An Agent-based Infrastructure for Enterprise Integration *

R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff
Laboratory for Advanced Information Technology
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland 21250
cost@acm.org, {finin, jklabrou, xluan1, ypeng, ian}@cs.umbc.edu

James Mayfield
Research and Technology Development Center
Johns Hopkins University Applied Physics Laboratory
Laurel, Maryland 20723
james.mayfield@jhuapl.edu

Akram Boughannam
Advanced Manufacturing Solutions Development
IBM Corporation
Boca Raton, Florida 33431
akram@us.ibm.com

Contact:	R. Scott Cost, cost@acm.org Voice (410) 419-8538, FAX (202) 478-0300
Track:	Emerging Application, Technology and Issue
Domain:	Enterprise Integration Automation
Tools:	Agent Technology, Java
Status:	Operational Research Prototype

Abstract

Jackal is a Java-based tool for communicating with the KQML agent communication language. Some features that make it extremely valuable to agent development are its conversation management facilities, flexible, blackboard style interface and ease of integration. Jackal has been developed in support of an investigation of the use of agents in enterprise-wide integration of planning and execution for manufacturing. This paper describes Jackal at a surface and design level, and demonstrates its use in a multi-agent system that supports intelligent of enterprise planning and execution.

Introduction

Jackal is a Java package that allows applications written in Java to communicate via the KQML (Finin, Labrou, & Mayfield 1997) agent communication language. It is designed to be used as a 'tool' by other applications, in that it does not require that applications be modified or extend some standard shell. Additionally, Jackal is designed so that multiple instances of it, and therefore multiple agents, may be run within the same Java Virtual Machine.

Jackal has been developed as part of a larger effort to develop an agent infrastructure for manufacturing information flow. It has been used to facilitate communication among diverse agents responsible for collecting, processing and distributing information on a manufacturing shop floor.

In next section, we introduce Jackal within the context of some other related agent systems, and follow that with some motivation for higher-level conversation specification. Next, we present Jackal's design in some detail. Finally, we discuss the domain within which Jackal has been developed - enterprise integration automation - and illustrate this with an example.

Jackal and Agent Development

Agents that will interact with one another require some method of communication in order to coordinate their activities and distribute and collect information. To this end, several agent communication languages (e.g., KQML (Finin, Labrou, & Mayfield 1997), FIPA ACL (FIPA 1997), ARCOL (FIPA 1997), ICL (Martin, Cheyer, & Moran 1998), AgenTalk (Kuwabara 1995), KaOS (Bradshaw *et al.* 1998), and AOP (Shoham 1993)), and various software tools for them (e.g., TKQML (Cost *et al.* 1997), OAA (Martin, Cheyer, & Moran 1998), JAT and JATLite (Frost 1998; Petrie 1998)), have been developed. Jackal is a tool for the use of KQML by agents written in the Java programming language. Java is a useful language for writ-

*This work is supported in part by the Advanced Technology Program, administered by the National Institute of Standards and Technology, under agreement number: 70NANB6H2000.

ing agents because it is platform independent, as an interpreted language, and has good language support for multi-threading. Jackal benefits from these properties, and relies exclusively on the Sunsoft JDK 1.2 classes and virtual machine, unmodified. This maximizes the likelihood that Jackal-based agents can run without modification on any platform that supports Java. Not only can Jackal-based agents run on diverse or remote environments; many may coexist within the same Java Virtual Machine. This is exploited by transparent protocol adapters for shared memory message passing.

Adding communication abilities to any Java program requires minimal modification of existing code. This is because Jackal's functionality is accessed through a class instance, which can be shared among agent components. Thus, after creating an instance of Jackal (the J3.Intercom Class) the agent accesses Jackal's functionality through method calls on this instance, which can be shared or passed as a parameter to other classes. This is in contrast to systems that require a program to subclass an agent shell, or otherwise restructure itself. With this Jackal instance, the agent gains more than just the ability to send and receive messages, however. Jackal's design is based in large part on, and implements, the KQML Naming Scheme (KNS), an evolving standard for resolving agent names in a hierarchically structured, dynamic environment. This means that the agent application need only deal with symbolic agent names, and may leave issues such as physical address resolution and alias identification to the Jackal infrastructure.

Two components that work together to provide the greatest benefit to the agent are the conversation management routines and the Distributor, a blackboard for message distribution. The conversation system supports the use of easily interchangeable protocols for interaction, which guide the behavior of the system. The Distributor presents a flexible, active interface for internal message retrieval by agent components. While the Distributor optimizes access to the message flow, it is the conversation system that gives it its real value; the next section will discuss in depth the rational behind the conversation-based approach.

Conversation-Based Protocols

The study of agent communication languages (ACLs) is one of the pillars of current agent research. KQML and the FIPA ACL are the leading candidates as standards for specifying the encoding and transfer of messages among agents. While KQML is good for message-passing among agents, the message-passing level is not actually a very good one to exploit directly in building a system of cooperating agents. After all, when an agent sends a message, it has expectations about how the recipient will respond to the message. Those expectations are not encoded in the message itself; a higher-level structure must be used to encode them. The need for such conversation policies is increasingly recognized by the KQML community, and has been formally recognized in the latest FIPA draft standard (FIPA 1997; Dickenson 1997).

It is common in KQML-based systems to provide a message handler that examines the message performative to determine what action to take in response to the message. Such a method for handling incoming messages is adequate for very simple agents, but breaks down as the range of interactions in which an agent might participate increases. Missing from the traditional message-level processing is a notion of message context.

We claim that the unit of communication between agents should be the conversation. A conversation is a pattern of message exchange that two (or more) agents agree to follow in communicating with one another. In effect, a conversation is a communications protocol, albeit one that may be initiated through negotiation, and may be short-lived relative to the way we are accustomed to thinking about protocols. A conversation lends context to the sending and receipt of messages, facilitating interpretation that is more meaningful. The adoption of conversation-based communication carries with it numerous advantages to the developer, including:

- There is a better fit with intuitive models of how agents will interact than is found in message-based communication.
- There is also a closer match to the way that network research approaches protocols, which allows both theoretical and practical results from that field to be applied to agent systems.
- Conversation structure is separated from the actions to be taken by an agent engaged in the conversation. This allows the same conversation structure to be used by more than one agent, in more than one context.
- The standard advantages of the underlying ACL accrue, including language- and ontology-independence.

To date, little work has been devoted to the problem of conversation specification and implementation for mediated architectures. Strides must be taken in the following directions:

- Potential conversations must be easy to specify.
- Conversation specifications must be easy to reuse.
- Libraries of standard conversations should be developed.
- An ontology of conversations must be developed.

To achieve these goals, we must solve three main problems:

1. Conversation specification: How can conversations best be described so that they are accessible both to people and to machines?
2. Conversation sharing: How can an agent use a conversation specification standard to describe the conversations in which it is willing to engage, and to learn what conversations are supported by other agents?
3. Conversation aggregation: How can sets of conversations be used as agent 'APIs' to describe classes of capabilities that define a particular service?

Conversation specification

A specification of a conversation that could be shared among agents must contain several kinds of information about the conversation and about the agents that will use it. First, the sequence of messages must be specified. Traditionally, deterministic finite-state automata (DFAs) have been used for this purpose; DFAs can express a variety of behaviors while remaining conceptually simple. For more sophisticated interactions, however, it is desirable to use a formalism with more support for concurrency and verification. Currently, we are investigating the use of colored petri nets as an alternative mechanism for more sophisticated conversation specification. Next, the set of roles that agents engaging in a conversation may play must be enumerated. Many conversations will be dialogues, and will specify just two roles; however conversations with more than two roles are equally important, representing the coordination of communication among several agents in pursuit of a single common goal.

DFAs and roles dictate the syntax of a conversation, but say nothing about the conversation's semantics. The ability of an agent to read a description of a conversation, then engage in such a conversation, demands that the description specify the conversation's semantics. To be useful though, such a specification must not rely on a full-blown, highly expressive knowledge representation language. We believe that a simple ontology of common goals and actions, together with a way to relate entries in the ontology to the roles, states, and transitions of the conversation specification, will be adequate for most purposes. This approach sacrifices expressiveness for simplicity and ease of implementation. It is nonetheless perfectly compatible with attempts to relate conversation policy to the semantics of underlying performatives, as proposed for example by (Bradshaw *et al.* 1998).

These capabilities will allow the easy specification of individual conversations. To develop systems of conversations though, developers must have the ability to extend existing conversations through specialization and composition. Specialization is the ability to create new versions of a conversation that are more detailed than the original version; it is akin to the idea of subclassing in an object-oriented language. Composition is the ability to combine two conversations into a new, compound conversation. Development of these two capabilities will entail the creation of syntax for expressing a new conversation in terms of existing conversations, and for linking the appropriate pieces of the component conversations. It will also demand solution of a variety of technical problems, such as naming conflicts, and the merger of semantic descriptions of the conversations.

Conversation sharing

A standardized conversation language, as proposed above, dictates how conversations will be represented; however, it does not say how such representations are shared among agents. While the details of how conversation sharing is accomplished are more mundane than those of conversation representation, they are nevertheless crucial to the viability of dynamic conversation-based systems. Three questions present themselves:

- How can an agent map from the name of a conversation to the specification of that conversation?
- How can one agent communicate to another the identity of the conversation it is using?
- How can an agent determine what conversations are handled by a service provider that does not yet know of the agent's interest?

Conversations Sets as APIs

The set of conversations in which an agent will participate defines an interface to that agent. Thus, standardized sets of conversations can serve as abstract agent interfaces (AAIs), in much the same way that standardized sets of function calls or method invocations serve as APIs in the traditional approach to system-building. That is, an interface to a particular class of service can be specified by identifying a collection of one or more conversations in which the provider of such a service agrees to participate. Any agent that wishes to provide this class of service need only implement the appropriate set of conversations. To be practical, a naming scheme will need to be developed for referring to such sets of conversations, and one or more agents will be needed to track the development and dissolution of particular AAIs. In addition to a mechanism for establishing and maintaining AAIs, standard roles and ontologies, applicable to a variety of applications, will need to be created.

There has been little work on communication languages from a practitioner's point of view. If we set aside work on network transport protocols or protocols in distributed computing (e.g., CORBA) as being too low-level for the purposes of intelligent agents, the remainder of the relevant research may be divided into two categories. The first deals with theoretical constructs and formalisms that address the issue of agency in general and communication in particular, as a dimension of agent behavior (e.g., AOP (Shoham 1993)). The second addresses agent languages and associated communication languages that have evolved somewhat to applications (e.g., TELESCRIPT (White 1995)). In both cases, the bulk of the work on communication languages has been part of a broader project that commits to specific architectures.

Agent communication languages like KQML provide a much richer set of interaction primitives (e.g., KQML's performatives), support a richer set of communication protocols (e.g., point-to-point, brokering, recommending, broadcasting, multicasting, etc.), work with richer content languages (e.g., KIF), and are more readily extensible than any of the systems described above. However, as discussed above, KQML lacks organization at the conversation level that lends context to the messages it expresses and transmits. Limited work has been done on implementing conversations for software agents, and almost none has been done on expressing those conversations. As early as 1986, Winograd and Flores (Winograd & Flores 1986) used state transition diagrams to describe conversations. The COOL system (Barbuceanu & Fox 1995) has perhaps the most detailed current finite state automata model to describe agent conversations. Each arc in a COOL state transition diagram represents a message transmission, a message receipt, or both. One consequence of this policy is that two different agents must use different automata to engage in the same conversation. COOL also uses an :intent slot to allow the recipient to decide which conversation structure to use in understanding the message. This is a simple way to express the semantics of the conversation, though it is not sufficient for sophisticated reasoning about and sharing of conversations.

Other conversation models that have been developed include those of Parunak (Parunak 1996), Chauhan (Chauhan 1997), who uses COOL as the basis for his multi-agent development system, Kuwabara et al. (Kuwabara 1995), who add inheritance to conversations, Nodine and Unruh (Nodine & Unruh 1997), who use conversation specifications to enforce correct conversational behavior by agents, Bradshaw (Bradshaw *et al.* 1998), who introduces the notion of a conversation suite as a collection of commonly-used conversations known by many agents, and Labrou (Labrou & Finin 1997a), who uses definite clause grammars to specify conversations. While each of these makes contributions to our general understanding of conversations, none show how descriptions of conversations might be shared by agents and used directly by them in implementing conversations.

Defining common agent services via conversations

A significant impediment to the development of agent systems is the lack of basic standard agent services that can be easily built on top of the conversation architecture. Examples of such services are: name and address resolution; authentication and security services; brokerage services; registration and group formation; message tracking and logging; communication and interaction; visualization; proxy services; auction services; workflow services; coordination services; and performance monitoring services. Services such as these have typically been implemented as needed in individual agent development environments. Two such examples are an agent name server and an intelligent broker.

An Overview of Jackal's Design

Jackal was designed to provide comprehensive functionality, while presenting a simple interface to the user. Thus, although Jackal consists of roughly seventy distinct classes, all user interactions are channeled through one class, hiding most details of the implementation.

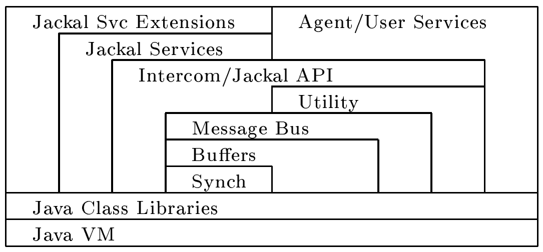

Figure 1: Jackal Architecture

Although there are significant benefits in some cases to sharing a Jackal instance among several agents, the typical usage is as an accessory to an individual agent. Thus, the Jackal architecture does not describe a multi-agent system based around a shared tuple space, as it is often perceived, but a private system of which each agent in a system owns an instance.

Architecture

As illustrated in Figure , Jackal has a layered architecture which facilitates dynamic reconfiguration. Its native execution environment is standard, off-the-shelf Java. Central to Jackal's operation is a set of enhanced synchronization primitives and buffers, which are used to tie together its very loosely coupled components. The *Message Bus* is the essence of Jackal. Consisting principally of the conversation interpreters and a message redistribution system, it is the common path for all message traffic in a Jackal-based agent. This Bus, wrapped along with some additional utilities, by the Jackal API, is referred to as the Jackal *Core.* Both Jackal and agent services interact with the Core and each other through the API. Some examples of Jackal services are the Agent Naming Services, and Message Transport Services. The Jackal *Package* as it is typically distributed consists of the Core and a set of standard services.

Intercom and the Jackal Core

The Intercom class is the bridge between the agent application and Jackal. The only visible component of the Core, it controls startup and shutdown of Jackal, provides the application with access to internal methods, houses some common data structures, and plays a supervisory role to the communications infrastructure.

Message Bus

All messages, between agents or even intra-agent components, traverse Jackal's Message Bus. Through use of the Message Transport Service, the Bus can be viewed as a distributed entity, and messages may be passed to symbolically named entities, without regard to their physical location.

Conversations Based largely on the work of Labrou and Finin (Labrou & Finin 1997b) regarding a semantics for KQML, we have created protocols which describe the correct interactions for various performatives and subsequent messages. These protocols are 'run' as independent threads for all current conversations. This allows for easy context management, while providing constraints on language use and a framework for low-level conversation management. This is in contrast with earlier approaches (e.g., TKQML (Cost *et al.* 1997)) that require the agent to maintain context on their own.

The Conversation Space is a virtual entity, consisting of the collection of currently active conversations, run by distinct threads on individual protocol interpreters. Messages are associated with current (logical) threads based on their ID and assigned to ongoing conversations. If no such assignment can be made, a new conversation appropriate to the message is started. Declarative conversation specifications are downloaded as needed at runtime from an online repository. They can specify something as simple as a query-response interaction, or as complex as a sophisticated, multi-party negotiation and beyond. In conjunction with an ontology of well-known actions, these conversations can be made to implement a wide range of agent behaviors.

The conversation management component offers a number of significant benefits to the agent:

- Running conversations in individual threads provides maximum flexibility.
- Conversations, in conjunction with the Distributor, route messages automatically to the threads that need them.
- Each conversation maintains a local store, which can be accessed by the agent via a message ID, and which serves as the conversation's context.
- Since conversations are declaratively specified, they can be loaded on demand. Our current agents download only the conversations they will need.
- The conversation mechanisms and the specification are almost completely independent of the content or message language used, and so could be easily be tuned work in a 'multi-lingual' environment.
- Actions can be associated with conversation structures, enhancing their utility.

Distributor The Distributor is a Linda-like blackboard, which serves to match messages with requests for messages. This is the sole interface between the agent and the message traffic. Its concise API allows for comprehensive specification of message requests. Requesters are returned message queues, and receive all return traffic through these queues. Requests for messages are based on some combination of message, conversation or thread ID, and syntactic form. They also permit actions, such as removing an acquired message from the blackboard or marking it as read only. A priority setting determines the order or specificity of matching. Finally, requests can be set to persist indefinitely, or terminate after a certain number of matches.

Services

A service here refers to either components of the controlling agent, or subthreads of Jackal itself. Two services packaged with Jackal are the Message Transport Service and the Agent Naming Service.

Message Transport Service Jackal runs a Transport Module for each protocol it uses for communication. Jackal 3.0 comes with a module for TCP/IP, which supports SSL, and one for shared memory communication within a Java Virtual Machine. Users can create and add additional modules for other protocols. A Transport Module is responsible for receiving messages at some known address, and transmitting messages out via a given protocol.

A mechanism known as the Switchboard acts as an interface between the Transport Modules and the rest of Jackal, facilitating the intake of new messages, and carrying out transmission requests from the application. Utilizing an intelligent address cache, the Switchboard must formulate a plan for the delivery of a message and implement it, without creating a bottleneck to message traffic. The address cache is a multilayered cache supporting various levels of locking, allowing it to provide high availability. Unsuccessful address queries trigger underlying KNS lookup mechanisms, while blocking access to only one individual listing.

Naming and Addressing Service In any multi-agent system, the problem of *agent naming* arises: how do agents refer to each other in a simple, flexible, and extensible way? If the system in question employs a standard communication language such as KQML, another requirement is that agents must be able to refer to KQML-speaking agents in the outside world. Within the development of Jackal, we propose KNS, a hierarchical naming scheme designed to support dynamic communities of collaborating, mobile KQML-speaking agents using a variety of transport protocols. Jackal supports KNS transparently through an intelligent address cache.

Standard Jackal services exist to implement KNS, and allow any agent to register with any other agent, facilitating the formation of relationships or teams. Agents can hold multiple identities, and choose which to use in different situations. Protocols implemented by the naming services allow agents to easily discover other agents, regardless of the their current location or chosen identity.

Enterprise Integration

The production management system used by most of today's manufacturers consists of a set of separate application softwares, each for a different part of the planning, scheduling, and execution (P/E) process (Vollmann, Berry, & Whybark 1992). Most P/E applications are legacy systems developed independently over many years, and are not equipped to handle complex business scenarios (Bermudez 1996; Jennings *et al.*

1996). Typically, such scenarios involve the coordination of responses by several P/E applications to external environment changes (price fluctuations, changes of requests from customers and suppliers, etc.) and internal execution dynamics within an enterprise (resource changes, mismatches between plan and execution, etc.). Timely solutions to these scenarios are crucial to agile manufacturing, especially in the era of globalization, automation, and telecommunication (Dourish & Bellotti 1992). Currently, these scenarios are primarily handled by human managers, and the responses are often slow and less than optimal.

The Consortium for Intelligent Integrated Manufacturing Planning-Execution (CIIMPLEX), consisting of several private companies and universities, was formed in 1995 with the primary goal of developing technologies for intelligent enterprise-wide integration of planning and execution for manufacturing (Chu *et al.* 1996). CIIMPLEX has adopted as one of its key technologies the approach of intelligent software agents, and has experimented with several multi-agent systems (MAS) for various difficult tasks involved in enterprise integration. Our effort on MAS development has been concentrated on those P/E scenarios that represent exceptions to the normal or expected business processes and whose resolution involves several P/E applications (Peng *et al.* 1999). Routine, normal communication between P/E applications is handled by another, non-agent based infrastructure that provides persistent data transfer with static, pre-defined communication patterns.

The scenarios for which we developed MASs include:

1. *Process rate change.* Significant changes in the process rate of an essential operation may affect the production plan and schedule. Moreover, depending on the severity of the change, different corrective actions may be required, ranging from doing nothing to to increasing shift or machinery, or even rescheduling production (and possibly delaying delivery of some orders).
2. *Exception in data transfer.* Even in routine exchange transaction data between applications, exceptions such as missing messages, messages out of sync, or messages with incorrect format or parameters may occur. The source of theses errors needs to be identified and corrected, and, if necessary, data needs to be re-sent.
3. *Application initialization.* It is, at times, necessary to introduce into the integrated environment a new application in order to replace an outmoded application or to provide function that is not available in the existing environment. The new application needs to be brought into sync with the rest of the system (e.g., it needs to populate its own database with appropriate data from existing applications so that it can start work from a state that is consistent with the rest of the system.)

To provide integrated solutions to the above outlined scenarios, as simple as they are, is by no means a trivial undertaking. First, specialized agents need to be developed to provide functions which are not covered by any of the existing P/E applications, such as exception detection, data collection and mining, and impact analysis. As integration tasks, these functions fall into the 'white space' between the P/E applications. Next, a reliable and flexible inter-agent communication infrastructure needs to be developed to allow agents to effectively share information, knowledge, and services. Finally, a mechanism for the runtime collaboration of all these pieces also needs to be developed.

In the next section, we will describe in detail an MAS we developed for the process rate change scenario. In general, all MASs for the above scenarios include an Agent Name Server (ANS) and a Broker Agent (BA) in order to facilitate the coordination of other, specialized agents. All agents use the KQML as the agent communication language, and use a subset of KIF that supports Horn clause deductive inference as the content language. A special service agent, called the Gateway Agent (GA), is created to provide interface between the agent world and the application world. GA's functions, among other things, include making connections between the transport mechanisms (e.g., between TCP/IP and MQ Series) and converting messages between the two different formats (KQML/KIF and Business Object Document (BOD)). These agent systems are all supported by Jackal, the agent communication infrastructure developed by the consortium (Cost *et al.* 1998). From a pragmatic point of view, we have found these experiences to demonstrate the value of the following features of Jackal in supporting the development of an MAS.

- It is light-weight with minimum operational overhead.
- It is easy to use by the agent developer.
- It provides mechanisms to ensure the syntactical and semantic correctness of messages.
- It is flexible in switching between different transport mechanisms and in specifying conversation policies.

An Application Example

In this section, we demonstrate how the CIIMPLEX agent system supports intelligent enterprise integration through a simple business scenario involving some real manufacturing management application software systems.

The Scenario

The scenario selected, called *process rate change* and depicted in Figure 2, occurs when the process time of a given operation on a given machine is reduced significantly from its normal value. When this type of event occurs, different actions need to be taken based on the type of operation and the severity of the rate reduction. Some of the actions may be taken automatically according to the given business rules, and others may involve human decisions. Some actions may be as simple as recording the event in the logging file, while others may be complicated and expensive, such as requesting such as a rescheduling based on the changed operation rate. Two real P/E application programs, namely the FactoryOp (a MES by IBM) and MOOPI (a Finite Scheduler by Berclain), are used in this scenario.

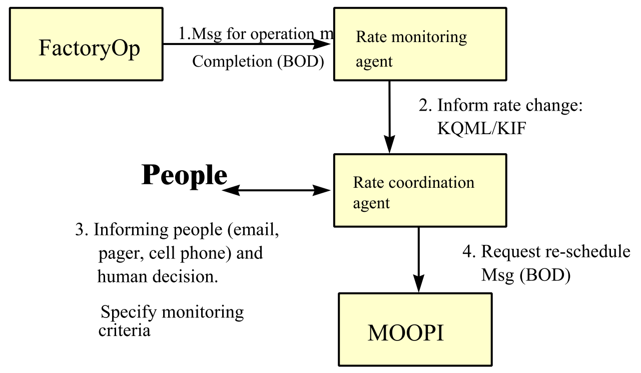
FactoryOp
1.Msg for operation m
Completion (BOD)
Rate monitoring agent
2. Inform rate change: KQML/KIF
People
Rate coordination agent
3. Informing people (email, pager, cell phone) and human decision.
Specify monitoring criteria
4. Request re-schedule Msg (BOD)
MOOPI

Figure 2: The "process rate change" scenario

The Agents

Besides the three service agents, Agent Name Server (ANS), Broker Agent (BA), and GA, the multi-agent system also employs the following special agents to support managing this scenario.

1. The Process Rate Agent (PRA), featured below, is both a mining agent and a monitoring agent for shop-floor activities. As a mining agent, PRA requests and receives the messages containing transaction data of operation completion from GA. The data originates from FactoryOp in the BOD Format, and is converted into KIF format by GA. PRA aggregates the continuing stream of operation completion data and computes the current mean and standard deviation of the processing time for each operation. It also makes the aggregated data available for other agents to access. As a monitoring agent, PRA receives from other agents the monitoring criteria for disturbance events concerning processing rates and notifies the appropriate agents when such events occur.
2. The Scenario Coordination Agent (SCA) sets the rate monitoring criterion, receives the notification for rate changes that meet the criterion, and decides, in consultation with human decision-makers, appropriate action(s) to take for the changed rate.
3. The Directory Assistance Agent (DA) is an auxiliary agent responsible for finding appropriate persons for SCA when the latter needs to consult human decision-makers. It also finds the proper mode of communication to that person.
4. The Authentication Assistance Agent (AA) is another auxiliary agent used by SCA. It is responsible for conducting authentication checks to see if a person in interaction with SCA has proper authority to make certain decisions concerning the scenario.

The Predicates

Three KIF predicates of multiple arguments are defined. These predicates, OP-COMPLETE, RATE, and RATE-CHANGE, are used to compose the contents of messages between agents in processing the process rate change scenario. The OP-COMPLETE predicate contains all relevant information concerning a completed operation, including P/E-Application-id, machine-id, operation-id, starting and finishing time-stamps, and quantity. The RATE predicate contains all relevant information concerning the current average rate of a particular operation at a particular machine with a particular product. The RATE-CHANGE predicate contains all the information needed to construct a BOD that tells MOOPI a significant rate change has occurred and a reschedule based on the new rate is called for. It is the responsibility of the SCA to compose an instance of the RATE-CHANGE predicate and send it to GA when it deems necessary to request MOOPI for a re-schedule, based on the process rate change notification from PRA and consultation with human decision makers.

Agent Collaboration and the Message Flow in the Agent System

Figure 3 depicts how agents cooperate with one another to resolve the rate change scenario, and sketches the message flow in the agent system. For clarity, ANS and its connections to other agents are not shown in the figure. The message flow employed to establish connections between SCA and DA and AA (brokered by BA) is not shown.

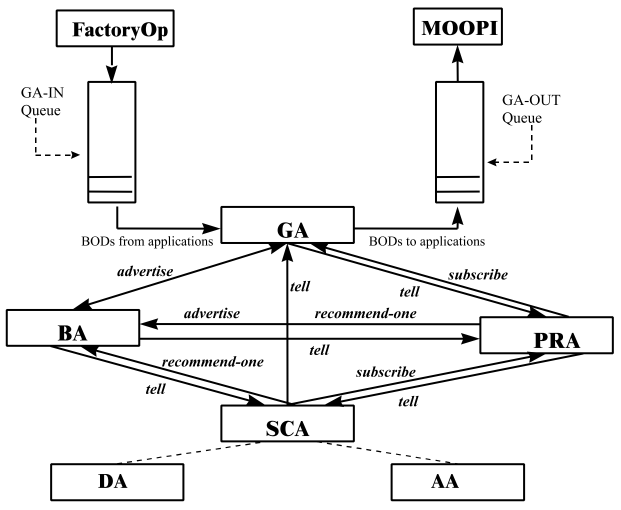

Figure 3: The agent system for "process rate change" scenario

Each of these agents needs information from others to perform its designated tasks. Since there is no predetermined connection among the agents, the broker agent (BA) plays a crucial role in dynamically establishing communication channels for inter-agent information exchange.

GA advertises that it can provide the OP-COMPLETE predicate. It also advertises its ability to handle the RATE-CHANGE predicate. PRA advertises that it has current process rates available for some operations in the form of the RATE predicate. The following is an example an of advertise message from GA to BA.

```
(advertise
  :sender     GA
  :receiver   BA
  :reply-with <a unique id>
  :content    (subscribe :content (ask-one
              :content (OP-COMPLETE ?x1  ?xn))))
```

PRA asks BA to recommend an agent that can provide the OP-COMPLETE predicate, and receives the recommendation of GA in response. Similarly, SCA asks BA to recommend an agent that can answer queries about the RATE predicate and receives PRA in response. It also asks BA to recommend an agent that can provide RATE-CHANGE predicates and receives GA in response. The following is an example of recommend-one message from PRA.

```
(recommend-one
  :sender     PRA
  :receiver   BA
  :reply-with <a unique id>
  :content    (subscribe :content (ask-one
              :content (OP-COMPLETE ?x1 ?xn))))
```

In response, BA sends the following tell message to PRA.

```
(tell
  :sender      BA
  :receiver    PRA
  :in-reply-to <id of last>
  :content     (GA))
```

Upon the recommendation from BA, an agent then obtains the needed information by sending ask or subscribe messages to the recommended agent.

When SCA knows from BA that PRA has advertised that it can provide the current rate for certain operations, it may send PRA the following subscribe message.

```
(subscribe
  :sender     SCA
  :receiver   PRA
  :reply-with <a unique id>
  :language   KQML
  :content    (ask-one :language KIF :content
              (and (RATE ?mean) (< ?mean 50))))
```

With this message, SCA tells PRA that it is interested in receiving new instances of the RATE predicate whenever the mean value of the new rate is less than 50. This effectively turns PRA to a process rate monitor with the $mean < 50$ as the monitor criterion. Whenever the newly updated rate satisfies this criterion, PRA immediately notifies SCA by sending it a tell message with the new rate's mean and standard deviation.

Figure 4 shows the abbreviated Java source code for the PRA agent. The PRA first initializes its databases, and prepares for communication by creating an instance of Jackal; Intercom performs startup functions (including registration with the ANS) and provides access to the Jackal API. Next, PRA advertises itself to the broker (BA) as a source of statistical data, and requests a recommendation for a raw data source. Note that Intercom's one-parameter attend method causes a message to be sent, and blocks waiting for that messages reply. This is the simplest use of Jackal's messaging facilities. One it receives the name of an agent, PRA sends that agent a subscription request for a raw data stream; it does this by spawning a subthread which will manage the incoming data, passing the thread an reference to the agent's Jackal instance. Then the PRA enters a cycle of waiting for data to accumulate, and compiling statistics. The subscription thread will also manage incoming requests for data.

```
class PRA {
  public static RateDatabase Rate = new RateDatabase();
  public static Database msgDB = new Database(); // messages
  public static int Rate_updated = 0;  // # samples observed

  public static void main(String[] args) throws Exception
  {
    ShowOpWin win = new ShowOpWin();        // PRA interface
    Intercom intercom = new
      Intercom("PRA","file:///C:/agents/pra.kqmlrc");
    try {             // next, send a ADVERTISE to BA(Broker)
      KQMLMessage advertise =
        new KQMLMessage("(advertise :receiver BA.ANS :content " +
                        "(subscribe :content (ask-one :content " +
                        " (RATE 1 1 ? ? ? ? ?))))");
      KQMLMessage response = intercom.attend(advertise);

      while(true) { // send RECOMMEND to BA
        KQMLMessage recommend =
          new KQMLMessage("(recommend-one :content " +
                          "(subscribe :content " +
                          "(ask-one :content (RO 1 1 ? ? ? ?)))))");
        recommend.put("receiver","BA.ANS");
        response = intercom.attend(recommend);
        if (response!=null) break;
      }

      KQMLMessage subscribe = // PRA now sends a SUBSCRIBE
        new KQMLMessage("(subscribe :content " +
                        "(ask-one :content (RO 1 1 ? ? ? ?)))");
      subscribe.put("receiver", response.get("content"));
      Sub__Client subClient(this, subscribe);
    }
    catch (MessageX exception) {intercom.stderr(e) ;}
    catch (InterruptedException e) { intercom.stderr(e); }

    // set up computational elements
    ROmessageFromPRAForRATE Ref = new ROmessageFromPRAForRATE(1);
    ROmessageFromPRAForRATE RefA = new ROmessageFromPRAForRATE();
    ROmessageFromPRAForRATE RefB = new ROmessageFromPRAForRATE();

    while (true) { // poll intermittently for data
      while ((msgDB.size())<5) {
        Thread.currentThread().sleep(20); }

      for (int i = 0; i<msgDB.size(); i++) { // comp statistics
        Ref.set((String)msgDB.elementAt(i));
          if (Ref.machn == 65) { /* 65 = 'A' */
            if (RefA.set(Ref)) // PERFORM CALCULATIONS/UPDATE
          else {
            if (RefB.set(Ref)) // PERFORM CALCULATIONS/UPDATE
      }
      msgDB.removeAllElements();
    }
  }
}
```

Figure 4: CIIMPLEX's Process Rate Agent (PRA)

Summary

Jackal provides developers with an easy to use facility for KQML, supporting the use of conversation based protocols. In addition, it provides basic services such as hidden address resolution. These features make it a valuable asset in developing agents for manufacturing information flow.

References

Barbuceanu, M., and Fox, M. S. 1995. COOL: A language for describing coordination in multiagent systems. In Lesser, V., ed., *Proceedings of the First International Conference on Multi–Agent Systems*, 17–25. San Francisco, CA: MIT Press.

Bermudez, J. 1996. Advanced planning and scheduling systems: Just a fad or a breakthrough in in manufacturing and supply chain management? Technical report, Advanced Manufacturing Research, Boston, Massachussetts.

Bradshaw, J. M.; Dutfield, S.; Benoit, P.; and Woolley, J. D. 1998. KAoS: Toward an industrial-strength open agent architecture. In Bradshaw, J. M., ed., *Software Agents*. AAAI/MIT Press.

Chauhan, D. 1997. JAFMAS: A java-based agent framework for multiagent systems development and implementation. Master's thesis, ECECS Department, University of Cincinnati.

Chu, B.; Tolone, W. J.; Wilhelm, R.; Hegedus, M.; Fesko, J.; Finin, T.; Peng, Y.; Jones, C.; Long, J.; Matthes, M.; Mayfield, J.; Shimp, J.; and Su, S. 1996. Integrating manufacturing softwares for intelligent planning-execution: A CIIMPLEX perspective. In *Plug and Play Software for Agile Manufacturing, SPIE International Symposium of Intelligent Systems and Advanced Manufacturing.*

Cost, R. S.; Soboroff, I.; Lakhani, J.; Finin, T.; and Miller, E. 1997. TKQML: A scripting tool for building agents. In Wooldridge, M.; Singh, M.; and Rao, A., eds., *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *LNAI*. Berlin: SV. 336–340.

Cost, R. S.; Finin, T.; Labrou, Y.; Luan, X.; Peng, Y.; Soboroff, I.; Mayfield, J.; and Boughannam, A. 1998. Jackal: A java-based tool for agent development. In Baxter, J., and Brian Logan, C., eds., *Working Notes of the Workshop on Tools for Developing Agents, AAAI '98*, number WS-98-10 in AAAI Technical Reports, 73–82. Minneapolis, Minnesotta: AAAI.

Dickenson, I. 1997. Agent standards. Technical report, Foundation for Intelligent Physical Agents.

Dourish, P., and Bellotti, V. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work: Sharing Perspectives (CSCW '92)*, 107–114.

Finin, T.; Labrou, Y.; and Mayfield, J. 1997. *Software Agents*. MIT Press. chapter KQML as an agent communication language.

FIPA. 1997. FIPA 97 specification part 2: Agent communication language. Technical report, FIPA - Foundation for Intelligent Physical Agents.

Frost, H. R. 1998. Java Agent Template. Online Documentation: http://cdr.stanford.edu/ABE/JavaAgent.html.

Jennings, N. R.; Faratin, P.; Norman, T. J.; O'Brien, P.; Wiegand, M. E.; Voudouris, C.; Alty, J. L.; Miah, T.; and Mamdani, E. H. 1996. Adept: Managing business processes using intelligent agents. In *Proceedings of BCS Expert Systems Conference (ISP Track).*

Kuwabara, K. 1995. AgenTalk: Coordination protocol description for multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS '95).* AAAI/MIT Press.

Labrou, Y., and Finin, T. 1997a. Comments on the specification for FIPA '97 AGENT COMMUNICATION LANGUAGE. Internet document.

Labrou, Y., and Finin, T. 1997b. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97).* Morgan Kaufman.

Martin, D. L.; Cheyer, A. J.; and Moran, D. B. 1998. Building distributed software systems with open agent architecture. In *Proceedings of the Third Internations Conference on Practical Applications of Intelligent Agents.*

Nodine, M. H., and Unruh, A. 1997. Facilitating open communication in agent systems: the InfoSleuth infrastructure. In Singh, M.; Rao, A.; and Woolridge, M., eds., *Proceedings of the 14th Annual Workshop on Agent Theories, Architectures and Languages (ATAL '97).*

Parunak, H. V. D. 1996. Visualizing agent conversations: Using enhanced dooley graphs for agent design and analysis. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS '96).*

Peng, Y.; Finin, T.; Labrou, Y.; Cost, R. S.; Chu, B.; Long, J.; Tolone, W. J.; and Boughannam, A. 1999. An agent-based approach for manufacturing integration - the CIIMPLEX experience. *International Journal of Applied Artificial Intelligence.*

Petrie, C. 1998. JATLite. Online Documentation: http://java.stanford.edu/.

Shoham, Y. 1993. Agent–oriented programming. *Artificial Intelligence* 60:51–92.

Vollmann, T.; Berry, W.; and Whybark, D. 1992. *Manufacturing Planning and Control Systems.* New York: Irwin.

White, J. 1995. Mobile agents. In Bradshaw, J. M., ed., *Software Agents.* MIT Press.

Winograd, T., and Flores, F. 1986. *Understanding Computers and Cognition.* Addison-Wesley.