The Jackal Agent Communication Infrastructure

R. Scott Cost

Laboratory for Advanced Information Technology/
Center for Architectures for Data-driven Information Processing
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland 21250
cost@acm.org

Abstract. Jackal is a Java-based tool for communication using 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. Additionally, Jackal has been used as a framework for exploring alternative conversation-based approaches to managing agent interaction. In particular, we have been investigating the use of Colored Petri Nets as the underlying formal model for a conversation specification language. This model carries the relative simplicity and graphical representation of the more familiar Finite State Machine, along with greater expressive power and support for concurrency. This paper describes Jackal at a surface and design level, and demonstrates its use in a multi-agent system that supports intelligent integration of enterprise planning and execution. It further describes the use of Colored Petri Nets for the specification of interation protocols within a system such as Jackal.

1 Introduction

Jackal is a Java package that allows applications written in Java to communicate via the KQML [24] 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 this role, it has be used to investigate conversation-based approaches to managing inter-agent communication.

Conversations are a useful means of structuring communicative interactions among agents, by organizing messages into relevant contexts and providing a common guide to all parties. The value of a conversation-based approach is largely determined by the conversational model it uses. The presence of an underlying formal model supports the use of structured design techniques and formal analysis, facilitating development, composition and reuse. Most conversation modeling projects to date have used or extended finite state machines (FSM) in various ways, and for good reason. FSMs are simple, depict the flow of action/communication in an intuitive way, and are sufficient for many sequential interactions. However, they are not adequately expressive to model more complex interactions, especially those with some degree of concurrency. Colored Petri Nets (CPN) [34, 35, 36] are a well known and established model of concurrency, and can support the expression of a greater range of interaction. In addition, CPNs, like FSMs, have an intuitive graphical representation, are relatively simple to implement, and are accompanied by a variety of techniques and tools for formal analysis and design.

We have explored the use of model-based conversation specification in the context of multi agent systems (MAS) supporting manufacturing integration [58]. Agents in our systems are constructed using the Jackal agent development platform [14], and communicate using the KQML agent communication language (ACL) [24]. Jackal, primarily a tool for communication, supports conversation-based message management through the use of abstract conversation specifications, which are interpreted relative to some appropriate model. Conversation specifications, or protocols, can describe anything from simple message/acknowledgment interactions to complex negotiations.

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, followed by a discussion of the domain within which Jackal has been developed - enterprise integration automation - and illustrate this with an example. Having discussed the Jackal framework, we then present CPNs, the model we propose to use, in more detail. Following this, we discuss the implementation of these ideas in a real MAS framework. Finally, we present two examples of CPN use: the first, specification of a simple KQML register conversation, and the next, a simple negotiation interaction.

2 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 [24], FIPA ACL [25], ARCOL [25], ICL [47], AgenTalk [38], KaOS [8], and AOP [64]), and various software tools for them (e.g., TKQML [15], OAA [47], JAT and JATLite [26, 59]), 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 writing agents because it is relatively platform independent and has good language support for multi-threading. Jackal benefits from these properties, and relies exclusively on the Sun 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 KQML 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.

3 Conversation-Based Specification of Interaction

The study of 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 [40, 41, 42, 43], and has been formally recognized in the latest FIPA draft standard [25, 17].

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 begins to break down as the range of interactions in which an agent might participate increases, necessitating selection based on a number of additional factors relating to the current message and others which preceded it. Missing from the traditional message-level processing, but required for this more complex scenario, is a notion of message context.

A notion growing in popularity is that the unit of communication between agents should be the conversation. This is evidenced by the advent of a conversation policies workshop at the 1999 Autonomous Agents Conference. 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. 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. Also, conversation structure can be separated from the actions to be taken by an agent engaged in the conversation, facilitating the reuse of conversations in multiple contexts.

Until very recently, little work has been devoted to the problem of conversation specification and implementation for mediated architectures. Increased interest is evidenced by the advent of a workshop on conversation policies at the Third International Conference on Autonomous Agents, in 1999. Strides must be taken to make conversation specifications easy to encode and reuse. Additionally, libraries of specifications must be compiled, along with an ontologies of conversations.

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 or capability?

3.1 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. This is the motivation behind our investigation of CPNs as an alternative mechanism. 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. For some conversations, the set of participants may change during the course of the interaction.

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. However, reliance on a full-blown, highly expressive knowledge representation language may limit a specification's usefulness. 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 many basic 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 [7, 8]. Most complex interactions, however, will require the use of a model that is more expressive, but which retains many of the positive features of DFAs; we will return to this later.

The capabilities we have outlined 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 inheriting a subclass 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.

3.2 Conversation Sharing

A standardized conversation language, as proposed above, dictates how conversations should be represented; however, it does not say how such representations are to be 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 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?

3.3 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.

As mentioned, until recently 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., Agent Oriented Programming (AOP) [64]). The second addresses agent languages and associated communication languages that have evolved to some degree to applications (e.g., TELESCRIPT [68], now Odyssey [28]). 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 [70] used state transition diagrams to describe conversations. The COOL system [3] has perhaps the most detailed current state transition-based 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. We believe that a conversation standard should clearly separate message matching from actions to be carried out when a match occurs; doing so will allow a single conversation specification to be used by all participants in a conversation. This, in turn, will allow conversation specifications to describe standard services, both from the viewpoint of the service provider, and from that of the service user.

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. We argue below that more general descriptions of conversation semantics will be needed if agents are to acquire and engage in new conversations on the fly. The challenge will be to develop a language that is general enough to express the most important facts about a conversation, without being so general that it becomes an intellectual exercise, or too computationally expensive to implement.

Other conversation models have been developed, using various approaches. Extended FSM models, which, like COOL, focus more on expressivity than adherence to a model include Kuwabara et al. [39, 38], who add inheritance to conversations; Wagner et al. [67]; and Elio and Haddadi [19], who defines a multi-level state machine, or ATM. A few others have chosen to stay within the bounds of a DFA, such as Chauhan [9], who uses COOL as the basis for her multi-agent development system, [1] Nodine and Unruh [54, 55], who use conversation specifications to enforce correct conversational behavior, and Pitt and Mamdani [60], who use DFAs to specify protocols for BDI agents. Also using automata, Martin et al. [48] employ Push-Down Transducers (PDT). Lin et al. [46] and Cost et al. [13] demonstrate the use of Colored Petri Nets, and Moore [53] applies state charts. Parunak [57] employs Dooley Graphs. Bradshaw [7] introduces the notion of a conversation suite as a collection of commonly used conversations known by many agents. Labrou [40] uses definite clause grammars to specify conversations. While each of these works makes contributions to our general understanding of conversations, more work must be done in getting agents to share and use conversations.

3.4 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, treated below, and an intelligent broker.

[1] More recent work with this project, JAFMAS, explores conversion of policies to standard Petri Nets for analysis [27].

Agent Name Server At first blush, the problem of mapping from an agent name to information about that agent (such as its address) seems trivial. However, solving this problem in a way that can easily scale as the number of users and amount of data to be processed grows is difficult. We believe that development of a successful symbolic agent addressing mechanism demands at least two advances:

1. A simple naming convention to place each role an agent might play in an organization at a unique point in a namespace for that organization. Currently there is no widely accepted mechanism for universal unique agent naming (in the way that there now is, e.g., for Internet hosts or web documents).
2. An efficient, scalable name service protocol for mapping from symbolic role names to information about the agents that fill those roles.
   The proposed KNS (Section 4) meets both of these demands.

To a large extent, the desired techniques can be modeled after existing name service techniques such as the DNS (which is widely implemented) and CORBA (whose namespace mechanisms are only narrowly implemented). Such techniques are well studied, highly reliable, and scalable. Agent name service will differ from DNS primarily in that agents will tend to appear, disappear, and move around more frequently than do Internet hosts. This will necessitate the development of naming conventions that are less rigid than those used in DNS, and algorithms for mapping from names to agent information that do not rely on the static local databases found in DNS.

Intelligent Broker A system that is to respond to the demands of multiple users, with needs that vary over time, under an ever-increasing query load must be able to do on-the-fly matching of queries to documents and services. In an agent-based architecture, this means that one agent must be able to dynamically discover other agents based on the content of their knowledge. It should exploit the research on conversations and the symbolic agent-addressing scheme described above, while at the same time fitting neatly into existing brokered systems. Such systems will continue to see a single broker where there had been a single broker all along; now, however, that broker will have the option of coordinating many other disparate brokers of varying capabilities.

4 KNS

Before communication can take place, there must be a known destination. KNS adds a communication layer in which symbolic names are mapped to actual transport addresses. In addition, however, it offers advanced support for dynamic group formation and disbanding, and maintenance of persistent, distributed agent identity. KNS is currently being used within Jackal, UMBC's Java-based agent development framework. This section introduces the basic concepts underlying KNS.

The problem of agent naming is central to agent communication. We would like to be able to talk about agents with reasonable certainty that we are all discussing the same ones, and we would like to be able to send messages to agents that we know by name. The former statement argues that names should be unique, within some context, and the latter that they should be resolvable into addresses which can be used by our underlying transport mechanism. This can be accomplished by having the address either implicitly or explicitly encoded in the name, or by providing a service to perform the resolution.

We can think of the problem in three layers of abstraction. At the top is the agent's identity, that which differentiates it from all other agents. One step below this is the name, and at the base is the address. Although an agent's identity will never change (by definition), its name(s) may, based on changing roles or associations. Addresses may change even more frequently, because of physical relocation or constraints of the underlying operating system. This dynamism argues for the use of service-based resolution (SBR) between both layers. In addition, SBR allows for the use of symbolic names with useful meaning.

Uniqueness is a more difficult problem. It is trivial to assign agents simple unique identifiers (e.g. serial numbers) from some central authority. However, if an agent holds such an identifier, it must still present the tag to some authority for resolution. An address, for instance a URL, eliminates the need for SBR altogether, but ties the agent to that address. We propose to represent an agent's identity by the collection of names it uses which we call the persistent distributed identity (PDI). This set can change as names are added or removed, but it remains a constant reference point for the agent itself. Protocols added to the basic agent registration scheme maintain the PDI with little overhead. In addition to the benefits of identity, the scheme provides a valuable mechanism for storing and retrieving information relating to the agent such as certificates.

KNS is a set of protocols for agent naming and addressing. They were developed and used as a basis for the design of Jackal 3.0. This section provides an overview of KNS.

The KNS covers several layers of abstraction, and provides basic support for agent operation. It should be noted that the KNS protocols are layered on top of the KQML, or linguistic, layer.

First, some definitions:

1 Definition (Given Name) *A symbolic name chosen for the agent application by itself or some other authority.*

2 Definition (Local Name) *A Given Name qualified by a numeric index, and assigned by a Domain Registrar upon registration.*

3 Definition (FQAN) *Fully Qualified Agent Name; the canonical form for names in KNS. Every FQAN names a domain.*

4 Definition (Domain) *A virtual group, defined by registration and unregistration, and managed by the owner of the FQAN which names it.*

5 Definition (Alias) *A FQAN that is owned by the same agent as another FQAN is an alias for that FQAN.*

6 Definition (Alias Set) *Also 'Alias Net'; for an agent A, the set of domains with which A is registered.*

7 Definition (AS) *Agent Server; holds a database of information for given agent.*

8 Definition (PAS) *Primary AS; there is only one for any given agent.*

9 Definition (SAS) *Backup (secondary) AS; serve as backup to PAS. There can be any number.*

4.1 Assumptions

KNS makes some basic assumptions about the environment in which it is used:

- Message delivery by the underlying transport mechanisms is reliable. The protocols do not incorporate any retry mechanism for delivery failure to a specified address. Further, if KNS protocols are properly implemented, including strong message delivery, an agent may be consider unreachable if an expected acknowledgment is not received on a single transmission.
- Authentication (KNS does not specify what kind) in the message transport layer assures that the name in the sender field of the message is in fact the sender of the message. Security in KNS is identity based, so any privileges enjoyed by the named sender are applied to the accompanying transaction.
- Agents purporting to implement KNS correctly and responsibly render services as appropriate.
- It is possible to distinguish agents that implement KNS from those that do not. This relates to the general problem of determining an agents language or message format. Initially, an agent should be given the name of another with which to register; that agent does implement KNS, as do any agents located through KNS name/address resolution. However, new contacts in unrelated systems may not.

4.2 Agent Names

The foundation of KNS is its agent-naming scheme. It encompasses both symbolic and direct (URL-based) names. The symbolic component is modeled after the DNS scheme [50, 51], and extends it to allow a URL to fill the root position of a name. A FQAN is defined as follows:

$$\begin{aligned}
GivenName &= [a - zA - Z0 - 9\_]\{1, 64\} && (1)\\
NameIndex &= 0 \mid ([1 - 9][0 - 9]\{, 10\}) && (2)\\
LocalName &= < GivenName >< NameIndex > && (3)\\
FQAN &= (< LocalName > .) * \\
&\quad (< GivenName > | < LocalName > | < URL >) && (4)
\end{aligned}$$

As in DNS, names registered within a Domain must be unique. Rather than accepting only applications for unique names, KNS adopts the policy of accepting any name and adding a distinguishing suffix. Some examples of FQANs are: bob[4].ans, freida, barbecue[34].cs and fred[2].http://www.umbc.edu/. Since names correspond to entity/Domain relationships, an agent may have any number of names, and may use them interchangeably.

In light of our earlier discussion of name uniqueness, it should be clear that this definition allows for unqualified names. This is included as a convenience, since many contained systems use well-known names for common resources. We assume then that unqualified names are used only in closed contexts in which the address of name root is publicly known. In general, the use of fully qualified names is preferred. The technically correct definition provides for a root of URL only.

Every FQAN represents a Domain. Thus, an agent can 'have', or manage, multiple Domains, although none is required to actively accept Domain registrations. An agent registers with a Domain either with its Given Name, or under another FQAN that it holds. In the latter case, protocols are engaged to update the Alias Set for that agent. In either case, the agent is given a new FQAN, which is derived from the Given Name of the name submitted. For example, if an agent registers orianus.local with freckles.cs[1].umbc.ans (alternatively, freckles.cs[1].umbc.http://jackal.cs.umbc.edu/ans), it may receive the FQAN orianus[14].cs[1].umbc[23].ans.

An alternative is to represent the name of an agent as an actual URL. While this would be enormously convenient, it creates unacceptable naming ambiguity. Any URL should be usable as a legal agent name, for reasons of flexibility, and for compatibility with systems that use only URLs as agent names. Given that constraint, it becomes impossible to determine which portion of a URL constitutes the root, and which the domain. For example, http://jackal.cs.umbc.edu/ans.umbc.cs[1].freckles could indicate four different names, depending on where one decided the root name ended. It is difficult to remedy this problem directly without abusing or outright violating the URL syntax.

4.3 KNS Architecture

KNS is served by a dynamic, distributed database system, depicted in Figure 1. The two databases maintained are the Domain Registry (one for each Domain), and the Agent Registry (one for each agent, and one or more backups). We impose one additional virtual structure on the name hierarchy, called the Alias Set (or AliasSet). The Alias Set consists of all Domains with which an agent has ever registered. One Domain is designated as the Primary Agent Server (PAS), and it hosts the primary agent registry. Likewise, a Backup Agent Server (BAS) hosts the backup agent registry. All Agent Servers (AS) maintain a reference to the target agent's PAS and BASs. ASs are arranged in a star configuration in order to minimize messaging overhead. The AliasSet itself is treated as a single entity; queries are directed to any member, and if necessary, are forwarded directly to the PAS. Member agents notify the PAS of any changes, and the PAS broadcasts updates to the remaining members of the set.

One dependency is that agent information is not discarded. While this is not entirely realistic, is means that agents can be located most of the time, and that more resources can be dedicated to specific localities to increase the level of fault tolerance. For example, under the KNS scheme, if agent bob.erols.ans unregisters from erols.ans, it will still be possible to locate bob through the erols.ans domain. If erols.ans terminates, and ans has lifted its domain, location is still possible. However, if erols.ans goes down catastrophically or otherwise dissolves the domain, it will not be possible to reach bob via its previous name. Agents who are concerned with reachability would therefore prefer to register with strong domains, and would show preference for names that they felt would more reliably persist. This situation could be improved by allowing agents to register, have included in their address information, or send with messages an alternate name; this is reminiscent of the use of sender and reply-to fields.

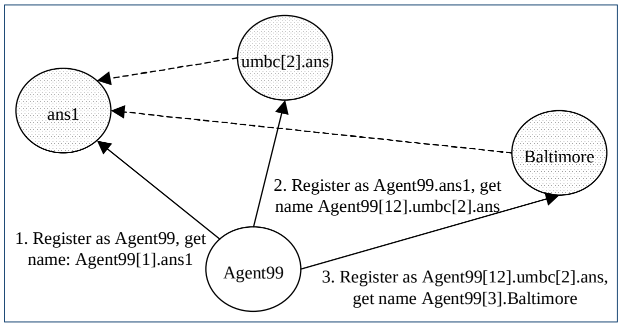
umbc[2].ans
ans1
Baltimore
2. Register as Agent99.ans1, get name Agent99[12].umbc[2].ans
1. Register as Agent99, get name: Agent99[1].ans1
Agent99
3. Register as Agent99[12].umbc[2].ans, get name Agent99[3].Baltimore

Fig. 1. KNS Alias Network. The registrars of Agent99 coordinate to maintain the agent's distributed identity.

4.4 KNS Protocols

KNS specifies protocols for agent addressing and naming, authentication, aliasing and Domain registration. These are sketched below:

1. Group Membership
   (a) Register: Register with a new Domain (multiple registrations are permitted). Registration implies a commitment to membership in a Domain. A registration must contain one address that is reachable by the registrar. The name given must be a FQAN. Registration causes the intended registrar to invoke the protocol for joining an AliasSet, if the name given by the registrant indicates a prior domain association.
   (b) Join: Identify the Alias Set for a registrant, and join. The agent accepts the responsibility of forwarding PAS queries, and becomes eligible to become a BAS for the registering agent, though the latter is not required.

(c) Unregister: Terminate association with a Domain. The registration entry is not deleted; it is moved to a dormant status, and the addresses are cleared.
The potential for unregistration creates instability in the naming hierarchy. For this reason, one of two protocols should be followed in the event that an agent must leave a Domain.
  - Domain Lifting: For each domain owned by the departing agent, the owner of the parent domain takes on the subdomain and its registration responsibilities. This involves a transfer of the registry, and a conversion of the departing agents registration entry from real to virtual.
  - Recursive Domain Dissolution: The agent wishing to unregister first excuses or discharges (below) all agents registered in the Domain that is to be eliminated, using existing protocols. Each in turn does the same until Domain and all of its subdomains are eliminated.

Clearly, lifting is preferable to dissolution, since no naming information is lost. However, dissolution does at least prevent the use of names after they have become invalid.
(d) Excuse: Request that a registrant unregister from a named group. A positive acknowledgment constitutes an implicit unregistration. A negative or no acknowledgment is followed by a discharge.
(e) Discharge: Revoke an agent's membership in a Domain. This action does not require consent or acknowledgment; it should be used only in order to elicit a response once a request to unregister has failed.
(f) Leave an Alias Set: Terminate relationship with principal for that set. If an agent is the principal or a secondary, it must first arrange successful transfer of the database and database responsibilities.

2. Registry Query/Update
   (a) Query for the address(es) of an agent: Note that address queries are posed to Domain registries; therefore, querying an agent for its own address(es) is not provided for. KNS does not prohibit responding to queries about one's own addresses. However, some systems which integrate KNS, e.g. Jackal, do not provide agents with access to information at the message transport level directly.
   (b) Update a registry entry: by adding or deleting an address or other data. It is permitted for an agent to remove all addresses from its registration entry; this does not imply unregistration.
   (c) Invalidate: notify an agent that an address it has provided is invalid. The agent receiving the invalidate should take steps to right the registry for the domain in question, either by posing queries, marking or canceling the offending entry.
3. Agent Information Server Query/Update
   (a) Identify the alias server for an agent of a Given Name.
   (b) Verify a FQAN: This is implemented as an address query, which will return an address packet if the agent's name is found in the registry.
   (c) Get the aliases for an agent of a Given Name.
   (d) Request that another agent replicate a (local) alias database: An agent's PAS may at its discretion request that any or all members of the agent's AliasSet replicate the AID. If an agent accepts the request, it becomes a BAS, and receives updates from the PAS. Its new status is broadcast to the members of the AliasSet.

   (e) Abdicate: A PAS relinquishes control of the AliasSet to a member BAS. Upon acceptance, the abdicating PAS begins forwarding all incoming traffic to the new PAS, while the new PAS broadcasts the change of status to all members of the set. Any agent that serves as a BAS accepts the responsibility of potentially serving as PAS.
   (f) Resign: A BAS notifies the AliasSet's PAS that it will no longer serve as BAS. Only a cursory acknowledgment is required. The resigning agent is still a member of the AliasSet.
4. Additional Features
   (a) Broadcast. Messages sent to a virtual Domain are automatically copied by the registrar of the virtual Domain to all members. This is done as a 'direct' forward; that is, no modification or wrapping of the message. This process repeats itself recursively.

5 An Overview of Jackal's Design

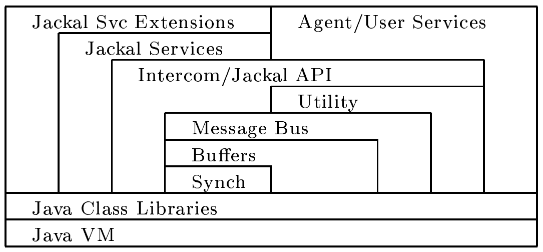

Fig. 2. Jackal Architecture

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. 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.

5.1 Architecture

As illustrated in Figure 5, 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.

5.2 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.

5.3 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 [43] 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 [15]) 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, [2] 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.

The use of the Distributor in Jackal allows the integration of the conversation management utilities easily into existing agents, by providing a flexible, message-based interface.

5.4 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.

[2] Messages in Jackal are represented as Java objects, essentially collections of attribute/value pairs. The values can be of a variety of types. Jackal expects a message to have certain basic attributes (e.g. sender, performative), and places no restrictions on additional attributes. Values which are critical to Jackal's operation are mapped to/from corresponding, internally correct values. The conversation framework itself specifies the methods to be applied to messages, such as comparisons, and so conversation templates can be tuned to any language. Note that this still leaves open the question of management issues, which often vary from system to system.

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 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.

6 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 [66]. Most P/E applications are legacy systems developed independently over many years, and are not equipped to handle complex business scenarios [5, 33]. 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 [18]. 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 [12]. 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 [58]. 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 message format is the OAG's Business Object Document (BOD)

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 BOD). These agent systems are all supported by Jackal [14]. 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.

7 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.

7.1 The Scenario

The scenario selected, called *process rate change* and depicted in Figure 3, 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.

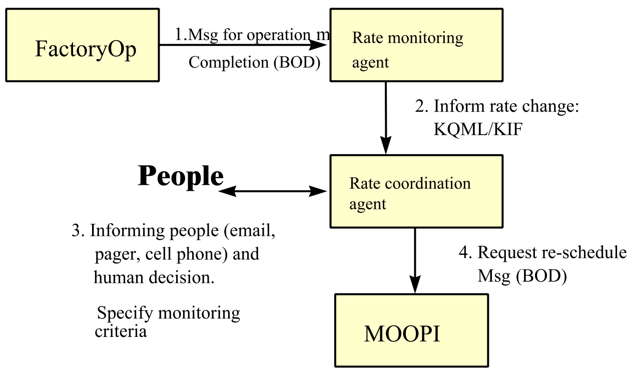
FactoryOp
1.Msg for operation m
Completion (BOD)
Rate monitoring agent
2. Inform rate change: KQML/KIF
People
Rate coordination agent
3. Informing people (email, pager, cell phone) and human decision.
Specify monitoring criteria
4. Request re-schedule Msg (BOD)
MOOPI

Fig. 3. The “process rate change” scenario

7.2 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.

7.3 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 re-schedule 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.

7.4 Agent Collaboration and the Message Flow in the Agent System

Figure 4 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.

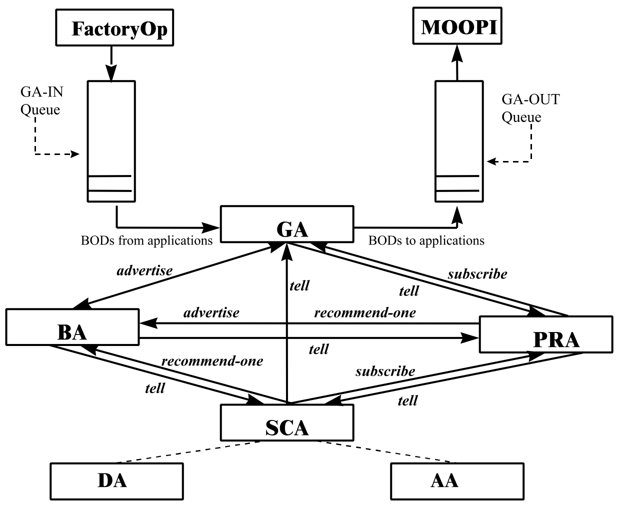

Fig. 4. The agent system for "process rate change" scenario

Each of these agents needs information from others to perform its designated tasks. Since there is no pre-determined 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 5 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.

Figure 6 shows the declarative specification for the ask-one conversation used by the agents in this scenario.

```
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();
    }
  }
}
```

Fig. 5. CIIMPLEX's Process Rate Agent (PRA)

8 Relationship to Current Work

A number of groups are currently developing or marketing Java-based tools and infrastructures, and Jackal shares many features with them. Some of Jackal's defining characteristics are its use of conversation policies, internal message distribution blackboard, independence from the transport layer. Also, it's restriction to agent communication support differentiates it from most other systems, which often integrate more agent functionality, making them at the same time more powerful and less easily integratable. In this section, we will introduce a few of the more well-known Java-based agent frameworks, and discuss their relationship to Jackal.

The InfoSleuth project [54, 4] is very much committed to the domain of distributed information retrieval, although the agent architecture is fairly general.

```
// Conversation Template
// Convention: Initial and accepting states all caps,
//             other states initial caps,
//             arc-labels lower case.
(conversation
  (name kqml-ask-one)
  (author "R. Scott Cost")
  (date "3/4/98")
  (start-state START)
  (accepting-states TOLD)
  (transitions
    (arc (label ask-one) (from START) (to Asked) (match "(ask-one)"))
    (arc (label tell)    (from Asked) (to TOLD)  (match "(tell)"))
    (arc (label deny)    (from Asked) (to TOLD)  (match "(deny)"))
    (arc (label untell)  (from Asked) (to TOLD)  (match "(untell)"))
    (arc (label sorry)   (from Asked) (to TOLD)  (match "(sorry)"))
    (arc (label error)   (from Asked) (to TOLD)  (match "(error)"))))
```

Fig. 6. Conversation Template for KQML Ask-one

Their overall system design employs a standard resource brokered approach. InfoSleuth implements a Java agent shell, which is carefully separated into cleanly interfaced layers: message (astride the Transport Layer), conversation, generic agent, and agent application. The message layer handles message addressing, parameter marshaling, and so forth. The conversation layer imposes language constraints on sequences of messages. The generic agent layer provides the agent application layer with basic services. Conversation policies enforced roughly correspond to the Finin and Labrou [40, 43] semantics for KQML, but the model used is a basic DFA; transitions are determined by performative name only. Aside from its use of an agent shell, InfoSleuth is similar in principal to Jackal, but is internally less sophisticated, and relies on simple DFAs for conversation specification.

Java Agent Template (JAT) [26] is essentially a Java implementation of KQML, in the form of an agent shell. JAT agents can run stand-alone, or as applets with some restrictions. Basic message passing is supported for KQML. An ANS is used to coordinate agents, and the AEE is the basic JVM. JAT is intended to be flexible yet comprehensive. Agents developed with JAT are tightly integrated with the agent shell. JATLite [59] is a successor to JAT, intended to be a much lighter-weight package suitable for use with applets. Of note is its Router facility: applet agents can communicate with other applet agents by sending messages back through an associated Router or Routers (communication by proxy). The Router buffers undeliverable messages, and is supported by a standard ANS. JATLite agents in general are not constrained to communicate through the Router. Neither platform supports the use of conversation policies. Of the two, Jackal is perhaps more similar to JATLite than JAT.

The Aglet project [2, 45], developed at IBM, is a very promising framework for agent mobility. It provides support for the construction of small, roving agents called Aglets, through extension of an agent shell class. Aglets move from place to place by agent-initiated, single entry-point transfer. Places maintain a persistent ‘context’, which the agent can access. Some security is achieved by the use of an agent proxy. All access to the agent, including peer-to-peer communication, takes place via the proxy agent. Since the agent and its proxy need not be collocated, this allows for location transparency. Access to the agent's methods can also be selectively restricted with the proxy's intervention. Communication is supported only through direct or remote method invocation on the receiving agent (or its proxy). The Aglet system differs from Jackal in its comprehensive MAS framework, which includes mobility support. Many infrastructure components, such as name serving, can be added onto Jackal as services, but the library itself is an agent component. Jackal provides more highly developed communication facilities than do Aglets.

Zeus [56] and AgentBuilder [62] are good examples of MAS design tools. While they serve a higher level purpose than Jackal, they could facilitate the development of agents with Jackal components and libraries.

Zeus is a toolkit for building complete agents, from the ground up. It consists of a component library, a suite of visual design tools, and a set of predefined utility agents. The components are designed such that their behaviors are largely defined declaratively, and can be changed at runtime. Agents constructed with Zeus typically have components which facilitate planning and reasoning, communication (via KQML), and which provide a collection of interaction protocols. Agents and MASs are created through a process of task and relationship specification. The resulting entities are produced as Java source code, for independent compilation and execution.

AgentBuilder is a commercial platform for constructing agents based on the RADL (Reticular Agent Definition Language). RADL is an extension of PLACA [65] and AGENT-0 [63], and views the agent as a core of behavioral rules, constrained by beliefs, capabilities, commitments and commitment rules.

9 Colored Petri Nets

Petri Nets (PN), or Place Transition Nets, are a well known formalism for modeling concurrency. A PN is a directed, connected, bipartite graph in which each node is either a *place* or a *transition*. *Tokens* occupy places. When there is at least one token in every place connected to a transition, we say that transition is *enabled*. Any enabled transition may *fire*, removing one token from every input place, and depositing one token in each output place. Petri nets have been used extensively in the analysis of networks and concurrent systems. For a more complete introduction, see [1].

CPNs differ from PNs in one significant respect; tokens are not simply blank markers, but have data associated with them. A token's *color* is a schema, or type specification. Places are then sets of tuples, called *multi-sets*. *Arcs* specify the schema they carry, and can also specify basic boolean conditions. Specifically, arcs exiting and entering a place may have an associated function which determines what multi-set elements are to be removed or deposited. Simple boolean expressions, called *guards*, are associated with the transitions, and enforce some constraints on tuple elements. This notation is demonstrated in examples below. CPNs are formally equivalent to traditional PNs; however, the richer notation makes it possible to model interactions in CPNs where it would be impractical to do so with PNs.

CPNs have great value for conversational modeling, in that:

- They are a relatively simple formal model.
- They have a graphical representation.
- They support concurrency, which is necessary for many non-trivial interactions.
- They are well researched and understood, and have been applied to many real-world applications.
- Many tools and techniques exist for the design and analysis of CPN-based systems.

9.1 Related Work

CPNs are not new, and they have been used extensively for a broad range of applications (see [36] for a survey of current uses). Since their target domain is distributed systems, and the line between that domain and MASs is vague at best, there is much work on which to build. We will review here a few of the more directly related research endeavors.

Holvoet and Verbaeten have published extensively on the subject of agents and PNs. In their 1995 paper, "Agents and Petri Nets" [29], they introduced the idea of enhancing AOP by using high-level nets to model agents, and extended this thought in [30] to a variant called 'Generic Nets'. In 1997, Holvoet and Kielmann introduced PNSOL (Petri Net Semantics for Objective Linda) [31, 32], used to model agents which live in and communicate through the Objective Linda [37] tuple space.

Yoo, Merlat and Briot [71] describe a contract-net based system for electronic commerce that uses a modular design. Among the components are BRICS (Block-like Representation for Interacting Components) [23]), which are derived from CPNs.

Fallah-Seghrouchni and Mazouzi have demonstrated the use of CPNs in specifying conversation policies in some detail, using FIPA ACL as a framework [21, 20, 22]. This work suggests an approach for hierarchical construction of conversations.

Moldt and Wienberg have developed an approach called AOCPN (Agent Oriented Colored Petri Nets) [69, 52]. This system employed an object-oriented language, syntactically similar to C++, which maps onto CPN, extended by 'test arcs' [11, 44]. They show how this approach can be used to model societies of agents as described by Shoham [64]. Their model extends down to the level of individual agent theorem provers, facilitating the logical specification of agent behavior.

Other work of note includes Billington et al. [6], Purvis and Cranefield [61], Lin et al. [46] (above), and Merz and Lamersdorf [49].

Currently, we are investigating the value of CPNs in a general framework for agent interaction specification. Within this scheme, agents use a common language, Protolingua, for manipulating CPN-based conversations. Protolingua itself is very sparse, and relies on the use of a basic interface definition language (IDL) for the association of well known functions and data types with a CPN framework. Agents use Protolingua interpreters to execute various protocols. Protolingua itself is simple in order to facilitate the porting of interpreters to many different platforms.

One advantage to this approach is that a variety of interpreter implementations may be used, and the agent may trade resources for conversational 'power'. A very simple CPN interpreter may be able to efficiently execute very small or simple protocols; an agent may chose to use this in most interactions, while employing more expensive and powerful interpreters for more complex negotiations. In addition to using direct CPN simulators, CPN specifications have a very natural embedding in a general rule-based framework.

To clarify the relationship between agents, interpreters, and protocols, let us assume that a Java-based agent would like to converse with another agent, and that it has determined, through assumption, negotiation, or other means, that it needs to use protocol xyz. It can obtain the declarative specification for xyz, if it does not already have it, from the other agent or from some third party; let's say a protocol server identified through a broker. Xyz contains the wire-frame specification of the protocol (arcs, places, transition), plus schema and functions given in the IDL. The agent can then obtain the executable attachments (as it did the specification) and type specifications appropriate for its interpreter (in the case of Jackal, Java classes and associated methods), and then use the protocol to engage the other agent.

This CORBA-like approach allows the use of very lightweight, universal interpreters without restricting the expressiveness of the protocols used. Note that the purpose of the IDL in Protolingua however is the identification and retrieval of executable modules, not the interaction of distributed components. If types and actions are appropriately specified, they should be suitable for analysis, or translation into some analyzable form. For example, we are using Design/CPN, a tool from Aarhus University, Denmark, for high level design and analysis of protocols. This system uses an extension of ML, CPN-ML, as its modeling language. We plan to translate developed protocols into Protolingua and Java extensions, and restrict modification in such a way that CPN-ML equivalents of the extensions can be used to facilitate analysis of the protocols. As such, CPN-ML has played a major role in influencing the development of Protolingua. For the remainder of this paper, we will focus on the abstract application of CPNs to conversations, rather than their specification in Protolingua.

11 Example: Conversation Protocol

From its inception, Jackal has used JDFA, a loose Extended Finite State Machine (EFSM), to model conversations [14, 58]. The base model is a DFA, but the tokens of the system are messages and message templates, rather than simply characters from an alphabet. Messages match template messages (with arbitrary match complexity, including recursive matching on message content) to determine arc selection. A local read/write store is available to the machine.

CPNs make it possible to formalize much of the extra-model extensions of DFAs. To make this concrete, we take the example of a standard JDFA representation of a KQML Register conversation (see Figure 7) and reformulate it as a CPN. Note that this simplified Register deviates from the [40] specification, in that it includes a positive acknowledgment, but does not provide for a subsequent 'unregister' event. The graphic depiction of this JDFA specification can be seen in Figure 8.

```
// Conversation Template
// Convention: Initial and accepting states all caps,
//             other states initial caps,
//             arc-labels lower case.
(conversation
  (name kqml-ask-one)
  (author "R. Scott Cost")
  (date "3/5/98")
  (start-state START)
  (accepting-states STOP)
  (transitions
    (arc (label reg)   (from START) (to R) (match "(register)"))
    (arc (label reply) (from R) (to STOP)  (match "(reply)"))
    (arc (label error) (from R) (to STOP)  (match "(error)"))
    (arc (label sorry) (from R) (to STOP)  (match "(sorry)"))))
```

Fig. 7. Conversation template for simplified KQML Register

There are a number of ways to formulate any conversation, depending on the requirements of use. This conversation has only one final, or accepting, state, but in some situations, it may be desirable to have multiple accepting states, and have the final state of the conversation denote the *result* of the interaction.

In demonstrating the application of CPNs here, we will first develop an informal model based on the simplified Register conversation presented, and then describe a complete and working CPN-ML model of the full Register conversation.

Some aspects of the model which are implicit under the DFA model must be made explicit under CPNs. The DFA allows a system to be in one state at a time,

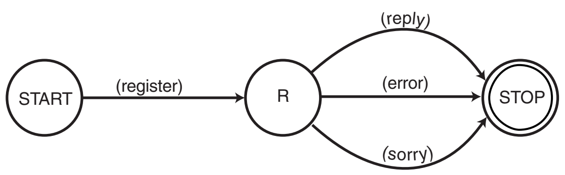

Fig. 8. Diagrammatic DFA representation of the simplified KQML Register conversation

and shows the progression from one state to the next. Hence, the point to which an input is applied is clear, and that aspect is omitted from the diagrammatic representation. Since a CPN can always accept input at any location, we must make that explicit in the model.

We will use an abbreviated message which contains the following components, listed with their associated variable names: performative(p), sender(s), receiver(r), reply-with(id), in-reply-to(re), and content(c).

We denote the two receiving states as places of the names **Register** and **Done** (Figure 9). These place serve as a receipt locations for messages, after processing by the transitions **T1** and **T2**, respectively. As no message is ever received into the initial state, we do not include a corresponding place. Instead, we use a a source place, called **In**. This is implicit in the DFA representation. It must serve as input to every transition, and could represent the input pool for the entire collection of conversations, or just this one. Note that the source has links to every place, but there is no path corresponding to the flow of state transitions, as in the DFA-based model.

The match conditions on the various arcs of the DFA are implemented by transitions preceding each existing place. Note that this one-to-one correspondence is not necessary. Transitions may conditionally place tokens in different places, and several transitions may concurrently deposit tokens in the same place.

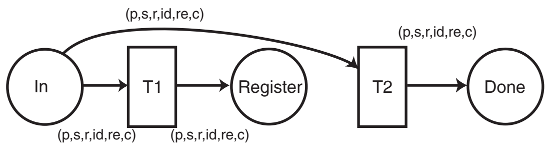

Fig. 9. Preliminary CPN model of a simplified KQML register conversation.

Various constants constrain the actions of the net, such as performative (Figure 10). These can be represented as color sets in CPN, rather than hard-coded constraints. Other constraints are implemented as guards; boolean conditions associated with the transitions. Intermediate places **S**, **R** and **I** assure that sender, receiver and ID fields in the response are in the correct correspondence to the initial messages. **I** not only ensures that the message sequence is observed, as prescribed by the message IDs, but that only one response is accepted, since the ID marker is removed following the receipt of one correct reply. Not all conversations follow a simple, linear thread, however. We might, for example, want to send a message and allow an arbitrary number of asynchronous replies to the same ID before responding (as is the case in a typical Subscribe conversation), or allow a response to any one of a set of message IDs. In these cases, we allow IDs to collect in a place, and remove them only when replies to them will no longer be accepted. Places interposed between transitions to implement global constraints, such as alternating sender and receiver, may retain their markings; that is implied by the double arrow, a shorthand notation for two identical arcs in opposite directions.

We add a place after the final message transaction to denote some arbitrary action not implemented by the conversation protocol (that is, not by an arc-association action). This may be some event internal to the interpreter, or a signal to the executing agent itself. A procedural attachment at this location would not violate the conversational semantics as long as it did not in turn influence the course of the conversation.

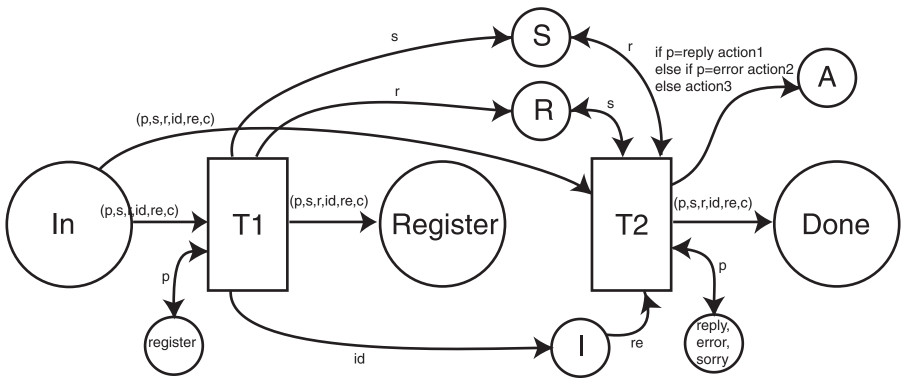

Fig. 10. Informal CPN model of a simplified KQML register conversation.

This CPN is generally equivalent to the JDFA depicted in Figure 8. In addition to modeling what is present in the JDFA, it also models mechanisms implicit in the machinery, such as message ordering. Also, the JDFA incorporates much which is beyond the underlying formal DFA model, and thus cannot be subjected to verification. The CPN captures all of the same mechanisms within the formal model.

11.1 Register Implemented in CPN-ML

We further illustrate this example by examining a full, executable CPN implementation of the complete Register conversation. Register as given in [40] consists of an initial ‘register’ with no positive acknowledgment, but a possible ‘error’ or ‘sorry’ reply. This registration may then be followed by an unacknowledged ‘unregister’, also subject to a possible ‘error’ or ‘sorry’ response. This Register conversation (Figure 12) has been extracted from a working CPN model of a multi-agent scenario, implemented in CPN-ML, using the Design/CPN modeling tool. The model, a six agents scenario involving manufacturing integration, uses a separate, identical instance of the register conversation, and other KQML conversations, for each agent. They serve as sub-components to the agent models, which communicate via a modeled network. The declarations (given in Figure 11) have been restricted to only those elements required for the register conversation itself. The diagram is taken directly from Design/CPN. The full model uses concepts for building hierarchical CPNs, such as place replication and the use of sub-nets, which are beyond the scope of this paper. The interested reader is encouraged to refer to [34, 35, 36].

The declarations specify a message format **MES**, a six-tuple of performative, sender and receiver names, message IDs, and content. For simplicity, performative and agent names in the scenario are enumerated, and IDs are integers. For the content, we have constructed a special **Predicate** type, which will allow us to represent content in KIF-like expressions. The **Reg** type is used for registry entries, and encodes the name and address of the registrant, the name of the registrar, and the ID of the registration message. Finally, the **Signature** type is used to bind the names of the sender and receiver with the ID for a particular message.

The model is somewhat more complex than our informal sketch (Figure 10) for several reasons, which will become clear as we look more closely at its operation. For one thing, it is intended to model multiple concurrent conversations, and so must be able to differentiate among them. Also, it implements the complete registration operation, rather than simply modeling the message flow. All messages are initially presented in the **In** place, and once processed by each transition are moved to the **Out** place. Messages from the **Out** place are moved by the agent to the model network, through which they find their way to the **In** place of the same conversation in the target agent. The first transition (**T4**) accepts the message for the conversation, based on the performative ‘register’, and makes it available to the **T1** transition. **T1**, accepts the message if correct, and places a copy in the **Out** place. It also places an entry in the registry (**Reg**), and a message signature in **Sig1**. This signature will be used to make sure that replies to that message have the appropriate values in the sender and receiver fields. Message ID is included in the signature in order to allow the net to model multiple Register conversations concurrently. Note that because KQML does not provide for an acknowledgment to a ‘register’ message, the registration is made immediately, and is then retracted later if an ‘error’ or ‘sorry’ message is received.

Transition **T2a** will fire if an 'error' or 'sorry' is received in response to the registration. It unceremoniously removes the registration from **Reg**. The message signature constrains the names in the reply message. It is also possible for the initiating agent to send a subsequent 'unregister'; in that case **T2b** will fire (again, contingent on the constraints of the message signature being met), also removing the registration. However, since it is possible for an 'unregister' to be rejected (by an 'error' or 'sorry'), **T2b** archives the registration entry in **Arc**, and constructs a new signature for the possible reply. Such a reply would cause transition **T3** to restore the registration to **Reg**.

```
color Performative = with register | unregister | error | sorry;
color Name = with ANS | Broker | AnyName;
color ID = int;
color Address = with ans | broker | anyAddress;
color PVal = union add:Address + nam:Name;
color PVals = list PVal;
color PName = with address | agentName;
color Predicate = product PName * PVals;
color Content = union pred:Predicate + C;
color MES = product Performative * Name * Name * ID * ID * Content;
color Reg = product Name * Name * Address * ID;
color Signature = product Name * Name * ID;

var c : Content;
var message : MES;
var s, r, anyName, name : Name;
var i, j : ID;
var p : Performative;
var a : Address;
```

Fig. 11. Declarations for the Register Conversation.

12 Example: Negotiation Model

In this section we present a simple negotiation protocol proposed in [10]. The CPN diagram in Figure 13 describes the pair-wise negotiation process in a simple MAS, which consists of two functional agents bargaining for goods. The messages used are based on the FIPA ACL negotiation performative set.

The diagram depicts three places places: **Inactive**, **Waiting**, and **Thinking**, which reflect the states of the agents during a negotiation process[3]; we will use

[3] It is not always the case with such a model that specific nodes correspond to states of the system or particular agents. More often the state of the system is described by the combined state of all places.

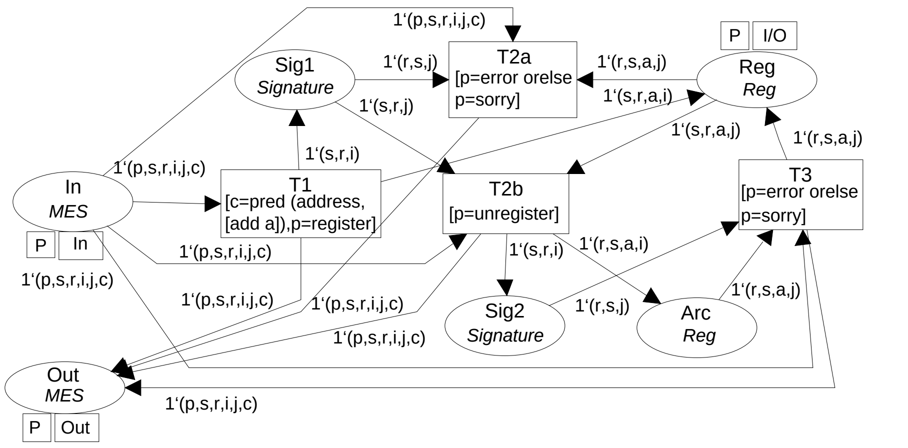

Fig. 12. KQML Register.

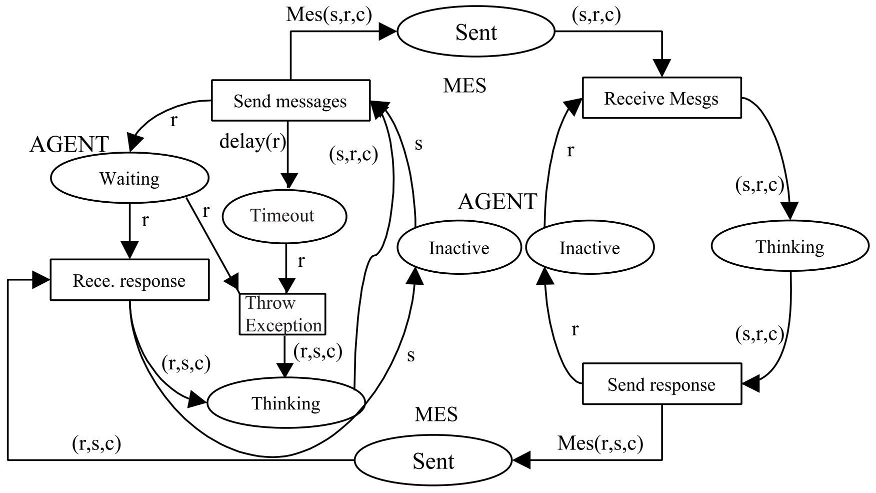

Fig. 13. Pair-wise negotiation process for a MAS constituted of two functional agents.

the terms state and place interchangeably. Both agents in this simple MAS have similar architecture, differing primarily in the number of places/states. This difference arises from the roles they play in the negotiation process. The agent that begins the negotiation, called the *buyer* agent, which is shown on the left side of the diagram, has the responsibility of handling message failures. For this, it has an extra 'wait' state (**Waiting**), and timing machinery not present in the other agent, *seller*. For simplicity, some constraints have been omitted from this diagram; for example, constraints on message types, as depicted in the previous examples.

In this system, both agents are initially waiting in the **Inactive** places. The buyer initiates the negotiation process by sending a call for proposals ('CFP') to some seller, and its state changes from **Inactive** to **Waiting**. The buyer is waiting for a response ('proposal', 'accept-proposal', 'reject-proposal' or 'terminate'). On receipt, its state changes from **Inactive** to **Thinking**, at which point it must determine how it should reply. Once it replies, completing the cycle, it returns to the **Inactive** state. We have inserted a rudimentary timeout mechanism which uses a delay function to name messages which have likely failed in the **Timeout** place. This enables the exception action (**Throw Exception**) to stop the buyer from waiting, and forward information about this exception to the agent in the **Thinking** state. Timing can be handled in a number of ways in implementation, including delays (as above), the introduction of timer-based interrupt messages, or the use of timestamps. CPN-ML supports the modeling of time-dependent interactions through the later approach.

Note that this protocol models concurrent pairwise interactions between a buyer and any number of sellers.

13 Verification

The ability to verify the properties of a specification is one of the important benefits of applying formal methods. These benefits can be derived in two ways:

- Verification of the conversation policies or protocols directly, and
- Verification of agents/MASs that are based on such protocols.

We will first consider the range of properties amenable to analysis, and then discuss their value in the two contexts described. The focus will be on the methods provided by Design/CPN and associated tools.

In addition to 'proof by execution', CPNs can be checked for a variety of properties. This is done by way of an Occurrence Graph (OG) [16]. Each node in an OG consists of a possible marking for the net. If another marking (B) can be reached by the firing of a transition, the graph contains a directed arc from the node representing the initial marking to B. All nodes in an OG are therefore derived from some initial marking of the net.

The properties subject to verification are:

1. Reachability Properties: This relates to whether or not the marking denoted by node B is reachable by some sequence of transition firings from node A.
2. Boundedness Properties: The upper or lower bound on the contents of place X in the net, over all possible markings. This can be the cardinality of the multiset at node X, or the greatest or least multiset itself.
3. Home Properties: The marking or set of markings which are reachable from all other markings in the OG define a homespace. One can verify that a marking or set of markings constitutes a homespace, or determine whether or not a home marking exits, and what the minimal such marking is.

4. Liveness Properties: A marking from which no further markings can be derived is 'dead'. Liveness, then, relates to the possible progressions from a given node in the OG. One can verify that a marking is dead, or list dead markings in the OG.
5. Fairness Properties: Relates to the degree to which certain transition instances (TI) will be allowed with respect to other TIs.

Many of these properties have different value depending on whether we are regarding a CP or a MAS, and also on the complexity of the net. CPs describe/operate on a message stream, which in most cases is finite; they are themselves static. One can imagine analyzing a CP in the context of (1) a single message stream, or (2) in the presence of a generator for all or many representative streams. In that sense, we may be interested in boundedness or home properties, and possibly reachability or fairness, but not liveness. On the other hand, liveness and fairness will often be more important in the analysis of a system as a whole.

For example, consider a simple CP such as Register. Given some sequence of messages, we might be interested in reachability; does this initial marking result in the correct behavior (e.g. a registration being placed). If we were to construct a net which generates a broad range of messages or message sequences, we could combine this with our Register conversation, and analyze Register's behavior with respect to this set of messages streams. Home properties would be useful here; we could designate certain terminal markings (e.g accept, reject), and then, designating them as members of a home space, determine whether or not all test sequences resulted in one of the two acceptable markings.

In a MAS, we are concerned with more dynamic properties of the system, which is assumed to be engaged in some self-sustaining activity. Liveness tests will assure us that the system has not entered a state in which no further activity can occur. Fairness assures us that no elements of the system which are able to act will remain inactive indefinitely. The identification of a home space could allow us to determine that a MAS will successfully achieve its goal.

It is possible to verify properties even for very large and complex nets. The version of Design/CPN used in this research supports the computation and analysis of OGs of 20,000 - 200,000 nodes and 50,000 to 2,000,000 arcs.

14 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.

The use of conversation policies greatly facilitates the development of systems of interacting agents. While FSMs have proven their value over time in this endeavor, we feel that inherent limitations necessitate the use of a model supporting concurrency for the more complex interactions now arising. CPNs provide many of the benefits of FSMs, while allowing greater expression and concurrency. Using the Jackal agent development platform, we hope to demonstrate the value of CPNs as the underlying model for a protocol specification language, Protolingua.

References

1. Tilak Agerwala. Putting Petri Nets to work. *Computer*, pages 85–94, December 1979.
2. Yariv Aridor and Danny B. Lange. Agent design patterns: Elements of agent application design. In *Proceedings of the Second International Conference on Autonomous Agents (Agents '98)*, Minneapolis, May 1998. ACM Press.
3. Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multiagent systems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 17–25, San Francisco, CA, 1995. MIT Press.
4. R. J. Bayardo, Jr., W. Bohrer, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-based semantic integration of information in open and dynamic systems. In *Proceedings of (SigMod 97)*, 1997.
5. J. Bermudez. Advanced planning and scheduling systems: Just a fad or a breakthrough in manufacturing and supply chain management? Technical report, Advanced Manufacturing Research, Boston, Massachusetts, December 1996.
6. J. Billington, M. Farrington, and B. B. Du. Modelling and analysis of multi-agent communication protocols using CP-nets. In *Proceedings of the third Biennial Engineering Mathematics and Applications Conference (EMAC'98)*, pages 119–122, Adelaide, Australia, July 1998.
7. Jeffrey M. Bradshaw. KAoS: An open agent architecture supporting reuse, interoperability, and extensibility. In *Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1996.
8. Jeffrey M. Bradshaw, Stuart Dutfield, Pete Benoit, and John D. Woolley. KAoS: Toward an industrial-strength open agent architecture. In Jeffrey M. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1998.
9. Deepika Chauhan. JAFMAS: A Java-based agent framework for multiagent systems development and implementation. Master's thesis, ECECS Department, University of Cincinnati, 1997.
10. Ye Chen, Yun Peng, Tim Finin, Yannis Labrou, and Scott Cost. A negotiation-based multi-agent system for supply chain management. In *Working Notes of the Agents '99 Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain.*, Seattle, WA, April 1999.
11. Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with place capacities, test arcs and inhibitor arcs. Technical Report DAIMI PB-398, Computer Science Department, Aarhus University, Aarhus C, Denmark, May 1992.
12. B. Chu, W. J. Tolone, R. Wilhelm, M. Hegedus, J. Fesko, T. Finin, Y. Peng, C. Jones, J. Long, M. Matthes, J. Mayfield, J. Shimp, and S. Su. 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*, Boston, MA, 1996.

13. R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Modeling agent conversations with colored petri nets. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 59–66, Seattle, Washington, May 1999.
14. R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield, and Akram Boughannam. Jackal: A Java-based tool for agent development. In Jeremy Baxter and Chairs Brian Logan, editors, *Working Notes of the Workshop on Tools for Developing Agents, AAAI '98*, number WS-98-10 in AAAI Technical Reports, pages 73–82, Minneapolis, Minnesota, July 1998. AAAI, AAAI Press.
15. R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Tim Finin, Ethan Miller, and Charles Nicholas. TKQML: A scripting tool for building agents. In Michael Wooldridge, Munindar Singh, and Anand Rao, editors, *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 336–340. Springer-Verlag, Berlin, 1997.
16. Department of Computer Science, University of Aarhus, Denmark. *Design/CPN Occurrence Graph Manual*, version 3.0 edition, 1996.
17. Ian Dickenson. Agent standards. Technical report, Foundation for Intelligent Physical Agents, October 1997.
18. P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work: Sharing Perspectives (CSCW '92)*, pages 107–114, Toronto, November 1992.
19. Renée Elio and Afsaneh Haddadi. On abstract task models and conversation policies. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 89–98, Seattle, Washington, May 1999.
20. A. El Fallah-Seghrouchni, S. Haddad, and H. Mazouzi. A formal study of interactions in multi-agent systems. In *Proceedins of ISCA International Conference in Computer and their Applications (CATA '99)*, April 1999.
21. A. El Fallah-Seghrouchni and S. Haddad H. Mazouzi. Etude des interactions basée sur l'observation reépartie dans un systéme multi-agents. In Hermés, editor, *Proceedings of JFIADSMA '98*, Nancy, France, November 1998.
22. Amal El Fallah-Seghrouchni and Hamza Mazouzi. A hierarchial model for interactions in multi-agent systems. In *Working Notes of the Workshop on Agent Communication Languages, IJCAI '99*, August 1999.
23. Jaques Ferber. *Les Système Multi-Agents.* InterEditions, 1996.
24. Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeff Bradshaw, editor, *Software Agents.* MIT Press, 1997.
25. FIPA. FIPA 97 specification part 2: Agent communication language. Technical report, FIPA - Foundation for Intelligent Physical Agents, October 1997.
26. H. Robert Frost. Java Agent Template. Online Documentation: http://cdr.stanford.edu/ABE/JavaAgent.html, 1999.
27. Alan Galan and Albert Baker. Multi-agent communications in JAFMAS. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 67–70, Seattle, Washington, May 1999.
28. General Magic. *Introduction to the Odyssey API*, 1998.
29. T. Holvoet. Agents and petri nets. *The Petri Net Newsletter*, (49):3–8, 1995.

30. T. Holvoet and P. Verbaeten. Synchronization specifications for agents with net-based behavior descriptions. In *Proceedings of CESA '96 IMACS Conference, Symposium on Discrete Events and Manufacturing Systems*, pages 613–618, Lille, France, July 1996.
31. Tom Holvoet and Thilo Keilmann. Behavior specification of active objects in open generative communication environments. In Hesham El-Rewini and Yale N. Patt, editors, *Proceedings of the HICSS-30 Conference, Track on Coordination Models, Languages and Systems*, pages 349–358. IEEE Computer Society Press, January, 7–10 1997.
32. Tom Holvoet and Pierre Verbaeten. Using petri nets for specifying active objects and generative communication. In G. Agha and F. DeCindio, editors, *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
33. N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, M. E. Wiegand, C. Voudouris, J. L. Alty, T. Miah, and E. H. Mamdani. Adept: Managing business processes using intelligent agents. In *Proceedings of BCS Expert Systems Conference (ISP Track)*, Cambridge, UK, 1996.
34. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 1, Basic Concepts of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.
35. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 2, Analysis Methods of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1994.
36. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume Volume 3, Practical Use of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.
37. Thilo Kielmann. Designing a coordination model for open systems. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models: Proceedings of COORDINATION '96*, number 1061 in Lecture Notes in Computer Science, pages 267–284. Springer, Cesena, Italy, 1996.
38. K. Kuwabara. AgenTalk: Coordination protocol description for multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS '95)*. AAAI/MIT Press, 1995.
39. Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '95)*, pages 460–465, 1995.
40. Yannis Labrou. *Semantics for an Agent Communication Language*. PhD thesis, University of Maryland Baltimore County, 1996.
41. Yannis Labrou and Tim Finin. Comments on the specification for FIPA '97 AGENT COMMUNICATION LANGUAGE. Internet document, 1997.
42. Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical report, UMBC, 1997.
43. Yannis Labrou and Tim Finin. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*. Morgan Kaufman, August 1997.
44. C. Lakos and Søren Christensen. A general systematic approach to arc extensions for coloured petri nets. Technical Report R93-7, Department of Computer Science, University of Tasmania, Hobart, Tasmania, August 1993.

45. D. B. Lange and M. Oshima. *Programming and Deploying Agents with Java.* Addison-Wesley, Reading, MA, 1998.
46. Fuhua Lin, Douglas H. Norrie, Weiming Shen, and Rob Kremer. Schema-based approach to specifying conversation policies. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies, Third International Conference on Autonomous Agents*, pages 71–78, Seattle, Washington, May 1999.
47. D. L. Martin, A. J. Cheyer, and D. B. Moran. Building distributed software systems with open agent architecture. In *Proceedings of the Third Internations Conference on Practical Applications of Intelligent Agents*, London, 1998.
48. Francisco Martin, Enric Plaza, and Juan Rodríguez-Aguilar. Conversation protocols: Modeling and implementing conversations in agent-based systems. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 49–58, Seattle, Washington, May 1999.
49. M. Merz and W. Lamersdorf. Agents, services, and electronic markets: How do they integrate? In *Proceedings of the IFIP/IEEE International Conference on Distributed Platforms*, Dresden, Germany, 1996.
50. P. Mockapetris. RFC 1034: Domain names - concepts and facilities, 1987.
51. P. Mockapetris. RFC 1035: Domain names - implementation and specification, 1987.
52. Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured petri nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN '97)*, number 1248 in Lecture Notes in Computer Science, pages 82–101, Toulouse, France, June 1997.
53. Scott Moore. On conversation policies and the need for exceptions. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 19–28, Seattle, Washington, May 1999.
54. M. H. Nodine and A. Unruh. Facilitating open communication in agent systems: the InfoSleuth infrastructure. In Michael Wooldridge, Munindar Singh, and Anand Rao, editors, *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 281–295. Springer-Verlag, Berlin, 1997.
55. M. H. Nodine and A. Unruh. Facilitating open communication in agent systems: The InfoSleuth infrastructure. Technical Report MCC-INSL-056-97, MCC, April 1997.
56. Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. ZEUS: A toolkit for builing distributed multi-agent systems. *Applied Artificial Intelligence*, 13(1):129–186, 1999.
57. H. Van Dyke Parunak. 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)*, 1996.
58. Y. Peng, T. Finin, Y. Labrou, R. S. Cost, B. Chu, J. Long, W. J. Tolone, and A. Boughannam. An agent-based approach for manufacturing integration - the CIIMPLEX experience. *International Journal of Applied Artificial Intelligence*, 13(1–2):39–64, 1999.
59. Charles Petrie. JATLite. Online Documentation: http://java.stanford.edu/, 1998.
60. Jeremy Pitt and Abe Mamdani. Communication protocols in multi-agent systems. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 39–48, Seattle, Washington, May 1999.
61. M. Purvis and S. Cranefield. Agent modelling with petri nets. In *Proceedings of the CESA '96 (Computational Engineering in Systems Applications) Symposium*

on Discrete Events and Manufacturing Systems, pages 602–607, Lille, France, July 1996. IMACS, IEEE-SMC.

62. Inc. Reticular Systems. *AgentBuilder: An Integrated Toolkit for Constructing Intelligent Software Agents*, revision 1.3 edition, February 1999.
63. Y. Shoham. AGENT-0: A simple agent language and its interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 704–709, Anaheim, California, 1991.
64. Yoav Shoham. Agent–oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
65. S. R. Thomas. The PLACA agent programming language. In M. J. Wooldridge and N. R. Jennings, editors, *Proceedings of the ECAI '94 Workshop on Agent Theories, Architectures and Languages: Intelligent Agents I*, Lecture Notes in Artificial Intelligence, pages 355–370. Springer-Verlag, Berlin, 1994.
66. T. Vollmann, W. Berry, and D. Whybark. *Manufacturing Planning and Control Systems*. Irwin, New York, 1992.
67. Thomas Wagner, Brett Benyo, Victor Lesser, and Ping Xuan. Investigating interactions between agent conversations and agent control components. In *Working Notes of the Workshop on Specifying and Implementing Conversation Policies*, pages 79–88, Seattle, Washington, May 1999.
68. James White. Mobile agents. In Jeffery M. Bradshaw, editor, *Software Agents*. MIT Press, 1995.
69. Frank Wienberg. *Multiagentensysteme auf def Basis gefärbter Petri-Netze*. PhD thesis, Universität Hamburg Fachbereich Informatik, 1996.
70. Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley, 1986.
71. Min-Jung Yoo, Walter Merlat, and Jean-Pierre Briot. Modeling and validation of mobile agents on the web. In *Proceedings of the International Conference on Web-Based Modeling & Simulation (SCS Western MultiConference on Computer Simulation)*, San Diego, California, January 1998.

This article was processed using the LaTeX macro package with LLNCS style