International Journal on Artificial Intelligence Tools
Vol. 12, No. 1 (2003) 81–99
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

AGENT-BASED COMMUNICATION FOR DISTRIBUTED WORKFLOW MANAGEMENT USING JINI TECHNOLOGIES

M. BRIAN BLAKE*

Department of Computer Science
Georgetown University
234 Reiss Science Building, Washington, DC 20057
blakeb@cs.georgetown.edu

Received 18 November 2001
Accepted 14 October 2002

Agent communication has developed widely over the past decade for various types of multiple agent environments. Originally, most of this research surrounded simulation systems and inference systems. Subsequently, agents are expected to adapt to, dynamically create, and understand evolving conversation policies. This concept of agent communication is not completely necessary in some domains. One such domain is that of distributed workflow management with implications into Electronic Commerce. In this domain, agents are "middle-agents" that represent the distributed components that implement each individual workflow step. By representing the component-based services of each step, multiple distributed agents can essentially manage a workflow or supply chain that spans several on-line businesses (B2B). The WARP (**W**orkflow-Automation through **A**gent-Based **R**eflective **P**rocesses) architecture is a multi-agent architecture developed to support distributed workflow management environments where distributed components are used to implement each of the workflow steps. This paper describes an object-oriented workflow ontology for this distributed workflow management domain. There is also a software engineering process for integrating new component-based services into this ontology. Furthermore, the interaction protocol and supporting implementation based on the Knowledge Query and Manipulation Language (KQML) are presented. This agent communication architecture is implemented using Sun MicroSystems' Java and Jini technologies.

Keywords: Object-Oriented ontology; rule-based agent communication, Jini, workflow management.

1. Introduction

Electronic markets are becoming increasingly popular with higher expectations

*For identification purposes, it should be noted that the author also has an affiliation with The MITRE Corporation, 7515 Colshire Drive, Mailstop N420, McLean,VA 22102, bblake@mitre.org

in the future. Many business interactions occurring over the Internet follow either workflow or supply chain models. Moreover, on-line businesses are adopting the use of components to implement their services. Currently, components are being designed and developed with greater modularity. Independent components can fulfill substantial tasks in these on-line environments. On-line transactions occur both across distributed servers within a single company's Intranet as well as across multiple companies via the Internet (sometimes considered business-to-business or B2B). Subsequently, transactions are no longer the interaction of human-controlled business modules, these transactions are defined more by the configuration and coordination of independent components, regardless of where they are housed. When these transactions interact using workflow or supply chain paradigms, there must initially be a method to designate policy information, and secondly an architecture to sequentially invoke the independent components as specified by that policy.

The WARP architecture was conceptualized to operate in this environment specifically where on-line workflow enactment consists of the coordination of distributed components.[1,2] The WARP architecture uses a two-phased approach. In the first phase, the WARP architecture has semi-automated functionality where humans interact with workflow manager agents in the process of designing a workflow schema. In the second phase relevant to this paper, multiple agents collaborate to manage a workflow of on-line distributed components. Essentially, this is an approach that uses an agent-based middleware layer to coordinate internet-based workflow. One example might be an on-line stock purchasing scenario that requires the workflow coordination of an on-line broker, an on-line trader, and an on-line banker. Each of these on-line businesses may have independent components to perform such tasks as collection of customer requests, stock trade, and payment services, respectively. WARP role agents can act as proxies for the components located at the distributed sites of the independent companies. These agents collaborate on the pre-determined workflow schema to manage the interaction among the components. A high-level architecture in context of the on-line stock-purchasing domain is shown in Figure 1.

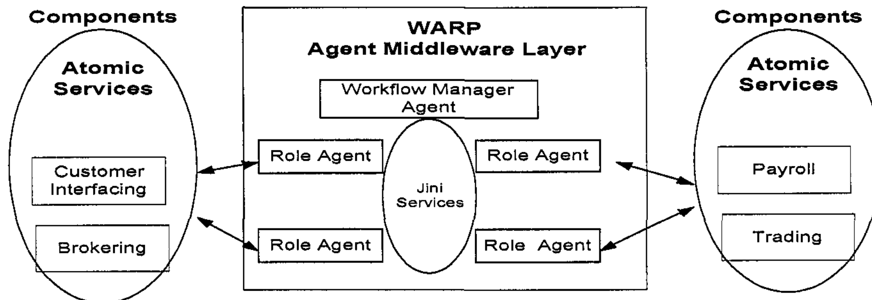

Fig. 1. The Agent Middleware Phase of the WARP Architecture.

The main focus of this paper is the communication among the role agents as a aspect of the workflow coordination of the distributed component-based services. This research uses a *tuple-spaces* approach (as first seen in the Linda project)[3] to communication among a group of agents. In this work, the Sun Microsystems' JavaSpaces implementation is used. This work

also incorporates the reflective capabilities in the Java programming language. The architecture is built on JavaBean component-based services. Hereafter, this approach to agent communication will be referred to as KOJAC (KQML over Jini for Agent Communication). This paper continues in the subsequent sections with a brief background of agent communication. Next, there is a brief overview of the WARP architecture. The following sections discuss the ontology and software engineering approach used to support KOJAC. The final sections discuss the actual agent communication architecture and results.

2. Agent Communication and Electronic Commerce

This section is divided into two sub-sections. The first sub-section gives a brief background on the Knowledge Query and Manipulation Language (KQML) and supporting protocols, which traditionally have been the methodology of choice for the implementation of agent communications. The second sub-section talks about how related research connects KQML with the Extensible Markup Language (XML) and how this approach relates to our work.

2.1. *KQML*

The motivation for KQML was to formalize a method by which agents can communicate effectively and efficiently.[4,5] 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 within the scope of this paper to cover the KQML specification in entirety but to introduce the portions of the protocol that may assist in later interpretations

KQML is separated into three 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 main importance to our work. The message layer, more specifically in content messages, is what is emulated in this work.

As all layers, the message layer format is in the 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, the KOJAC approach that is presented in this paper will be used to implement a subset of the common reserved performatives as in Table 1.[4]

Table 1. A Subset of Reserved Performatives.

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

2.2. *Related agent communication efforts*

Over the last decade, there have been several efforts to create a data format that is acceptable to all software environments. The most notable effort is the work developing the Extensible Markup Language (XML). Currently, XML is the best choice for a language for representing data across multiple platforms. As described in the previous section, KQML has been used to represent data in agent communication. However, KQML uses a Lisp-based text representation that is not widely accepted for business transactions. If agents are to be used in electronic commerce, there is a need for a consolidation of the accepted industry-based representations like XML and the languages that the agents can understand. Current related research trends in agent communication have extended this assertion further. Consequently, the *Agent Communication Markup Language (ACML)* combines the traditional agent communication concepts of KQML with the industry acceptable universal format of XML.[6] Underlying the ACML is the Business Rule Markup Language (BRML). BRML is the B2B-specific content language.[7]

The Foundation for Intelligent Physical Agents (FIPA) has specifications for interaction protocols, communicative acts, and content messages for agent communication.[8] KQML is a subset of both the complexity and completeness of these specifications. Early in the project, the decision was made to use KQML protocols. In the context of this project, KQML protocols are sufficient to support the workflow-based communication. In future efforts, there is a plan to further evaluate the benefits of the evolving FIPA standards.

The main goal of this research is toward implementation-level approaches to agent communication. Realizing this goal will require the connection of agent implementation practices and those currently accepted in industry. This work has set the foundation for upcoming work that unites KOJAC with ACML-type approaches. In our most recent investigations, XML-based schema formats are translated into the object ontology and vice versa. Furthermore, software objects can be constructed based on individual XML documents by which agents can use for communication. Consequently, this work is an initial step toward the consolidation of agent technologies and the general developmental activities for electronic market software systems creation.

3. A Background of the WARP Architecture

The approach to automated compositional configuration is called Workflow Automation through Agent-Based Reflective Processes (WARP). This approach is based on the use of an agent-based middleware architecture here after called the WARP architecture. This WARP architecture consists of software agents that can be configured to control the workflow operation of distributed services. The WARP architecture is divided into two layers. These layers are the application coordination layer and the automated configuration layer.

The application coordination layer is the level in which the workflow instances are instantiated and the actual workflow execution occurs. The application coordination layer consists of two agents, the Role Manager Agent (RMA) and the Workflow Manager Agent (WMA). The RMAs have knowledge of a specific workflow role. The WMA has knowledge of the workflow policy and applicable roles. When a new process is configured, the workflow policy is saved in a centralized database. The RMA plays a role in the workflow execution by fulfilling one or more services as defined by the workflow policy in the centralized database. The RMA registers for workflow step-level events in the event server based on its predefined role. When an initiation event is written into the event server, the RMA is notified. Subsequently based on its localized knowledge of services and its workflow role, the RMA invokes the correct service. The WMA has similar functionality, but instead registers for overall workflow level events (i.e. workflow initiation and nonfunctional concerns). The WMA does not control the workflow execution, but in some cases it adds events to bring about non-functional changes to the execution of the entire workflow.

At the automated configuration layer, agents accept new process specifications and deploy application coordination layer agents with the new corresponding policy. This layer consists of the Site Manager Agents (SMA) and the Global Workflow Manager Agent (GWMA). The GWMA accepts workflow representations from a workflow designer as input. The SMAs discover available services and provides service representations to the GWMAs. The GWMAs accept both of these inputs and writes the workflow policy to the centralized database. The GWMA then configures and deploys WMAs to play certain aspect-oriented roles. At the completion of workflow-level configuration, the SMA configures and deploys RMAs to play each of the roles specified in the workflow database. A general view of the WARP architecture is shown in Figure 2.

The application coordination layer is where the pertinent agent communication occurs. To consider this operational environment, again we motivate the approach using the on-line stock-purchasing domain (Figure 1). A configured WARP system contains a RMA for each of the

roles. RMAs act as middle agents for the components.[9] 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 one WMA that helps in the coordination of the entire workflow. Each RMA subscribes for service completion events that are the pre-conditions to its affiliated services. For example, an agent for the Broker (Portfolio Management) Role would monitor for the completion event of a getRequestInfo service as in Figure 3.

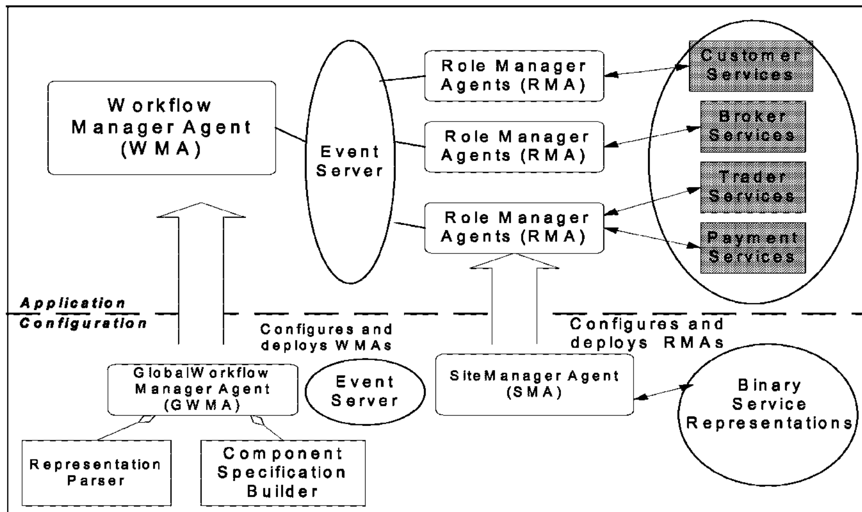

Fig. 2. The WARP Architecture.

In the event a customer invokes the getRequestInfo service, the Customer Interface RMA would receive a completion event from the component (actor) and would broadcast the pertinent data for this service completion. As a result, the RMA for the Broker Role is notified of this completion. First the Broker RMA would check to see if this service is pertinent to any of its workflow policy responsibilities. If so, the Broker RMA would wait for the ready event to be written to the server by the WMA. The WMA also monitors events and is notified of the getRequestInfo service completion. The WMA posts any amendments to the workflow based on nonfunctional concerns at the process level. Subsequently, the WMA publishes a ready event to the pertinent RMA. Through reflection, the RMA would invoke the proper service (searchPortfolio service) for this step in the workflow policy. Subsequently, the output data and the service completion would be broadcasted. This process sequence is shown in Figure 3 for the stock purchase process.

KOJAC is the approach to agent communication needed to manage the sequence of actions described in Figure 3. Agents, in this context, require the capability to understand service completions and must encapsulate the knowledge of resulting actions. Agents also communicate general nonfunctional workflow management concerns like exception-handling, atomicity, and performance.

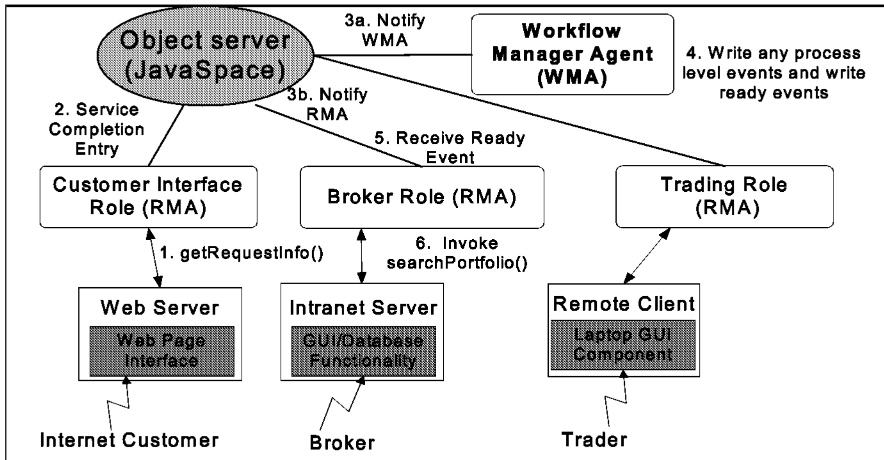

Fig. 3. WARP Operational Environment.

4. The Static Workflow-Oriented Ontology

The agent communication in this domain relies heavily on the concepts of workflow management. In fact, the communication protocols used in the approach are built on a workflow-based ontology. In the following section, the pertinent workflow terminology is defined. Subsequently, there are technical details of the workflow-based object-oriented ontology.

4.1. *Workflow terminology*

The workflow language and terminology used in this work extends general workflow terminology used by other researchers.[10] 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.

4.2. *The workflow-based object-oriented ontology*

The approach to agent communication in this paper defines as object-oriented ontology as the shared knowledge-based among agents. This solution is practical in the context of object-oriented domain analysis, since agents reason about a particular domain when they communicate.[11] We assert that E-market designers can use traditional object-oriented analysis

and design techniques to construct a domain model using object-oriented structural diagrams.[12] This domain model later translates into a physical set of classes. Objects from these domain classes can further be specialized as particular types of Jini/JavaSpaces entry objects. This is discussed in greater detail in the discussion of the operational semantics of KOJAC in Section 6.2.

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 applicable to electronic market domains that implement a workflow of distributed components. The static view of the workflow-based ontology is illustrated in Figure 4. 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.

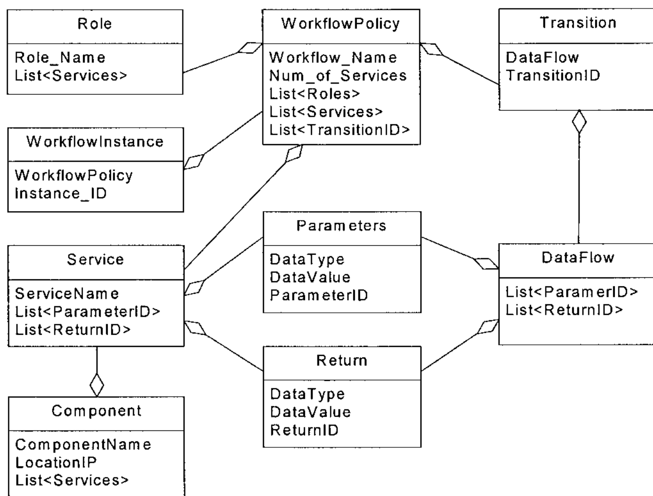

Fig. 4. Workflow-based Object-Oriented Ontology.

Each service has one or more parameters (pre-conditions) or return values (post-conditions). 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.

5. A Software Engineering Development Approach to Agent Communication

A common second step in object-oriented analysis and design is translating the object-oriented domain model into a software design model. In this translation, implementation classes (classes that only pertain to the software implementation domain) 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 software design model. The software design model is the basis for the software design and development.

KOJAC specifically isolates the original domain and proxy classes. 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 into a specific set of abstract classes. The abstract classes have additional communication-based information. The set of abstract classes, which were created in this approach, extends the set of classes defined in the Jini API that support JavaSpaces functionality. JavaSpaces 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 JavaSpaces server. Jini further specializes these Entry classes. These derived classes are *Address, Comment, Location, Name, ServiceInfo, ServiceType,* and *Status.* The structural view of the Entry classes is shown in Figure 5.

As aforementioned, the KOJAC approach extends the original Jini class design by adding a further layer of specialization. For example, an action attribute was added to the native *AbstractEntry* class. This attribute was added so this class would be more consistent with the WARP workflow environment. As this idea of agent communication expands into other domains, other extensions may have to be made to the native Jini classes.

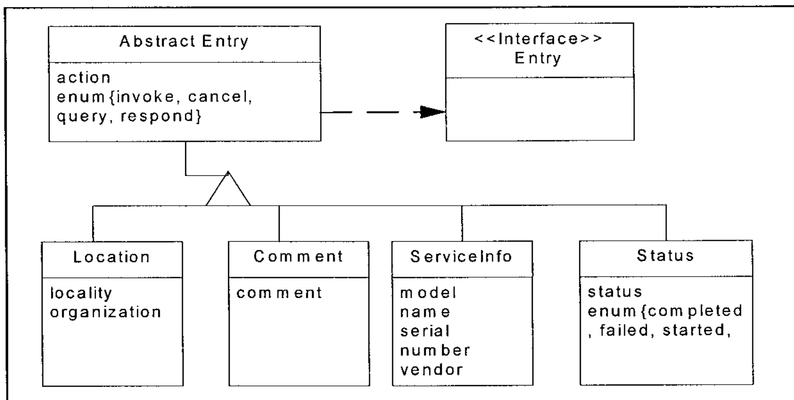

Fig. 5. Hierarchy of Specific Jini Classes used in KOJAC.

In order to incorporate the domain and proxy classes with KOJAC, the designer must specialize those classes with the pertinent Entry class. If the software development process incorporates tools that capture object-oriented models, such as Rational Corporation's Rose application, these specializations can be made with a few keystrokes. As a final step, the resulting agent communication-specific set of classes is compiled into a Java package. This package acts as the shared ontology for the agents. In the run-time environment, agents will reflectively access this ontology using introspection and reflection as defined in the Java development environment. The KOJAC-specific steps as they relate to a typical software development lifecycle are summarized in Figure 6.

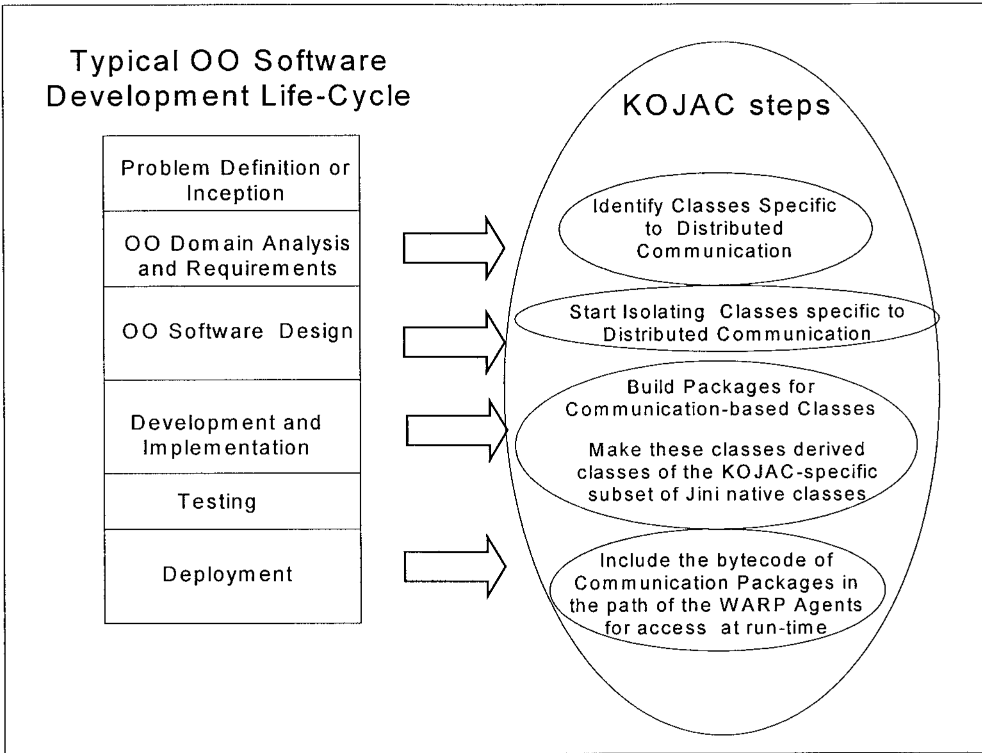

Fig. 6. KOJAC Steps Integrated with the Software Development Life-Cycle.

5.1. *KOJAC in the WARP environment*

In order for KOJAC to work in the WARP environment, the classes in Figure 4 were used as the distributed communication-based classes. These workflow-oriented classes derive functionality from the native Jini classes illustrated in Figure 5. As a result of the WARP approach, domain classes illustrated in Figure 4 are translated into classes derived from Jini foundational classes. In Figure 7, we use stereotype notation (i.e. << >>) to show from which of Jini-based classes that each of the workflow-oriented classes are derived. The *Service*, *Parameter*, *Return*, *DataFlow*, and *Transition* classes are *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 that the RMAs will be invoking. *Roles*, *WorkflowPolicy*, and *WorkflowInstance* classes are *ServiceInfo* Entry classes. Interpretations of the type of Entry Classes will vary from domain to domain. Later discussions of the use JavaSpaces, in the next section, will show how the act of sub-classing the domain-based classes is important for object matching.

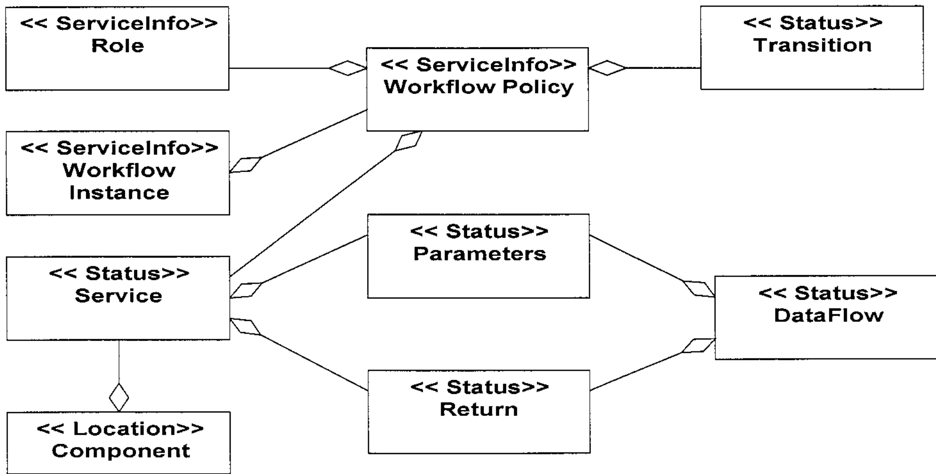

Fig. 7. Entry Class Specializations for the WARP Ontology.

6. KOJAC

The KOJAC approach has a set of semantics and an operational environment that extensively incorporates the operations of Sun Microsystems' JavaSpaces technology.[13,14] This section gives an introduction of JavaSpaces technology and then describes the operational semantics and tools associated with the KOJAC approach.

6.1. *Using Jini services and JavaSpaces technology*

Jini is based on a suite of services developed by Sun Microsystems that provide a simple substrate for distributed computing.[15] 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.[16] As aforementioned, JavaSpaces technology is based on the *Linda* project.[3] This approach allows distributed software processes to communicate autonomously. This approach 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. A component can also read and take matching entries based on a tuple-based pattern that that component submits. Although JavaSpaces technology was motivated by the Linda approach, it is slightly different. JavaSpaces is an "object" storing service. It supports read, write, take, and notify on actual software objects. A few basic interactions are illustrated in Figure 8.

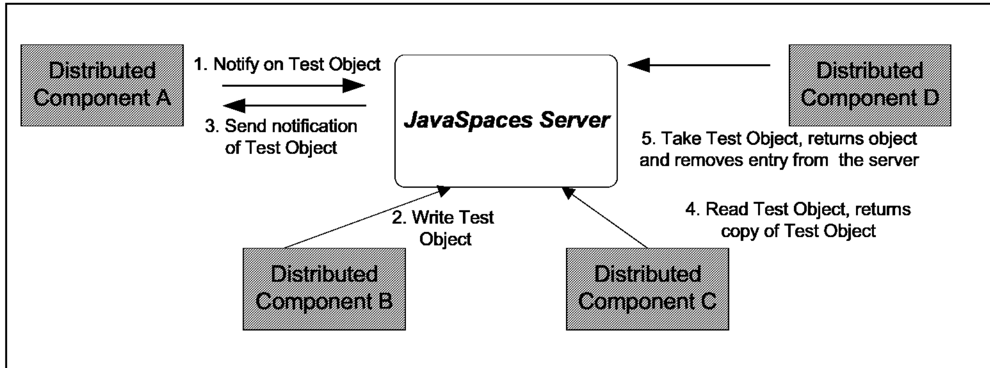

Fig. 8. Typical JavaSpaces Functions.

6.2. *KOJAC: operational semantics*

This section illustrates the interaction protocols of KOJAC based on a subset of the reserved performatives from Table 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.

6.2.1. *Register*

A typical first step of multiple agent coordination is for independent agents to register to the group of all agents. In KOJAC, an agent can register by connecting (Line 1) to the JavaSpaces server and setting notify commands for all entries that it is interested in. For example, a Broker (Portfolio Management) RMA as in Figure 3 would first connect to the JavaSpaces server, then it would set notifications for Status entries (Service Class) on services that it can perform. This will enable notifications to be sent to that agent when there are status messages pertinent to its services. Also, the Broker RMA would set notifications for ServiceInfo entries (Workflow Instance Class) that include services that it encapsulates (Line 8,9,14, and 15). Therefore, when an WMA distributes new workflow instances that contains a service that can be fulfilled by the Broker RMA, then that agent is notified. The Java-based syntax for agent registration is as stated below. This code shows hard-coded service names (Lines 5 and 9) for demonstration only. However, in operational environments, the service information is dynamically imported from a central database.

```
[1] // Get reference to JavaSpaces Server
[2] JavaSpace SpaceWARP = (JavaSpace)rh.proxy();

[3] //Instantiate Status Entry
[4] this_Service = new Service();
[5] Service.ServiceName = “searchPortfolioInfo”;

[6] // Other fields are set to null (WILDCARDS)
[7] // Instantiate ServiceInfo
[8] this_WFInstance = new WorkflowInstance();
[9] this_WFInstance.Service = “searchPortfolioInfo”;
```

```
[10] // Notify on Service
[11] EventRegistration thisReg =
[12] SpaceWARP.notify(thisService, null, null,Lease.ANY, null);

[13] // Notify on WFInstance
[14] EventRegistration thisReg =
[15] SpaceWARP.notify(thisWFInstance, null, null,Lease.ANY,null);
```

6.2.2. *Subscribe*

An agent may desire to receive notification messages from other agents. For example, the Trader RMA, as in Figure 3, subscribes for a *Status* entries of the searchPortfolio service. In this case, the agent is specifically interested in entries denoting that an invocation action was delivered (Line 3 and 4). This would allow the Trader RMA to receive completion Status notifications once the searchPortfolio service is completed. The sample syntax for this process is as follows.

```
[1] // Instantiate Status Entry
[2] this_Service = new Service();
[3] Service.ServiceName = "searchPortfolioInfo";
[4] Service.action = invoke;

[5] // Set other fields to NULL (WILDCARDS)

[6] // Notify on Service
[7] EventRegistration thisReg =
[8] SpaceWARP.notify(thisService, null, null,Lease.ANY, null);
```

6.2.3. *Tell-all*

In KOJAC, an agent can tell-all or broadcast a message by using a standard write entry to the JavaSpaces server (Line 7). 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 is 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.

```
[1] // Instantiate ServiceInfo
[2] this_WFInstance = new WorkflowInstance();

[3] // Populate all of the pertinent workflow policy information
[4] this_WFInstance.WorkflowPolicy = (all_info) ;

[5] // Write this entry into the Space
[6] // timeToLive sets a duration on the length of time the entry remains in the JavaSpace server
[7] SpaceWARP.write(thisWFInstance, null, timeToLive);
```

6.2.4. *Ask-all*

An agent may use an Ask-all performative to request information from the group of all agents. Using KOJAC, the implementation of the Ask-all performative is a two step process. The agent would preemptively insert a notification for a template of the expected response (Line 2,3,4, 7, and 8). Secondly, the agent would write an entry where the aforementioned action is denoted as a query. In the WARP environment, a WMA can request the location of a service from a group of RMAs. The WMA would first set a notification for the Component class entries of the desired service (Line 2,3,4). Secondly, the WMA would write a location entry of the Component class that specifies the desired services. The action, in this entry, is populated as a query (Line 7,8,9,10). This process is implemented in the following syntax.

```
 [1] // Set notifications for response
 [2] this_Comp = new Component();
 [3] this_Comp.Service.ServiceName = "searchPortfolioInfo";
 [4] this_Comp.action = respond;

 [5] // Other fields are set to null (WILDCARDS)
 [6] // Notify for response
 [7] EventRegistration thisReg =
 [8] SpaceWARP.notify(this_Comp, null, null,Lease.ANY, null);

 [9] // Change field to action field to query
[10] this_Comp.action = query;

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

6.3. *KOJAC tools*

KOJAC consists of a set of object-oriented tools that can be integrated with the Java-based agents to assist in using the JavaSpaces and Jini 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 component diagram for the KOJAC tools is detailed in Figure 9.

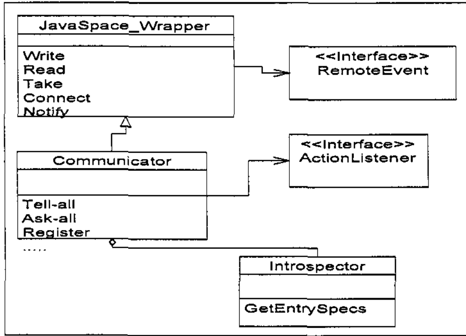

Fig. 9. KOJAC Components.

The KOJAC architecture consists of a Communicator class that inherits functionality from a JavaSpace_Wrapper. The JavaSpace_Wrapper class implements all of the native JavaSpaces 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 JavaSpaces server and the agents.

A basic flow of operations is illustrated in Figure 10. This operational flow is details the process when an agent communicates a service completion event. When a component completes a service, it triggers a completion event. The completion event is captured by the RMA. Since the WARP agents are workflow-based, they contain internal intelligence mapping workflow-based events to agent communication actions. This mapping would be different in different domains and would be incorporated in the agents of that domain. An efficient method of this mapping is being investigated in future work. 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 the JavaSpace_Wrapper parent class) is called with the introspected class as a parameter.

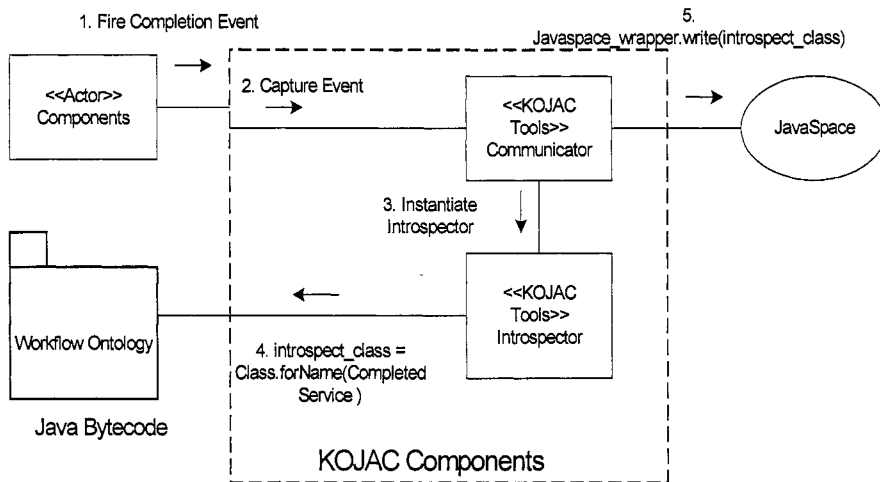

Fig. 10. KOJAC Tools Operational Flow.

7. KOJAC Prototype and Performance Details

A prototype of the WARP architecture was implemented using 3 Dell workstations. One workstation containing the WMA was contained on a Dell Workstation running Windows NT Server 4.0. This workstation also contained both Apache's Tomcat webserver and the Oracle 8i relational database. This workstation also contained Sun Microsystems' JavaSpace server. Two other Dell Workstations, running Windows 98 were connected as peers to the initial workstation. The peer workstations each contained RMAs. This environment was used to

simulate the workflow coordination of distributed components. Each peer-level workstation contained several JavaBean components that acted as the underlying services in this prototypical workflow management scenario. The aforementioned operational environment is illustrated in Figure 11.

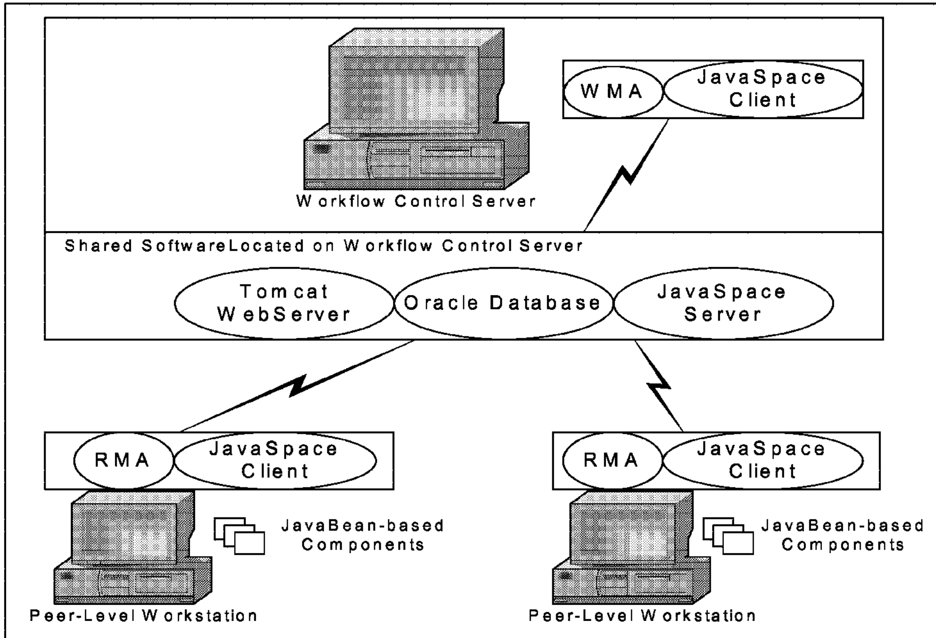

Fig. 11. WARP Prototype.

Early results from the WARP architecture have shown that there is a high degree of overhead when reflectively invoking the component-based services, specifically with large numbers of concurrent workflow instances. This overhead was mostly contributed to the invoking components reflectively over a registry. The WARP architecture dynamically accesses and invokes JavaBean components that are available on Java's RMI registry. Major overhead is associated with the introspection of components that are registered on that registry as opposed to components that are on the local disk. The WARP architecture uses introspection initially to discover the components. These components can later be invoked reflectively. This introspection and reflection over the registry is extremely expensive. However, since the *bytecode* for the communication-based ontology is local to the WARP agents, the communication classes are not registered on the registry and reflection occurs locally. As a result, the overhead of these type interactions, even with a large number of concurrent workflow instances being executed, is relatively small.

Table 2 shows the significant amount of overhead associated when invoking components reflectively with hard-coded workflow policies (i.e. without the WARP dynamic functionality)

over the RMI registry. The overhead was measured against a baseline where a workflow of local components was reflectively invoked locally. The baseline was consistently near 8.2 seconds to complete the workflow instance. However, the same increase of overhead does not occur when introspecting local agent communication-based classes (bytecode). Table 3 shows that the addition of the WARP architecture including all the agent communication has a relatively consistent overhead of 15% even with the increase of workflow instances.

Table 2. Overhead Involved in Reflectively Invoking Components from the RMI Registry.

Workflow Instance (10 steps/ 6 Roles)	**Completion Time/ Overhead (10 Instance)**	**Completion Time/ Overhead (20 Instances)**	**Completion Time/ Overhead (50 Instances)**	**Completion Time/ Overhead (100 Instances)**
Overhead for Reflective Components **without** WARP Architecture vs. one instance (~8.2 sec)	9.56/ 16.6%	14.2/ 73.2%	16.01/ 97.5%	56.7/ 700%

Table 3. Additional Overhead with the Addition of the WARP Architecture including KOJAC.

Workflow Instance (10 steps/ 6 Roles)	**Completion Time/ WARP and KOJAC Overhead (10 Instance)**	**Completion Time/ WARP and KOJAC Overhead (20 Instances)**	**Completion Time/ WARP and KOJAC Overhead (50 Instances)**	**Completion Time/ WARP and KOJAC Overhead (100 Instances)**
Overhead for Reflective Components **with** WARP architecture vs. Table 2	10.87/ 13.7%	16.27/ 14.6%	18.51/ 15.6%	65.32/ 15.2%

The results in Table 3 show that WARP and KOJAC together only add an additional 15% overhead to the workflow execution. However, using RMI registry services for reflectively introspecting components over a registry appears to be impractical. The results in Table 2, though on the surface do not appear to be relevant to KOJAC, are very important. For this approach to agent communication to be accepted in distributed operational environments, it is clear that the bytecode for the agent communication knowledge must be distributed. If this ontology is not distributed then multiple copies of the ontology must be local to each RMA and WMA. The overhead involved in remotely accessing this bytecode on the registry is proven to be extremely high by the aforementioned results. Therefore, this research must be extended to discover more efficient means of both invoking distributed components and introspecting agent communication knowledge. In initial research, we have experienced lower percentages of overhead using the CORBA-based processes of OrbixWeb. Figure 12 illustrates the overhead discussed in Table 2 and 3 using a graph.

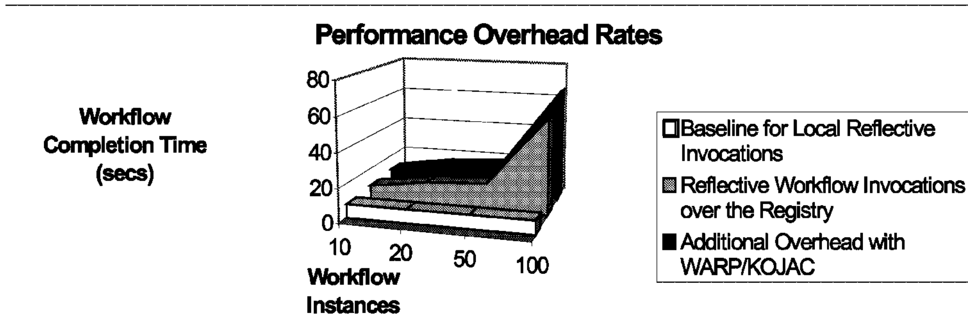

Fig. 12. Overhead rates detailed in Table 2 and Table 3.

8. Discussion and Future Work

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.[5] 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.

This approach integrates well with current software development lifecycles as the Rational Unified Process (RUP).[17] In fact, tools implementing RUP can automatically generate the source code that is used as the agent communication ontology. Using bytecode as a repository for storing agent communication knowledge makes a useful connection between software development processes and agent integration. In addition, this approach fits seamlessly with current distributed event-based development tools like Jini. However, this research has proven that with the distribution of this bytecode across networks and among separate networks, there is a huge amount of performance overhead. This overhead may even suggest that this approach may be impractical when large numbers of components are distributed among multiple networks.

Performance results have opened avenues for future research. Initially, we plan to investigate other architectures that may efficiently support this approach to agent communication. Another area of research is storing the ontology in XML schema or even ACML schema. This research would be promising in making a connection to other relevant agent communication efforts. The best connection would be the translation of the agent ontology from the KOJAC software design steps illustrated in Figure 6 directly to XML/ACML-based semantics. Finally, with the on-going development of the communicative

acts in the FIPA's Agent Communication Language (FIPA-ACL), we plan to incorporate these concepts with the present semantics that are mostly toward the use KQML protocols.

References

[1] M.B. Blake, *Rule-Driven Coordination Agents: A Self-Configurable Architecture*, Proc. 5th Int. Symposium on Autonomous Decentralized Systems (ISADS2001), Dallas, TX, IEEE Computer Society Press (March 2001) 271-278

[2] M.B. Blake, *WARP: Workflow Automation through Agent-Based Reflective Processes*, Proc. 5th Int. Conf. on Autonomous Agents, Montreal, Canada, (May 2001) (software demonstration)

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

[4] Y. Labrou and T. Finin, *A semantics approach for KQML – a general purpose communication language for software agents.* Proc. 3rd Int Conf. on Information and Knowledge Management (CIKM-94), Gaithersburg, MD (December 1994) 447-455

[5] Y. Labrou, T. Finin, and Y. Peng, *The current landscape of Agent Communication Languages*, Intelligent Systems, 14(2): IEEE Computer Society Press (1999) 45-52

[6] B. Grosof and Y. Labrou, *An Approach to using XML and a Rule-based Content Language with an Agent Communication Language*, IJCAI-99 Workshop on Agent Communication Languages, Stockholm, Germany (1999)

[7] B. Grosof, Y. Labrou, and H. Chan, *A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML*, Proc. 1st ACM Conf. on Electronic Commerce (EC-99) Denver, Colorado: ACM Press (1999) 68-77

[8] FIPA Interaction Protocol Specification (2002), http:// www.fipa.org/repository/ips.html

[9] K. Decker, K. Sycara, and M. Williamson, *Middle Agents for the Internet,* Proc. 15th Int. Joint Conf. on Artificial Intelligence, Nagoya, Japan (1997)

[10] K. Lei and M. Singh, *A Comparison of Workflow Metamodels*, Proc. ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling, Los Angeles, CA (November 1997) http://osm7.cs.byu.edu/ER97/workshop4/ls.html

[11] H. Gomaa and L. Kerschberg, *An Evolutionary Domain Life Cycle Model for Domain Modeling and Target System Generation,* Proc. of the Workshop on Domain Modeling for Software Engineering, Int. Conf. on Software Engineering, Austin, TX (May 1991)

[12] G. Booch, J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide.* Reading MA, Addison Wesley (1998)

[13] M.B. Blake, *KOJAC: Implementing KQML with Jini to Support Agent-Based Communications in Emarkets,* Proc. AAAI-2000 Workshop on Knowledge-based Electronic Markets (KBEM2000) (AAAI Press, Technical Report WS-00-04) Austin, TX (August 2000) 1-8

[14] M. B. Blake, *Using Agent Control and Communication in a Distributed Workflow Information System,* Cooperative Information Systems, Lecture Notes in Computer Science, Springer-Verlag, (Proc. of the 10th Int. Conf. on Cooperative Information Systems (CoopIS 2002), Irvine, California, (November 2002) 163-177

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

[16] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*, Reading, MA.:Addison Wesley (1999)

[17] P. Krutchen, *The Rational Unified Process: An Introduction* (2nd Ed.). Prentice Hall (2000)