be implemented both in the actual system software and in the agent-based communication software. This section describes this process in the terms of the WARP environment. Therefore, we first give an operational overview of the WARP environment. Then, we describe how KOJAC can be implemented into the system-level object-oriented design lifecycle. Finally, we show how the ontology can be implemented using a Java package of domain model-based classes.

An Overview of the WARP Environment

To consider the operational environment, again let's use the on-line stock-purchasing domain (Figure 1.1.). A configured WARP system would have a Role Manager Agent (RMA) for each of the roles. RMAs act as middle agents (Decker and Sycara, 1997) for components. The RMAs obtain system aspects of the component through introspection and are able to invoke component functions through the process of reflection. The three roles are the Customer Interface Role, the Broker Role, and Trading Role. There is a RMA for each role. There is one Workflow -Manager Agent (WMA) that helps in the coordination of the entire workflow. Each RMA would subscribe for service completion events prior to their affiliated services. For example, the Portfolio Management Role would monitor for the completion event of a getTradeRequest service. Suppose a customer invokes the getTradeRequest service. The Customer Interface RMA would receive a completion event from the component (actor) and would broadcast the pertinent data for this service completion. The RMA for the Portfolio

Management Role would be notified of this completion. First it would check to see if this service is pertinent to any of its workflow policy responsibilities. If so, the RMA for the Portfolio Management Role would wait for the ready event to be written to the server by the WMA. The WMA would have also been monitoring and notified of the getTradeRequest service completion. The WMA would post any amendments to the workflow based on nonfunctional concerns at the process level. Subsequently, the WMA would publish a ready event to the pertinent RMA. Through reflection, the RMA would invoke the proper service (searchPortfolio service) for this step (reflection) in the workflow policy. Subsequently the output data, and the service completion would be broadcast. This process sequence is shown in Figure 4.1 for the stock purchase process.

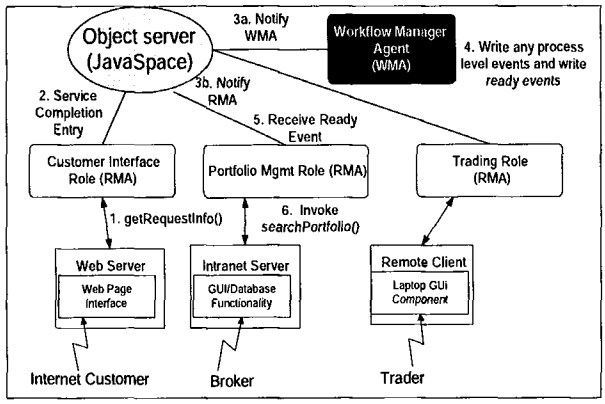

Figure 4.1 WARP Environment for On-line Stock Purchase Domain

Domain Model as the Object-Oriented Ontology

In the previous sections, we discuss the development of a "generic" domain model (Gomaa and Kerschberg, 1991) that can act as an agent-based ontology. Traditional object-oriented analysis techniques analyze the system domain prior to software analysis and design. This analysis clarifies the requirements of the system. In the development of KOJAC, we have noted that this same domain model has all the aspects necessary for communication among agents.

In the business process/workflow ontology, the RMAs communicate among themselves and the WMAs about the status of the process and of the status of services. For example, the Customer Interface RMA would tell other RMAs and the WMA if a particular service on the Customer Interface component had been completed, failed, etc. The WMA may notify multiple RMAs about the start of a process. The WMA may query RMAs to find out what services are available. Moreover, the RMAs can reply with the number of services and specific returns and parameters. The information required for the aforementioned communication is a subset of the information provided in the domain model-based ontology.

Integrating the Ontology with KOJAC

A common second step in object-oriented analysis is translating the domain model into a system analysis model. In this translation, implementation classes are added to the model such as servers, queues, stacks etc. Also, some domain classes are translated into "proxy" classes (i.e. software classes that represent domain entities). Furthermore, some domain classes are directly transferred to the analysis model. The analysis model is the basis for the software design and development.

KOJAC specifically isolates the original domain and proxy class implementations. The agents use the software implementations of these classes for communication. In order to facilitate this process, the software designer needs to specialize these classes to a specific set of abstract classes that adds additional communication based information.

JavaSpace communication relies heavily on the instantiation and use of objects that either implement Entry interfaces or subclass the AbstractEntry class. These objects can be written, taken, read or notified in the JavaSpace server. Jini further specializes these Entry classes. These specializations are Address, Comment, Location, Name, ServiceInfo, ServiceType, and Status. The structural view of the Entry classes is shown in Figure 4.2.

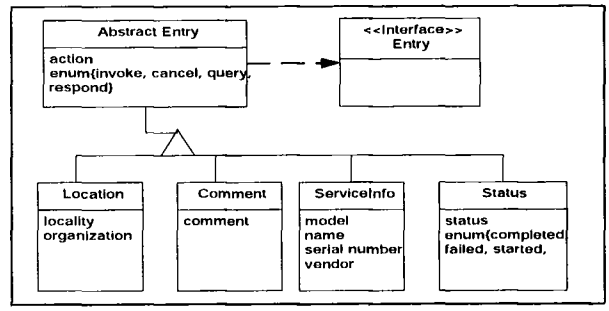

Figure 4.2 Entry: Class Diagram

In order to incorporate the domain and proxy classes with KOJAC, the designer must specialize those classes with the pertinent Entry class. In some cases, we added another layer of specialization to include some functionality that was not included in Jini's set of specializations. In the case of the WARP environment, we added the attribute, "action", to the AbstractEntry class. Finally, this set of classes with the new specializations are compiled into a Java package. This package acts as the shared ontology for the agents. In Figure 4.3, the domain classes from Figure 3.1 are stereotyped for their particular type of Entry. The Service, Parameter, Return, DataFlow, and Transition are all Status Entry classes that get passed among the RMAs and WMAs. The Component class is a Location Entry class because it reveals the location of the components. Roles, WorkflowPolicy, and WorkflowInstance classes are ServiceInfo Entry classes. Interpretations of the type of

Entry Classes will vary from domain to domain. Sub-classing the domain classes is important for object matching.

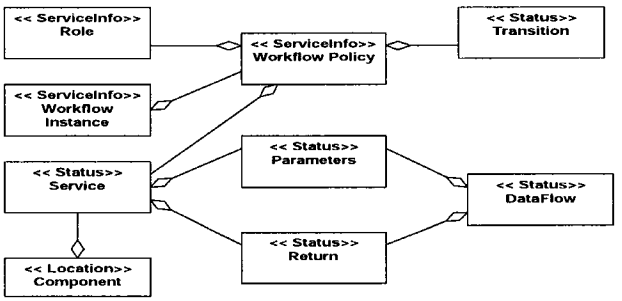

Figure 4.3 Entry Class Specializations

KOJAC: OPERATIONAL SEMANTICS

This section illustrates the interaction protocols of KOJAC based on a subset of the reserved performatives from Table 2.1 using the WARP environment as an example. It is not the intent to detail all possible interactions but more to show how typical interactions would occur.

Register

An agent can register by both connecting to the JavaSpace server and setting notify commands for all entries that it is interested in. For example, a Broker RMA would first connect to the JavaSpace, then it would set notifications for Status entries (Service Class) on services that it can perform. Also, the Broker RMA would set notifications for ServiceInfo entries (Workflow Instance Class) that include services that it encapsulates. The Java-based syntax to register is as stated below.

```
// Get reference to Javaspace Server
JavaSpace SpaceWARP = (JavaSpace)rh.proxy();

//Instantiate Status Entry
this_Service = new Service();
Service.ServiceName = "searchPortfolioInfo";

// Other fields are set to null (WILDCARDS)

// Instantiate ServiceInfo
this_WFInstance = new WorkflowInstance();
this_WFInstance.Service = "searchPortfolioInfo";

// Notify on Service
EventRegistration thisReg =
  SpaceWARP.notify(thisService, null, null,Lease.ANY, null);

// Notify on WFInstance
EventRegistration thisReg =
  SpaceWARP.notify(thisWFInstance, null, null,Lease.ANY,null);
```

Subscribe

An agent can subscribe by setting notifications for pertinent entries. For example, the Trader RMA would subscribe for a Status entry on the searchPortfolio service specifically for entries denoting an invocation action was delivered. This would allow the Trader RMA to prepare for the completion Status entry once the searchPortfolio service is completed. The syntax for this notification is as follows.

```
// Instantiate Status Entry
this_Service = new Service();
Service.action = invoke;

// Set other fields to NULL (WILDCARDS)

// Notify on Service
EventRegistration thisReg =
  SpaceWARP.notify(thisService, null, null,Lease.ANY, null);
```

Tell-all

An agent can tell-all or broadcast a message with a simple write entry to the JavaSpace. Since all agents register their own interests, they will get notified after the completion of the write command. An example in the WARP environment is when a WMA wants to notify all RMAs about a new workflow instance. The WMA would write a ServiceInfo entry of the WorkflowInstance class into the space. This WorkflowInstance class will be fully populated with the entire workflow policy information. The RMAs that are included in the list of services in the policy would be notified. The syntax is as follows.

```
// Instantiate ServiceInfo
this_WFInstance = new WorkflowInstance();

// Populate of all of the pertinent workflow policy information
this_WFInstance.WorkflowPolicy = all_info;

// Write this entry into the Space
SpaceWARP.write(thisWFInstance, null, timeToLive);
```

Ask-all

Implementing the Ask-all performative is a two step process. The agent would first insert a notification for a template of the response that was expected. Secondly, the agent would write an entry where the action is denoted as a query. Using the WARP environment, the WMA might request the location of service. The WMA would first set notify on Component class entries that have a specific service designated. Secondly, the WMA would write a location entry of the Component class that specifies pertinent services and populates the action field as a query.

This process is implemented in the following syntax.

```
// Set notifications for response
this_Comp = new Component();
this_Comp.Service.ServiceName = "searchPortfolioInfo";
this_Comp.action = respond;

// Other fields are set to null (WILDCARDS)

// Notify for response
EventRegistration thisReg =
  SpaceWARP.notify(this_Comp, null, null,Lease.ANY, null);

// Change field to action field to query
this_Comp.action = query;

// Write this entry into the Space
SpaceWARP.write(this_Comp, null, timeToLive);
```

KOJAC Tools

To implement KOJAC, there is a set of object-oriented tools that can be integrated with the Java-based agents to assist in using the JavaSpace and Entry classes. This toolkit can be incorporated into the agent communication functionality or it can be called remotely through Java Remote Method Invocation (RMI). The architecture for the KOJAC tools is detailed in Figure 5.1.

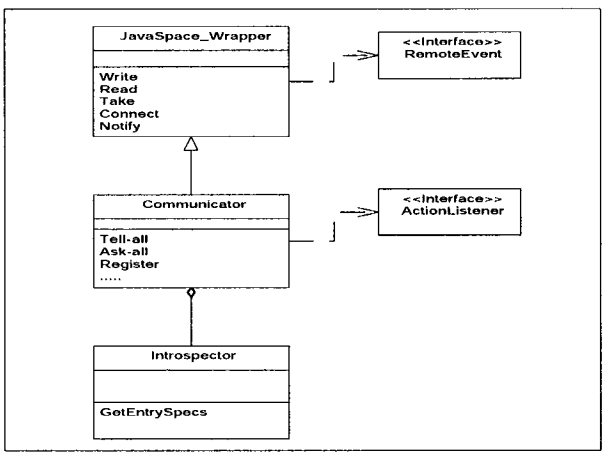

Figure 5.1 High-level KOJAC Components

The KOJAC architecture consists of a Communicator class that inherits functionality from a JavaSpace_wrapper. The JavaSpace_wrapper class implements all of the native JavaSpace commands. The Introspector class looks into the ontology-based package to construct entries used by the Communicator class. The Communicator class also brokers events between the JavaSpace server and the agents.

The flow of operation in the tools is illustrated in Figure 5.2. When a component completes a service, it fires a completion event. The completion event is captured by the RMA. The RMA classifies the event as a completion. The RMA invokes the Tell-all method. Within the Tell-all method the Introspector is instantiated. This Introspector searches the ontology-based package for an entry class that has the same name as the completed service. The introspected class is returned and the action field is populated as a completion. Finally, the inherited write function (from JavaSpace_wrapper parent class) is called with introspected class as a parameter.

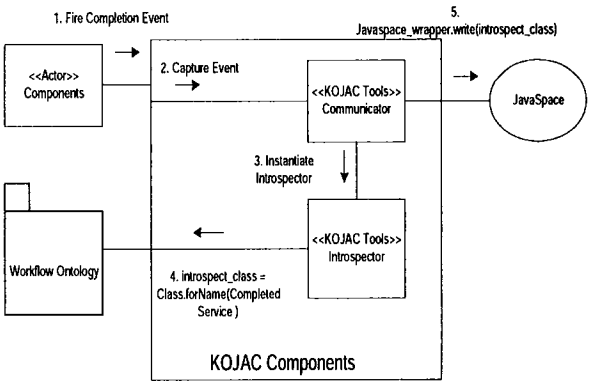

Figure 5.2 KOJAC Tools Operational Flow

Summary

This paper suggests an approach to agent communication that implements KQML semantics using Jini services. Two main focuses in specifying an implementation for agent communication languages are developing a standard suite of APIs that support message transfer and an infrastructure of services that support basic facilitation services (Labrou, Finin, and Peng, 1999). The problem with this currently is that there are many different implementations that tend to deviate from the semantics. KOJAC standardizes an implementation by integrating a standard ACL into a known set of tools and services. By using the primitive structures and functions, other agent-based developers using Java-based technologies can incorporate the same semantics. By using Jini services, agent communication inherits common distributed programming features by default. This use also enforces the standardization of the agent communication semantics.

References

Blake, B. and Bose, P. 2000a. An Agent-based Approach to Alleviating Packaging Mismatch, *Proceedings of the 4th International Conference on Autonomous Agents (AGENTS2000)*, Barcelona, Spain

Blake, M.B. 2000b. *WARP:* An Agent-Based Process and Architecture for Workflow-Oriented Distributed Component Configuration. *Proceedings of the 2000*

International Conference of Artificial Intelligence, Las Vegas, NV, June 2000

Booch, G., Rumbaugh, J., Jacobsen, I., 1999 *The Unified Modeling Language User Guide.* Reading MA, Addison Wesley

Decker, K., Sycara, K., and Williamson, M. 1997. Middle Agents for the Internet, *In the Proceedings of the 15th International Joint Conference on Artificial Intelligence,* Nagoya, Japan.

Edwards, K. 1999. *Core Jini.* Upper Saddle River, N.J.: Prentice Hall

Freeman, E., Hupfer, S., and Arnold, K. 1999. JavaSpaces Principles, Patterns, and Practice, Reading, MA.:Addison Wesley

Gelernter, D. 1985 Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 1, pp. 80-112.

Gomaa H and Kerschberg, L. 1991. An Evolutionary Domain Life Cycle Model for Domain Modeling and Target System Generation, *In Proceedings of the Workshop on Domain Modeling for Software Engineering, International Conference on Software Engineering,* Austin, TX

Labrou, Y. and Finin,T. 1994. A semantics approach for KQML – a general purpose communication language for software agents. *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM-94),* Gaithersburg, MD.

Labrou, Y., Finin, T. and Peng, Y. 1999 "The current landscape of Agent Communication Languages", *Intelligent Systems,* 14(2): IEEE Computer Society

Lei, K. and Singh, M. 1997. A Comparison of Workflow Metamodels, *Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling,* Los Angeles, CA