A MULTI-AGENT SYSTEM FOR ENTERPRISE INTEGRATION

Y. Peng[1] T. Finin[1] Y. Labrou[1] B. Chu[2] J. Long[2] W. J. Tolone[2] A. Boughannam[3]

[1]Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, MD 21250

[2]Department of Computer Science
University of North Carolina
Charlotte, NC 28223

[3]IBM Corporation
Boca Raton, FL 33431

ABSTRACT

The production management system used by most manufacturers today is comprised of disconnected planning and execution processes, and lacks the support for interoperability for enterprise wide integration. This situation often prevents the manufacturer from fully exploring market opportunities in a timely fashion. To address this problem, we propose an agent-based framework for intelligent enterprise integration. A set of agents with specialized expertise can be quickly assembled to help with the gathering of relevant information and knowledge, to cooperate with each other and with other management systems and human managers and analysts to arrive at timely decisions in dealing with various enterprise scenarios. The proposed multi-agent system, including its theoretical foundation, architecture, and implementation, are presented. The work of this system is demonstrated through an integration scenario involving real management software systems.

1. INTRODUCTION

The production management system used by most of today's manufacturers consists of a set of separate software applications, each for a different part of planning, scheduling, and execution processes [16]. For example, Capacity Analysis (CA) software determines a Master Production Schedule that sets long-term production targets. Enterprise Resource Planning (ERP) software generates material and resource plans. Scheduling software determines the sequence in which shop floor resources (people, machines, material, etc.) are used in producing different products. Manufacturing Execution System (MES) tracks real-time status of work in progress, enforces routing integrity, and reports labor/material claims.

Most of these business applications are legacy systems developed over years. Although each of these software systems performs well for its designated tasks, they are not equipped to handle business scenarios across domains of several applications. For example, consider the scenario of updating a purchase order when the shipment date on a purchased part is changed. This event may cause one of the following possible actions: (a) the manufacturing plan is still feasible, no action is required; (b) order substitute parts; (c) reschedule; or, (d) reallocate available material. To determine which of these actions to take, different applications and possibly human decision-makers must be involved. Examples of other similar scenarios include a favorite customer's request to move one of its orders ahead, a machine breakdown being reported by MES, a crucial operation having its rate decreased from the normal rate, to mention just a few. Timely solutions to these scenarios are crucial to agile manufacturing. Unfortunately, current production management systems cannot support integrated solutions to such scenarios. This is because existing management systems provide little

interoperability among isolated individual application softwares, and there is no software to coordinate information and knowledge gathering and decision-making at the enterprise level [1,15].

With matching funds from the National Institute of Standards and Technology, the *Consortium for Intelligent Integrated Manufacturing Planning-Execution* (CIIMPLEX), consisting of several private companies and universities, was formed to address this problem. The aim of the consortium is to develop technologies for intelligent enterprise-wide integration of planning and execution for manufacturing [3]. One approach to this problem might be to rewrite all application softwares into a monolithic integrated planning-execution system capable of handling all foreseeable scenarios. This approach is unfeasible because (a) rewriting legacy application systems is formidably expensive; (b) a monolithic system is too rigid to expand to cover unforeseen new scenarios once the system is put into use; and (c) unlike distributed systems, a monolithic system is difficult to develop, to test, and to maintain. Instead, CIIMPLEX adopts as one of its key technologies the approach of intelligent software agents, a rapidly emerging technology of software systems in networked distributed environment. Unlike traditional software programs, software agents are programs that help people solve problems by collaborating with other software agents and other resources in the network [2,4,13]. Working with a software agent is like working with a travel agent who is aware of customer's preferences and can collaborate with other agents and systems to obtain additional information and services in fulfilling customer needs. A collection of software agents can be designed to perform data collection and analysis of plans and schedules at different levels. They keep constant vigil against mismatches among these plans and schedules at different levels of abstraction and time horizons. Scenario coordination agents can be designed to resolve the conflicts either by themselves or in coordination with human managers and analysts. Personal assistant agents can be designed to assist human managers/analysts.

The rest of this paper is organized as follows. In the next section, we briefly describe the theoretical foundations and core technologies of software agents that are relevant to the task of manufacturing integration, with an emphasis on agent communication. Sections 3 and 4 are the core of this paper, where we first present the proposed agent system's architecture, and then its function through an example scenario. We conclude the paper with discussions of ongoing work to expand the agent system, along with the directions for future research in Section 5.

2. MULTI-AGENT SYSTEM AND AGENT COLLABORATION

The computing paradigm of multi-agent systems (MAS) has its origin in both distributed artificial intelligence (DAI) and object-oriented distributed systems. There is no consensus on the definition of software agents or of agency, and some people go so far as to suggest that any piece of software or object that can perform a specific given task is an agent. However, the prevailing opinion is that an agent may exhibit three important general characteristics: *autonomy*, *adaptation,* and *cooperation* [9,12]. By "autonomy" we mean that agents have their own agenda of goals and exhibit goal-directed behavior. They are not simply reactive, but can be pro-active and take initiatives as they deem appropriate. In this sense, agent systems can be viewed as a generalization of the client-server model in that each agent can be both a client and a server and can provide and request services to and from others. Adaptation implies that agents are capable of adapting to the environment, which includes other agents and human users, and can learn from the experience in order to improve themselves in a changing environment. Cooperation and coordination between agents is probably the most important feature of multi-agent systems [12]. Unlike those stand-alone agents, agents in a multi-agent system collaborate with each other to achieve common goals. In other words, these agents *share* information, knowledge, and tasks among themselves. The intelligence of MAS is not only reflected by the expertise of individual agents but also exhibited by the emerged collective behavior beyond individual agents. From a software engineering point of view, the approach of

MAS is also proven to be an effective way to develop large distributed systems. Since agents are relatively independent pieces of software interacting with each other only through message-based inter-agent communication, system development, integration, and maintenance become easier and less costly [10]. For instance, it is easy to add new agents into the agent system when needed. Also, the modification of legacy applications can be kept minimum when they are to be brought into the system. Aside from adding communication capabilities to a legacy application, nothing else is required to change.

Cooperation and coordination of agents in a MAS requires agents to be able to understand each other and to communicate effectively with each other. The infrastructure that supports agent cooperation in a multi-agent system is thus seen to include at least the following key components:

- A common agent communication language (ACL) and protocol,
- A common format for the content of communication, and
- A shared ontology.

In CIIMPLEX we take the *Knowledge-Sharing-Effort* (KSE) approach toward achieving the infrastructure needed for agent cooperation. KSE, sponsored by the Advanced Research Project Agency (ARPA), the Air Force Office of Scientific Research (AFOFR), the Corporation for National Research Initiative (NRI), and the National Science Foundation (NSF), is an initiative to develop technical infrastructure to support knowledge sharing among systems [14]. Three technologies developed from KSE are adopted. They are (a) Knowledge Query Manipulation Language (KQML) as a communication language and protocol, (b) Knowledge Interchange Format (KIF) as the format of the communication content, and (c) the concept of a shared ontology. In what follows we briefly describe the three components and justify their selections in the context of manufacturing integration environment.

2.1. KQML

KQML is based on speech act theory in which inter agent communication is thought of as similar to "conversations" between humans [6,7,14]. KQML considers that each message not only contains the content but also the intention the sender has for that content. To see the different intentions a sender may associate with different messages, consider the example that Agent *A* sends the following statement as the content of a message to Agent *B*:

"the processing rate of operation 1 at machine X is greater than 5."

Agent *A*, in different circumstances, may have different intentions about this statement. Agent *A* may simply

- *tell* *B* that this statement is true in its own database; or
- *ask if* this statement is true in *B*'s database; or
- *ask to obtain all records* in *B*'s database in which this statement is true; or
- ask *B* to *monitor* the change of the processing rate and report back whenever this statement becomes true; or
- request that *B make this statement true* in *B*'s database by, for example, adjusting the existing schedule or putting more machines or labor into the given operation.

KQML provides a formal specification for representing the intentions of messages through a set of pre-defined *performatives* used in the messages. There are about three dozen performatives defined in the current specification of KQML [6]. A particular agent system generally implements only a subset of these defined performatives and may introduce additional performatives of its own, as long as the new ones are defined in the same format and spirit of the KQML specification. The few most commonly used performatives that are particularly relevant to our agent system are briefly listed below, where S and R stand for the Sender and the Receiver of a message, respectively.

- *ask-one*: S wants one of R's answers to a question.
- *ask-if*: S wants to know if a sentence is true in R's knowledge base.
- *advertise*: S is particularly suited to processing a performative.
- *subscribe*: S wants to receive updates to R's response to a performative.
- *recommend-one*: S wants the name of an agent who can respond to a performative.
- *deny*: the embedded performative does not apply to S (any more).
- *tell*: the sentence in the message is true in S's knowledge base.
- *reply*: S communicates an expected reply to R.
- *sorry*: S cannot provide a more informed reply (i.e., S cannot answer the question asked in an earlier message from R).
- *error*: S considers R's earlier message to be malformed.

KQML does not impose any particular common format and interpretation on the content of all messages. In other words, the content of a message is opaque to KQML. The content language for the message body is left for the individual agents to decide. KQML allows an agent to specify in a message the language and ontology it uses for the content of a given message. Not only different agents in a system can have different content language, but different messages from the same agent can also use different content languages. Commonly used content languages include English, Lisp, Prolog, KIF (Knowledge Interface Format), SQL, etc.

A KQML message is thus divided into three layers: the content layer, the message layer, and the communication layer. The content layer bears the actual content of the message in a language chosen by the sending agent. The communication layer encodes a set of features to the message to describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication. The message layer encodes the message, including its intention (by a chosen performative), the content language used, and the ontology. The syntax of KQML messages is based on a balanced parenthesis list. The first element of this list is the performative; the remaining elements are the arguments of the performative, as key word/value pairs. The following is an example of an actual KQML message sent by agent "joe" to agent "stock-server", inquiring about the price of a share of IBM stock:

```
(ask-one
        :language       KIF
        :content        (Price IBM ?x)
        :sender         joe
        :receiver       stock-server
        :reply-with     zxcasd
)
```

In the message, the performative *ask-one* indicates that this is a query type of message. KIF is specified as the content language. The actual content is a KIF predicate, named *Price*, of two arguments: the name of the stock (instantiated to IBM) and the price of the stock (a variable, as indicated by the quotation marker preceding x). The string *zxcasd* is a unique machine generated reply id. In essence, by this message agent *joe* asks one answer from agent *stock-server* of the share price of IBM stock. Any reply from *stock-server* to *joe* concerning this query should carry the reply id of *zxcasd*. In due time, agent *stock-server* might send *joe* the following KQML message if or when it has an answer to the query.

```
(tell
        :sender         stock-server
        :language       KIF
        :content        (Price IBM 89)
        :receiver       joe
```

```
            :in-reply-to   zxcasd
    )
```

The second argument of the predicate Price in the reply message is instantiated to 89, indicating that agent *stock-server* believes (or it is true in *stock-server*'s knowledge base) that the price of IBM stock is 89. More complicated queries can be conveyed by other KQML performatives. For instance, *joe* might want *stock-server* to send the price of IBM stock whenever the price is updated. This can be done by a nested message, starting with the performative *subscribe*[1].

```
    (subscribe
            :language          KQML
            :content           (ask-one  :language KIF   :content (Price IBM ?x))
            :sender    joe
            :receiver          stock-server
            :reply-with        zxcasd
    )
```

2.2. **KIF**

Although KQML allows agents to choose their own content language, it is beneficial for all agents within one MAS to exchange most if not all of their messages in a single neutral format. One obvious advantage of adopting a common message format is efficiency. Instead of many-to-many format conversion, each agent only needs to convert messages between its own internal representation and the common format. KIF (Knowledge Interchange Format), due to its rich expressive power and simple syntax, is probably the most widely used neutral message format for agent communication.

KIF is a prefix version of First Order Predicates Calculus (FOPC) with extensions to support non-monotonic reasoning and definitions [8,14]. The language description includes both specifications for its syntax and for its semantics. First and foremost, KIF provides for the expression of simple data, in the form of predicates (or relations, as KIF calls them). For example, the sentence shown below encodes a tuple in a personnel database in which the first field gives the predicate or relation name, and the other fields represent employee ID number, department assignment and salary, respectively.

(salary 015-46-3946 widgets 60000)

More complicated information can be expressed in composite sentences through the use of relational and logical connectives. For example,

(> (* (width chip1) (length chip1)) (* (width chip2) (length chip2)))

states that one chip is larger than another, and

(=> (and (real-number ?x) (even-number ?n)) (> (expt ?x ?n) 0))

encodes a rule that the number obtained by raising any real number ?x to an even power ?n is positive. In KIF syntax, any symbol preceded by the question mark, as ?x and ?n in the last sentence, is taken to be a variable.

Besides FOPC expressions of facts and knowledge, KIF also supports extra-logical expressions such as those for the encoding of knowledge about knowledge and of procedures.

2.3. **Shared Ontology**

Sharing the content of formally represented knowledge requires more than a formalism (such as KIF) and a communication language (such as KQML). Individual agents, as autonomous entities

---

[1] Since this message has a KQML message as its content, its content language is specified as KQML.

specialized for some particular aspects of problem-solving in a MAS, may have different models of the world in which objects, classes and properties of objects of the world may be conceptualized differently. For example, the same object may be named differently ("machine-id" and "machine-name" for machine identification in databases of two agents). The same term may have different definitions ("salary-rate," referring to hourly rate in one agent and annual rate in another). Also, different taxonomies may be conceptualized from different perspectives by individual agents.

Therefore, to ensure correct mutual understanding of the exchanged messages, agents must also agree on the model of the world, at least the part of the world about which they are exchanging information with each other. In the terminology of the agent community, agents must share a common ontology [14]. An ontology for a domain is a conceptualization of the world (objects, qualities, distinctions and relationships, etc. in that domain). A shared or common ontology refers to an explicit specification of the ontological commitments of a group of agents. Such a specification should be an objective (i.e., interpretable outside of the agents) description of the concepts and relationships that the agents use to interact with each other, with other programs such as legacy business applications, and with humans. A shared ontology can be in the form of a document or a set of machine interpretable specifications.

2.4. Agent Collaboration

With a common communication language, content language, and a shared ontology, agents can communicate with each other in the same manner, in the same syntax, and with the same understanding of the world. In addition, to make agent collaboration more efficient and effective, some service agents are often created in multi-agent systems. One type of a service agent is the *Agent Name Server* (ANS). The ANS serves as the central repository of physical addresses (in the form of the chosen transport mechanism) for all involved agents. It maintains an address table of all registered agents, accessible through the agents' symbolic names. Newly created agents must register themselves with the ANS with their names, physical addresses and possibly other information by sending to the ANS a message with the performative *register*. (As a presumption, every agent in the system must know the physical address of the ANS.) The ANS maps the symbolic name of a registered agent to its physical address when requested by other agents.

Another type of a service agent is the *Facilitator Agent* (FA), who provides additional services to other agents. A simple FA is a *Broker Agent* (BA). The BA serves, to some extent, as a dynamic information hub or switchboard. It registers services offered and requested by individual agents and connects dynamically available services to requests whenever possible. Agents register their available services by sending messages with the performative *advertise*, and request services by sending to the BA messages with brokering performatives such as *recommend-one*. In both cases, the description of the specific service is in the content of the message. In a reply to a *recommend-one* message the BA will send the symbolic name of an agent who has advertised as being able to provide the requested service to the BA, or *sorry* if such request cannot be met by existing advertisers.

One of the key objectives of CIIMPLEX is to establish a monitoring/notification architecture for the enterprise integration. In this architecture, an application will define events it is interested in (e.g. changes in process rates, yield, material due dates) and have programs (agents) to monitor such events. When those events occur, the agents will notify the concerned applications. The monitoring/notification architecture is in sharp contrast to the polling architecture used by many existing systems where reports are periodically generated from production databases. These reports are interpreted either by humans or by other computer programs. The polling model has several major weaknesses:

- Reports cannot be generated in real time.

- Pre-scheduled reports may not catch critical events early enough to make corrective actions.
- Not all information is logged for reports.

A natural way to implement the monitoring/notification architecture is to use Broker agents (BA) to track the agents which can provide monitoring service for a type of event another agent is interest in. The ability of the BA to dynamically link services with requests in real time would greatly increase the flexibility toward manufacturing integration and interoperation.

3. CIIMPLEX AGENT SYSTEM ARCHITECTURE

In this section, we describe the agent system architecture that supports inter agent cooperation in the CIIMPLEX project, with the emphasis on the agent communication infrastructure. Figure 1 below gives the Architecture of the CIIMPLEX enterprise integration with MAS as an integral part.
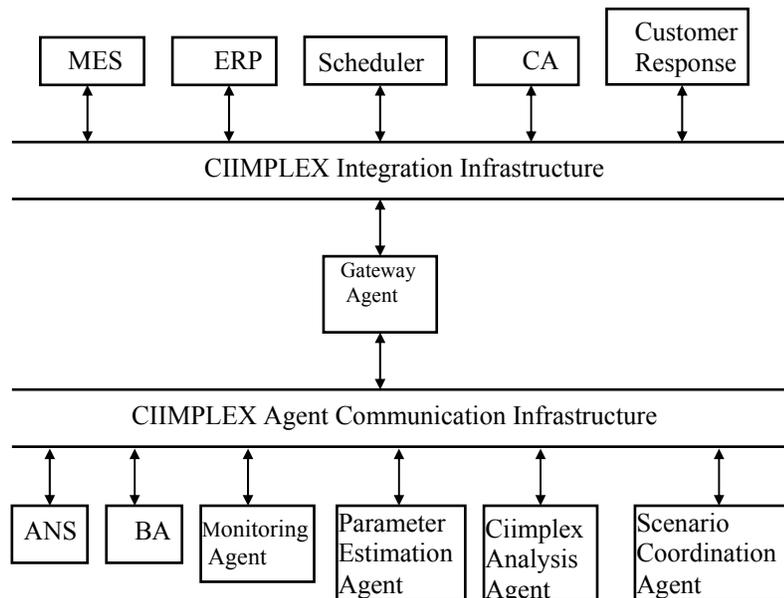

Figure 1. CIIMPLEX Integration Architecture

The CIIMPLEX Integration Infrastructure supports communication between legacy application systems. The CIIMPLEX Agent Communication Infrastructure supports communication between agents. The main reason to separate the two infrastructures is to reduce the necessary modifications to the applications to the minimum.

Besides the service agents ANS and BA, several other types of agents are useful for enterprise integration. For example, data-mining/parameter-estimation agents are needed to collect, aggregate, interpolate and extrapolate the raw transaction data of the low level (shop floor) activities, and to make this aggregated information available for higher level analyses by other agents. Event monitoring agents monitor, detect, and notify about abnormal events that need to be attended. The *CIIMPLEX Analysis Agents* (CAA) evaluate disturbances to the current planned schedule and recommend appropriate actions to address each disturbance. And the *Scenario Coordination Agents* (SCA) assist human decision making for specific business scenarios by providing the relevant context, including filtered information and actions, as well as workflow charts. All these agents speak KQML, and use SKIF, a subset of KIF that supports Horn clause deductive inference, as the content language. TCP/IP is chosen as the low-level transport mechanism for agent-to-agent communication. The shared ontology is an agreement document established by the application vendors and users and other partners in the consortium. The agreement adopts the format of the

Business Object Document (BOD) defined by the *Open Application Group* (OAG). BOD is also used as the message format for communication between applications such as MES and ERP, and between agents and applications. Different transport mechanisms (e.g., MQ Series of IBM and VisualFlow of Envisionit) are under experimentation for communication to and from applications. A special service agent, called the *Gateway Agent* (GA), is created to provide interface between the two infrastructures. GA's functions, among other things, include making connections between the two transport mechanisms (TCP/IP and MQ Series) and transforming messages between the two different formats (KQML and BOD).

The agent system architecture outlined above is supported by the agent communication infrastructure called *Jackal*. As indicated by the name, *JACKAL* is written in *J*ava to support *A*gent *C*ommunication using the *KQML A*gent communication *L*anguage. The decision to select Java as the implementation language was based mainly on its inter-platform portability, its networking facilities, and its support for multi-thread programming. Except for the ability to understand and process KQML messages, Jackal does not impose any restrictions on the internal architecture and representation of individual agents.

## 3.1. <u>Conversation Policies in Jackal</u>

KQML itself only defines the syntax of the language. There is no generally accepted semantics of KQML in the agent community. However, a good, workable semantics is imperative for a coherent conversation which is not merely a collection of messages but rather a proper sequence of messages. The infrastructure must support both syntactic and semantic aspects of the language. Jackal takes a semantic interpretation of KQML from [11] and realizes part of it as a set of conversation policies. In the next two subsections, we first discuss the conversation policies and then the architecture of Jackal.

The conversation policies are procedures which, based on the performatives involved, specify how a conversation is to start, to proceed, and to terminate. For example, a conversation started with an *ask-one* message will terminate as soon as the sender receives a proper response. (Possible replies include an *error* message, indicating that the format of the message is incorrect, a *sorry* message, indicating that the receiver cannot provide an answer to the question, or a *tell* message whose content contains an answer to the given question). A conversation started by a message with performative *subscribe* would have a different policy. When Agent *A* starts such a conversation with Agent *B*, the conversation remains open with *A* keeping listening for new messages from *B* that satisfy the subscription criterion.

Conversation policies chosen for Jackal can be described using a *Deterministic Finite Automata* (DFA) model. A DFA consists of a set of states (nodes) and a set of arcs representing transitions from one state to another. In this model, each conversation starts with a state called **START**, and ends with a state called **STOP.** A conversation moves from one state to another according to the given state transition diagram. The following are the state transition diagrams for some example conversations selected for the CIIMPLEX agent system. In these diagrams, bold upper case character strings are for the states, arcs are for the state transitions, and the lower case strings attached to arcs are for the conditions (inputs) that cause the transitions to occur. For example, in the diagram for *subscribe* conversation, the conversation goes from the initial state, **START,** to a state called **SUBSCRIBE** when an agent issues a message with the *subscribe* performative. The conversation remains at this state until an input (a reply message to the *subscribe* message) is received. If the input is a message with the performative *tell*, the conversation does not change its state (i.e., it loops back to the current state and waits for new messages). If the input is one of the performatives *deny*(*subscribe*), *sorry*, or *error*, then the conversation goes to the state **STOP**, and terminates there.
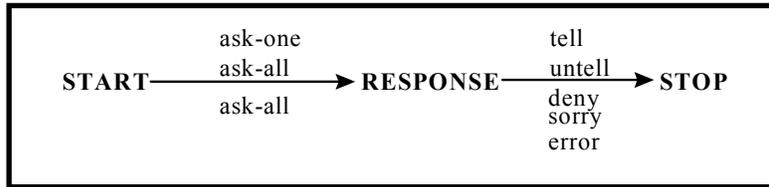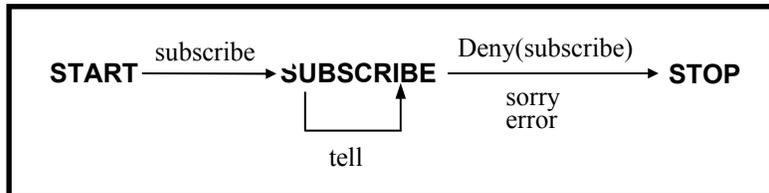
DFA state transition diagram for *ask-* conversations.
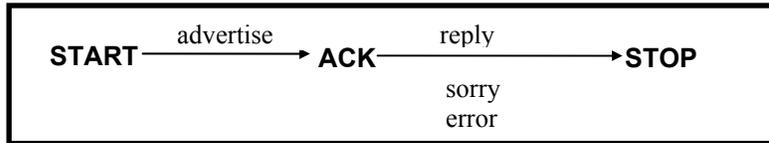
DFA state transition diagram for *query/acknowledge* conversations.
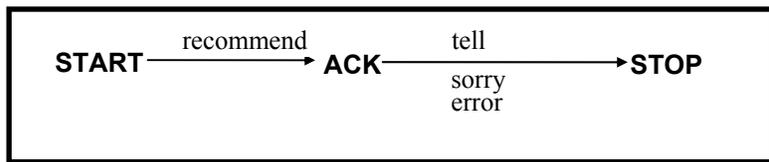
DFA state transition diagram for *recommend-* conversations.

DFA state transition diagram for *subscribe* conversations.

3.2. Jackal architecture

To have a better understanding of Jackal, we shall follow the incoming message, from the incoming transport channel, until it is consumed by the application, and follow the outgoing message, from the application, until it is sent out. The two paths of messages are shown in Figure 2 below. When an incoming message arrives at an agent, it will be picked up by the message-receiving module (a listener) and passed to the KQML message parser. The parser converts the message into a standardized internal format (Java object representation). Next, the message (in internal format) will be passed to the conversation module for conversation resolution. The conversation module will first determine whether the incoming message belongs to an ongoing conversation (e.g., a *tell* message in reply to an *ask-one* previously sent out from this agent). This is done by matching the unique conversation id of the message (the in-reply-to field of the incoming message) with id's of the ongoing conversations (the reply-with field in the message that initiates the conversation). If a match is established, the message is then attached to that conversation. Otherwise, a new instance of an appropriate conversation and its DFA are created, and the message is attached to that new conversation. In either case, when a message is attached to a conversation, the corresponding DFA changes its state; the message, together with the required action (default or user specified function), is passed to the application program.

When an agent wants to send a message (either in response to a previous message or a new message to start a new conversation), the message is passed to the conversation module by calling the

appropriate API. Similarly, in the case of an incoming message, the conversation resolution is conducted to either attach the message to an ongoing conversation or to create a new instance of an appropriate conversation and attach the message to it. The message is then passed to the lower layer where the destination address is resolved, and the message is converted into whatever format appropriate for the underlying transport protocol and sent out.
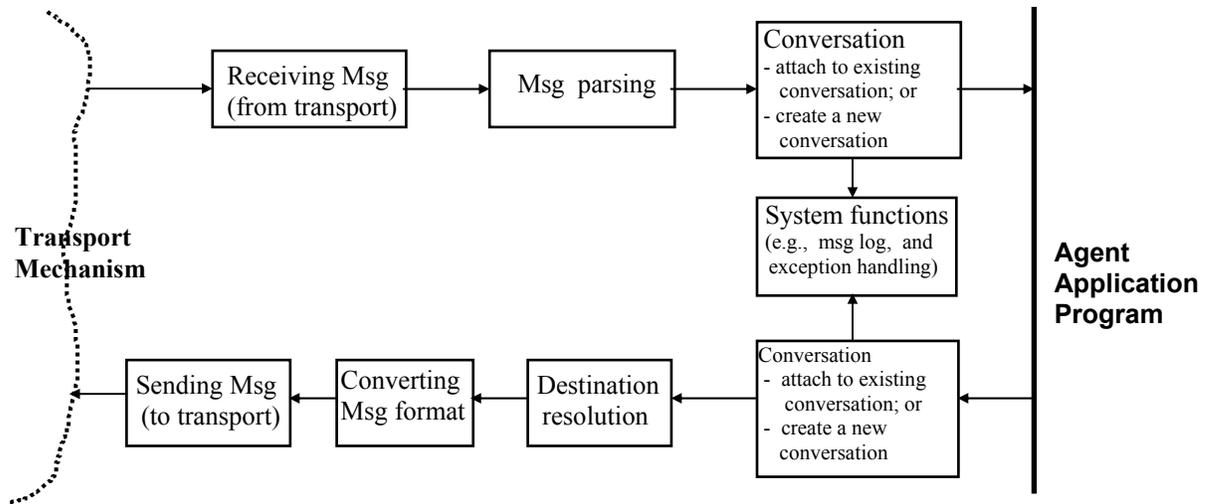

Figure 2. Life of a KQML message in Jackal

The diagram in Figure 3 below outlines the Jackal implementation. Several features distinguish our implementation from others. Jackal supports single, multiple and hierarchical ANS. It is designed in such as way that it can be easily extended to support multiple transport mechanisms in a single MAS. Also, to ensure that every message complies with the conversation policies across the board, the main components (*Message handler, Conversation arena,* and *Distributor*) treat incoming and outgoing message in the same way.
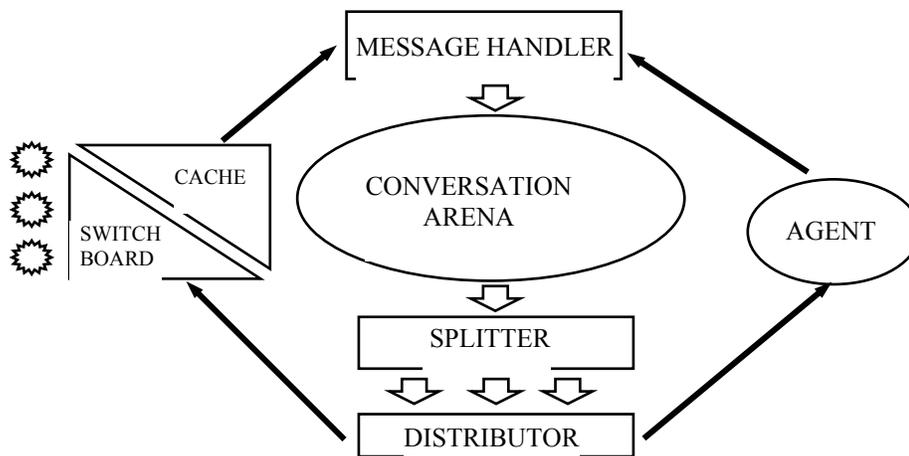

Figure 3. Jackal Architecture

The *Message handler* handles conversation resolution for both incoming and outgoing messages based on matching of the reply-key parameter. DFA state transitions for conversations are performed in the *Conversation arena*. If a conversation policy is violated (i.e., the message does not match any of the state transition conditions of the current state), an error message will be generated. The

Message splitter duplicates messages for multiple destinations (e.g., broadcast or cc messages). The *Distributor* is the communication focal point for agent processes. It maintains two queues: a message queue, where all unconsumed messages are stored, and a request queue where all unmet requests for messages are stored. Message requests are generated in accordance to the policies for ongoing conversations. When a new (incoming or outgoing) message arrives, the distributor will first check the request queue to see if there is a request for this message. If there is such as request, then the request is removed from the queue, and this message is consumed (either sent to the agent application program if the message is incoming, or sent to the switchboard if the message is outgoing). New requests are processed similarly. The *Switchboard* deals with the physical level of the communication, and performs the following functions. It translates the KQML messages between the internal format (Java objects) and the format required by the transport mechanism. It finds the physical addresses for outgoing messages with the help of the ANS and the agent's own address *Cache* module. It supports multiple transport mechanisms, and can switch from one to another according to the mechanism used by the message destination agent. To reduce the network traffic and increase the speed, each agent may maintain an address cache to store the name/address mappings of other agents it has recently contacted. It may also store the transport mechanisms used by these other agents if multiple mechanisms are used in the agent system. Therefore, an agent contacts the ANS only when it cannot find a needed address in its local cache.

To facilitate the construction of agents, the interface between the agent application program and Jackal is kept very simple. The application program interacts with the infrastructure only by sending outgoing messages to the *Message handler* and receiving incoming messages from the *Distributor*. A set of API is provided by Jackal for this purpose. To ease the memory requirement, Jackal allows multiple agents on the same computer to share a single Java interpreter. Each agent maintains its own set of threads (e.g., listening thread, conversation thread, distributing thread) within the single interpreter.

4. AN EXAMPLE

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

4.1. The Scenario

In general, scenarios represent exceptions to the normal or expected activities or events that need special attention. The user scenario selected, called "*process rate change*", occurs when the process time of a given operation 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, and others may be as complicated and expensive as requesting a re-scheduling based on the changed operation rate. The process rate change scenario is depicted in Figure 4 below. Note that two real application programs, namely the FactoryOp (a MES by IBM) and MOOPI (a Finate Scheduler by Berclain), are used in this scenario.

4.2. The agents

To support managing this scenario, we need mechanisms for the following activities.

- Collect information concerning operation completion originated from MES.
- Compute and constantly update the process rate from the collected information.

- Detect and notify the appropriate parties if the current rate change constitutes a significant reduction.
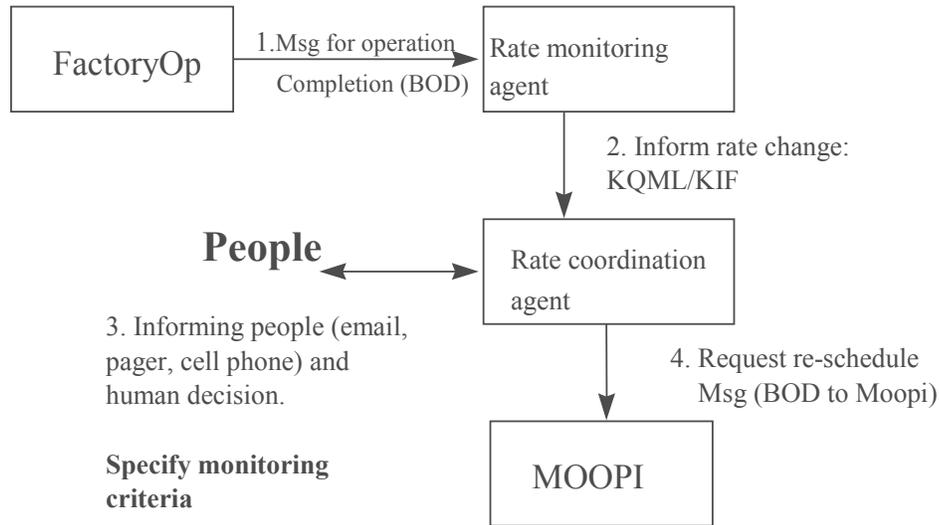- Carry out appropriate actions to handle the rate change.

Figure 4. The "process rate change" scenario

These activities must be coordinated in a coherent manner. A collection of agents is assembled to support the chosen scenario. All of these agents speak KQML, and are supported by Jackal. Besides the three service agents ANS, BA, and GA, the multi-agent system also includes the following special agents.

- The *Process Rate Agent* (PRA) 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 is originated from FactoryOp in the BOD Format, and is converted into KIF format by GA. PRA summarizes and aggregates the continuing stream of operation completion data (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.
- The *Scenario Coordination Agent* (SCA) sets the monitoring criterion, receives the notification for the rate change, and decides, in consultation with human decision-makers, appropriate action(s) to take for the changed rate. One of the actions would be to request MOOPI to reschedule if it is determined that the rate change makes the existing schedule impossible to meet. This request is sent from SCA as a KQML message to GA, where it is converted into the BOD format. Details of the internal logic and the algorithms of the SCA that handle the "rate change" scenario are reported elsewhere.
- 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.
- 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.

4.3. <u>The Predicates</u>

Three SKIF predicates of multiple arguments are defined for processing the process rate change scenario. Their names are OP-COMPLETE, RATE, and RATE-CHANGE.

Predicate OP-COMPLETE contains all relevant information concerning a completed operation such as the machine-id, product-id, operation-id, starting and finishing time of the operation. Each instance of this predicate corresponds to a BOD originating from FactoryOp, and GA is responsible for converting the BOD to this predicate. Predicate RATE contains all relevant information concerning the current operation rate of a particular operation at a particular machine with a particular product. The operation rate is represented by its mean and standard deviation. Instances of Rate predicate are computed and constantly updated by PRA, based on a stream of instances of predicate OP-COMPLETE obtained from GA. Predicate RATE-CHANGE 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. In particular, it contains the operation rate used to compute the current schedule and the new rate. It is the responsibility of the rate SCA to compose an instance of the RATE-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. Additional predicates and more complicated KIF expressions are needed when dealing with more complicated scenarios.

4.4. <u>Agent Collaboration And The Message Flow in The Agent System</u>

Figure 5 below outlines how agents are cooperating 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 also not shown.

Each of these agents needs information from others to perform its designated tasks. Each of them may also have information others need. There is no permanent stationery connection between agents. The broker agent (BA) can be seen to play a crucial role in collaborating agents' information exchange activities. This is done by sending *advertise* and *recommend* messages to BA.

Advertising to BA.

GA advertises that it can provide OP-COMPLETE predicate. It also advertises to be able to accept RATE-CHANGE predicate and forward it to MOOPI as BOD messages. PRA advertises that it has current process rates available for some operations in the form of RATE predicate. The following is an example of *advertise* message from GA.

```
(advertise
        :sender      GA
        :receiver    BA
        :reply-with  null8599132710561
        :content     (subscribe :content (ask-one :content (OP-COMPLETE  ?x1 … ?xn))))
```

Requesting recommendation from BA.

PRA asks BA to recommend an agent that can provide OP-COMPLETE predicate, and will receive the recommendation of GA in a responding *tell* message. Similarly, SCA asks BA to recommend an agent which can provide RATE predicate and receives PRA in response. It also asks BA to recommend an agent which can accept RATE-CHANGE predicate and receives GA in response. The following is an example of *recommend-one* message from PRA.

```
(recommend-one
        :sender         pra
```

```
:receiver        ba
:reply-with      null222222222
:content         (subscribe :content (ask-one :content (OP-COMPLETE ? ? … ? ?))))
```
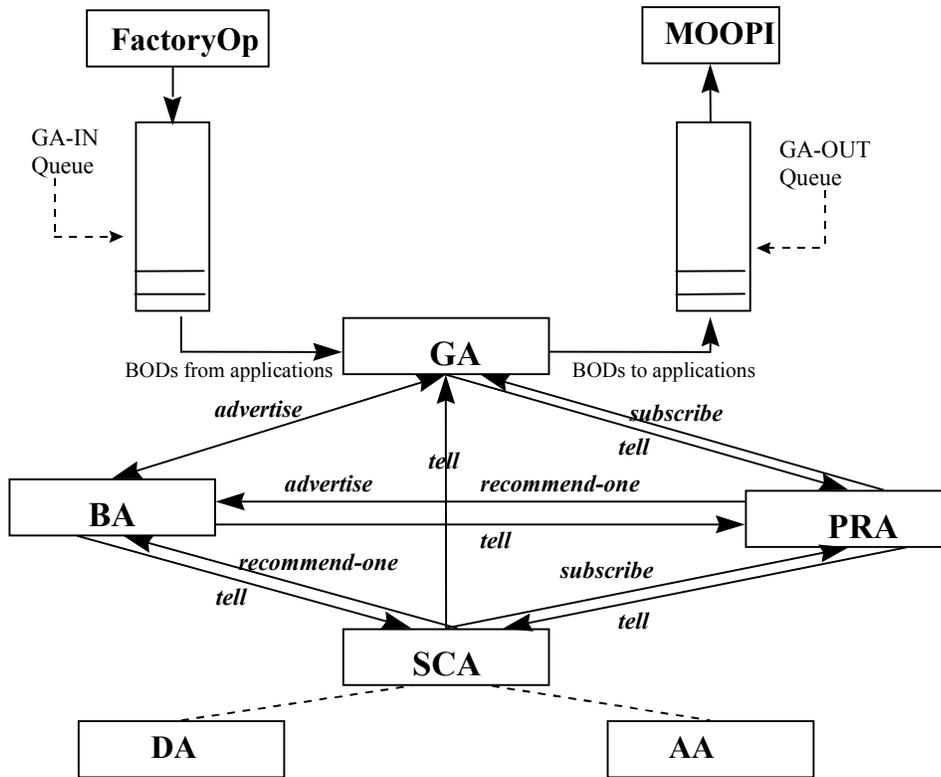

Figure 5. The agent system for "process rate change" scenario

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

```
(tell
        :sender          ba
        :receiver        pra
        :in-reply-to     null222222222
        :reply-with      null333333333
        :content         (ga))
```

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

Monitoring/notification

When SCA knows from BA that PRA has advertised to be able to provide the current rate for certain operation, it may send PRA the following *subscribe* message.

```
(subscribe
        :sender          sca
        :receiver        pra
        :reply-to        null444444444
        :language        KQML
        :content    (ask-one
```

```
:language  SKIF
:content    (and (RATE  … ?mean …) (< ?mean 50))))
```

With this message, SCA tells PRA that it is interested in receiving new instances of RATE predicate whenever the mean value of the new rate is 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.

5. CONCLUSION

In this paper we presented a multi-agent system that is capable of supporting intelligent integration of manufacturing planning and execution. With this approach, a set of software agents with specialized expertise can be quickly assembled to help gather relevant information and knowledge and to cooperate with each other, and with other management systems and human managers and analysts, to arrive at timely decisions in dealing with various enterprise scenarios. This system has been tested successfully with a real manufacturing scenario involving real legacy MES and scheduler.

The work presented here represents only the first step of our effort toward agent-based manufacturing integration. Further research and experiment are needed to extend the current work and to address its shortcoming. Although KQML does not impose much constraint and requirements on the internal structure of agents, it might be beneficial to have a common framework for the agent's internal structure within a single agent system. We are currently considering a lightweight blackboard architecture for such a framework which, among other advantages, may provide flexibility for agent construction, agent component re-usability and plug-and-play. Another research direction under active consideration is to increase the functionality of the broker agent and make it more intelligent. The BA in our current implementation can only conduct brokering activities at the level of predicates. With the help of a machine interpretable common ontology and an inference engine, more intelligent brokering can be developed to work with object hierarchies and to make intelligent choices. Work is also under way to identify more complex enterprise scenarios which require non-trivial interactions with more legacy systems, and their solutions represent significant added values to the manufacturing production management.

ACKNOWLEDGMENT

REFERENCES

1. Bermudez, J. "Advanced Planning and Scheduling Systems: Just a Fad or a Breakthrough in Manufacturing and Supply Chain Management?" *Report on Manufacturing, Advanced Manufacturing Research,* Boston, MA. Dec. 1996.
2. Bradshaw, J., Dutfield, S., Benoit, P. & Woolley, J. "KAoS: Toward An Industrial-Strength Open Agent Architecture." To appear in *Software Agents* Bradshaw, J.M. (Ed), MIT Press.
3. Chu, B., Tolone, W. J., Wilhelm, R., Hegedus, M., Fesko, J., Finin, T., Peng, Y., Jones, C. Long, J., Matthews, M. Mayfied, J., Shimp, J., & Su, S. "Integrating Manufacturing Softwares for Intelligent Planning-Execution: A CIIMPLEX Perspective." In *Plug and Play Software for Agile Manufacturing, Proceedings of SPIE* Vol. 2913. pp. 96-108, Boston, MA, 1996.
4. Compositional Research Group "Caltech Infosheres Project." Available at http://www. infospheres.caltech.edu.

5. Dourish, P. Bellotti, V. "Awareness and Coordination in Shared Workspaces." In *Proceedings ACM 1992 Conference on Computer-Supported Cooperative Work: Sharing Perspectives (CSCW '92*), pp107-114, Toronto, November 1992.

6. Finin, T., Weber J. *et al*. "Draft Specification of the KQML Agent Communication Language." June, 1993. Available at http://www.cs.umbc.edu/kqml/kqmlspec/spec.html.

7. Finin, T., Labrou, Y., & Mayfield, J. "KQML as an agent communication language." In *Software Agents*. Bradshaw, J.M. (Ed.) MIT Press, to appear

8. Genesereth, M. & Fikes, R. et al. "Knowledge Interchange Format, Version 3.0 Reference Manual." Technical Report, Computer Science Department, Stanford University, 1992.

9. Genesereth, M. & Katchpel, S. "Software Agents." *Communication of the ACM*. 37(7): 48-53, 1994

10. Hammer, M. *Beyond Reengineering: How the Process-Centered Organization Is Changing Our Work and Our Lives*. Harpercollins, 1996.

11. Labrou, Y. *Semantics for an Agent Communication Language*. Ph.D. Dissertation, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, August, 1996.

12. Nwana, H. "Software Agents: An Overview." The Knowledge Engineering Review Vol 11 (3), 1996

13. Parunak, V., Baker, A., & Clark, S. "AARIA Agent Architecture: An Example of Requirements-Driven Agent-Based System Design." Available at http://www.aaria.uc.edu

14. Patil, R., Fikes, R., Patel-Schneider, P., McKay, D., Finin, T., Gruber, T., & Neches, R. "The DAPA Knowledge Sharing Effort: Progree Report." In B. Neches, C. Rich, and W. Swartout (eds.) *Principles of Knowledge Representation and Reasoning: Proc. Of the Third International Conference on Knowledge Representation (KR'92)*, Dan Mateo, CA, November 1992. Morgan Kaufmann.

15. Tennenbaum, M., Weber, J., & Gruber, T. "Enterprise Integration: Lessons from Shade and Pact." In C. Peter (ed.) *Enterprise Integration Modeling*. MIT Press, 1993.

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