Agent-based approach for manufacturing integration: The ciimplex experience

Y. Peng , T. Finin , Y. Labrou , R.S. Cost , B. Chu , J. Long , W. J. Tolone & A. Boughannam

AGENT-BASED APPROACH FOR MANUFACTURING INTEGRATION: THE CIIMPLEX EXPERIENCE

Y. PENG, T. FININ, Y. LABROU, and R. S. COST
Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, Baltimore, Maryland, USA

B. CHU, J. LONG, and W. J. TOLONE
Department of Computer Science, University of North Carolina, Charlotte, North Carolina, USA

A. BOUGHANNAM
IBM Corporation, Boca Raton, Florida, USA

The production management system used by most manufacturers today consists of disconnected planning and execution processes and lacks the support for interoperability and collaboration needed for enterprise-wide integration. This situation often prevents the manufacturer from fully exploring market opportunities in a timely fashion. To address this problem, we are exploring an agent-based approach to intelligent enterprise integration. In this approach, 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 parts of the production management system and humans to arrive at timely decisions in dealing with various enterprise scenarios. The proposed multiagent system, including its architecture and implementation, is presented and demonstrated through an example integration scenario involving real planning and execution software systems.

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 processes (Vollmann et al., 1992).* 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,

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

Address correspondence to Yun Peng, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, USA. E-mail: ypeng@cs.umbc.edu

* In the rest of the article these planning and execution application software systems are referred to as P/E applications.

machines, material, etc.) are used in producing different products. The Manufacturing Execution System (MES) tracks real-time status of work in progress, enforces routing integrity, and reports labor/material claims. Most of these P/E applications are legacy systems developed over years. Although these software systems perform well for designated tasks, they are not equipped to handle complex business scenarios (Bermudez, 1996; Jennings et al., 1996; Tennenbaum et al., 1993). Typically, such scenarios involve coordination of several P/E applications to respond to external environmental changes (price fluctuations, changes of requests from customers and suppliers, etc.) and internal execution dynamics within an enterprise (resource changes, mismatches between plan and execution, etc.). Timely solutions to these scenarios are crucial to agile manufacturing, especially in the era of globalization, automation, and telecommunication (Dourish & Bellotti, 1992). Unfortunately, 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 matching funds from the National Institute of Standards and Technology. The primary goal of the consortium is to develop technologies for intelligent enterprise-wide integration of planning and execution for manufacturing (Chu et al., 1996). Our vision of a successful integrated P/E system for manufacturing has the following features.

1. Interoperability: heterogeneous P/E applications from different vendors are able to operate together as integrated parts of the system.
2. Integration: software tools and infrastructures to support integration tasks not covered by existing P/E applications are provided. In particular, the integrated solution should support runtime dynamic coordination in dealing with unexpected events.
3. Distributed: resources such as software and data are allowed to be physically or logically distributed.
4. Openness: the user shall be able to select and change different applications and tools easily and with little additional integration cost.

One approach to an integrated P/E system might be to rewrite all application software into a monolithic integrated P/E system capable of handling all foreseeable scenarios. This approach is judged to be unfeasible because of high development and maintenance cost, and the closedness and inflexibility of such monolithic systems (Hammer, 1996). Conventional object-oriented distributed systems also seem inadequate because they work at the level of objects and thus lack the support for abstraction at higher levels (Jennings & Wooldridge, 1998).

Instead, CIIMPLEX has adopted as one of its key technologies the approach of intelligent software agents and is developing a multiagent system (MAS) for enterprise integration. In sharp contrast to traditional software programs, software agents are programs that help people solve problems by collaborating with other software agents and other resources in the network (Bradshaw et al., 1998; Compositional Research Group, 1996; Jennings & Wooldridge, 1998; Nwana, 1996; Parunak et al., 1997). For instance, individual agents can be designed to perform data collection and analysis of plans and schedules and to keep constant vigil against mismatches among these plans and schedules at different levels of abstraction and time horizons. Other 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. Still other agents can be created to provide legacy systems with better communication and coordination capabilities so they can more effectively cooperate with each other and with other agents. Moreover, MAS, as a society of autonomous agents, is inherently open and distributed, and interagent communication capability provides the essential means for agent collaboration.

The environment of the CIIMPLEX consortium is different from that of academic-oriented research laboratories. Most of the companies in the consortium are P/E application system vendors and users. This situation gives us the opportunity to work with real-world P/E systems (rather than imagined toy problems) and appreciate the complexity of realistic business scenarios. On the other hand, the agent-base approach, as a relatively immature technology that has not yet reached industrial strength, understandably receives only guarded enthusiasm by some members in the consortium. They are more concerned with integration of the P/E systems, using more mature technologies, to better handle normal or expected business scenarios. Our immediate priority is thus not to design and develop a complete agent system that integrates all aspects of manufacturing planning and execution but to develop one that is limited in scope but reliable and scalable and clearly adds commercial value to the end user. In addition, the initial prototype agent systems must have minimum interference with the normal work of existing P/E applications.

Based on these considerations, we have decided to concentrate on those P/E scenarios that represent exceptions to the normal or expected business processes and whose resolution involves several P/E applications. For example, consider the scenario involving a delay of the shipment date on a purchased part from a supplier. This event may cause one of the following possible actions: the manufacturing plan is still feasible (no action is required); order substitute parts; reschedule; or reallocate available material. To detect this exception and determine which of these actions to take, different applications (e.g., MES, ERP, CA, and Scheduler) and possibly human decision makers must be involved. Examples of similar scenarios include a favored customer's request to move ahead the delivery date for one of its orders, a machine breakdown being reported by MES, a crucial operation having its processing rate decreased from the normal rate, to mention just a few.

Figure 1 illustrates at a conceptual level how an exception (e.g., a shipment of a purchased part is delayed) should be handled by an integrated system. The decision module decides, with the instruction from a human or an analysis module, what constitutes an exception. The monitoring module determines what data are to be monitored to detect such exceptions and conducts actual monitoring of the data stream. When notified by the monitoring module of the occurrence of an exception, the decision module makes appropriate decisions in consultation with other P/E applications and the analysis module. The decision (e.g., a request to reschedule) will then be carried out by the designated P/E application(s). Note that what constitutes an exception and how to monitor it is a dynamic decision that cannot be

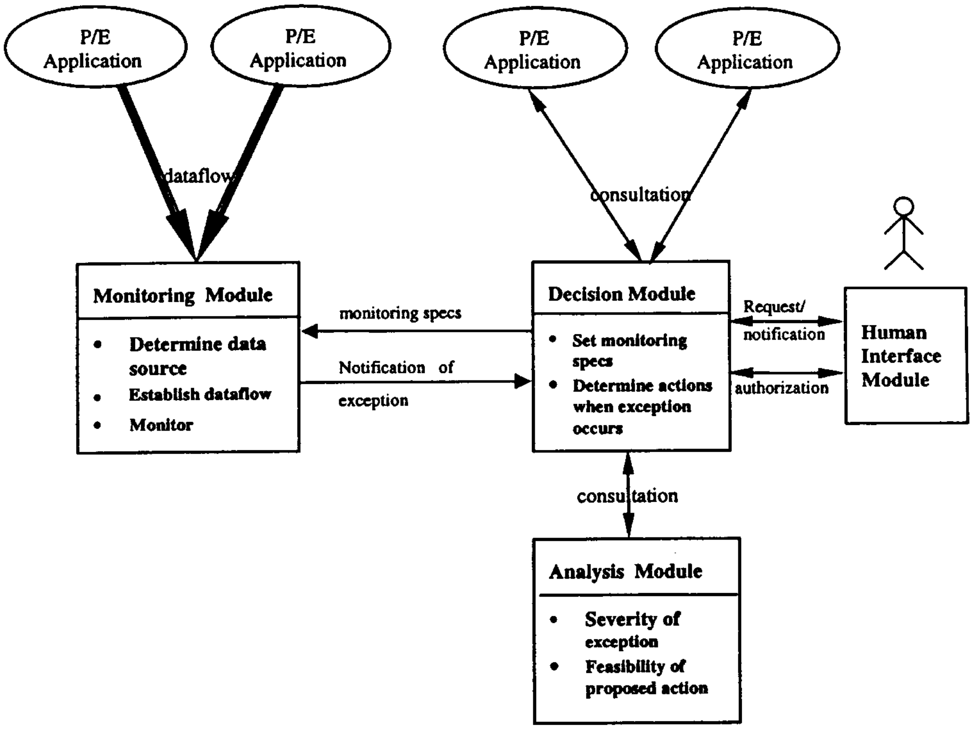

FIGURE 1. Manufacturing integration example: handling exceptions.

specified prior to the plan execution. For example, a factory may not normally consider a delay of shipment of an ordered part exceptional unless the delay is greater than five days. However, if a part is crucial for an order of a preferred customer or the inventory of a part is below a threshold, then a delay of greater than three days may become an exception. To make the situation more complicated, an action taken to address one exception may trigger another exception to occur (e.g., a reschedule to handle the delay of shipment of one part may delay the delivery date of an important order for which the respective sales representative needs to be notified).

To provide an integrated solution to the above outlined scenarios, simple as they are, is by no means a trivial undertaking. First, a set of agents of specialized expertise needs to be developed to provide functions, such as those performed by the analysis module, decision module, and monitoring module in Figure 1, which are not covered by any of the existing P/E applications. As integration tasks, these functions fall in the "white space" between these P/E applications. Second, a reliable and flexible interagent communication infrastructure needs to be developed to allow agents to effectively share information, knowledge, and services. Third, some means to support interaction with P/E applications, which will not be agentified at this stage, needs to be provided. And finally, a mechanism for the runtime collaboration of all these pieces also needs to be developed.

In this article, we describe our experience of developing an agent-based system for the CIIMPLEX project. The rest of this article is organized as follows. In the next section, we briefly describe the software agent technologies that are relevant to the task of manufacturing integration. The two subsequent sections form the core of this article, where we first present the proposed multiagent system architecture and then its work through an example scenario that involves real P/E applications. The final sections evaluate our approach and compare it with other, related work and conclude with discussions of ongoing work to expand the agent system, along with the directions for future research.

MULTIAGENT SYSTEM AND AGENT COLLABORATION

There is currently 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, to varying extents, three important general characteristics: *autonomy*, *adaptation*, and *cooperation* (Genesereth & Katchpel, 1994; Nwana, 1996). 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 proactive and take initiatives, as they deem appropriate. 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 among agents is probably the most important feature of MASs (Nwana, 1996). Unlike those standalone agents, agents in many MASs *share* information, knowledge, and tasks among themselves and collaborate with each other to achieve common goals. The intelligent properties of a MAS are not only reflected by the expertise of individual agents but also by the emergent behavior of the entire collection.

Cooperation and coordination of agents in a MAS requires agents to be able to understand each other and to communicate with each other effectively. The infrastructure that supports agent cooperation in a MAS 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 (Patil et al., 1992). Three component technologies developed by the KSE are adopted. They are Knowledge Query Manipulation Language (KQML) as a communication language and protocol, Knowledge Interchange Format (KIF) as the format of the communication content, and the concept of a shared ontology. In what follows, we briefly describe these three components and justify their selections in the context of a manufacturing integration environment.

KQML

KQML is a message-based ACL (Finin et al., 1993, 1998), and it considers that each message not only contains the content but also the attitude or "intention" the sender has for that content. For instance, consider 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 attaches some other intention to this statement. KQML provides a formal specification for representing the intentions of messages through a set of predefined *performatives* used in the messages. A subset of KQML performatives that is particularly relevant to our agent system includes *ask-one*, *tell*, *advertise*, *subscribe*, *recommend-one*, *error*, *sorry*, etc.

A KQML message is thus divided into three layers: the content layer, the communication layer, and the message 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 tag 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 structured syntax of KQML messages is based on a balanced parentheses list whose first element is the performative and the rest are the parameters in the form of keyword/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 where ?x is an uninstantiated variable. The reader is referred to Finin et al. (1993) for a detailed description of the KQML specifications.

```
(ask-one
   :language     KIF
   :content      (Price IBM  ?x)
   :sender       joe
   :receiver     stock-server
   :reply-with   ⟨ a unique string as the tag of this message⟩
```

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 content format is efficiency. Instead of many-to-many format conversion, each agent only needs to convert the content of the message between its own internal representation and the common format. KIF, because of its rich expressive power and simple syntax, is probably the most widely used neutral message content format for agent communication.

KIF is a prefix version of First Order Predicates Calculus (FOPC) with extensions to support nonmonotonic reasoning and definitions (Genesereth et al., 1992). The language description includes specifications for both its syntax and its semantics. Besides FOPC expressions of facts and knowledge, KIF also supports extralogical expressions such as those for the encoding of knowledge about knowledge and of procedures. KIF is currently the subject of an American National Standard Institute (ANSI) standardization study.

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 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 the databases of two agents). The same term may have different definitions in different agents' internal representations ("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 or at least that 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 (Patil et al., 1992). An ontology for a domain is a conceptualization of the world in terms of the 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.

Agent Collaboration

With a common communication language, a common 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 these three essential ingredients, some common service agents are often used in a MAS to make agent collaboration more efficient and effective. One type of a service agent is the Agent Name Server (ANS). The ANS, similar to the White Pages phone book, serves as the central repository of the contact addresses for all involved agents, i.e., it maintains an address table of all registered agents, accessible through the agents' symbolic names. Newly created agents must register with the ANS their names, contact 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 how to contact the ANS). The ANS maps the symbolic name of a registered agent to its contact address when requested by other agents.

Another type of a service agent is the Facilitator Agent (FA), which provides additional services to other agents. A simple FA is a Broker Agent (BA), which provides a "Yellow Pages" type service. It registers services offered and requested by individual agents and dynamically connects available services to requests whenever possible. Agents register their available services by sending BA 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 that has advertised as being able to provide the requested service at the BA, or *sorry* if such request cannot be met by current advertisers.

CIIMPLEX AGENT SYSTEM ARCHITECTURE

In this section, we describe the MAS architecture that supports interagent cooperation in the CIIMPLEX project, emphasizing the agent communication infrastructure. Figure 2 gives the architecture of CIIMPLEX enterprise integration with MAS as an integrated part.

At the current stage of the project, the entire P/E integration architecture is composed of two parts, the P/E application world and the agent world, and is supported by two separate communication infrastructures. Although these legacy P/E applications have not been agentified, they have been wrapped with APIs, which provide them with limited communication capability (Chu et al., 1998). Different transport mechanisms (e.g., MQ Series of IBM and VisualFlow of Envisionit) are under experimentation as communication infrastructures for the wrapped P/E applications. These mechanisms are persistent but only support static, predetermined communication patterns.

In the agent world, besides the service agents ANS and BA, several other types of agents are useful for enterprise integration. For example, datamining/parameter-estimation agents are needed to collect, aggregate, interpolate, and extrapolate the raw transaction data of the low-level (shop floor)

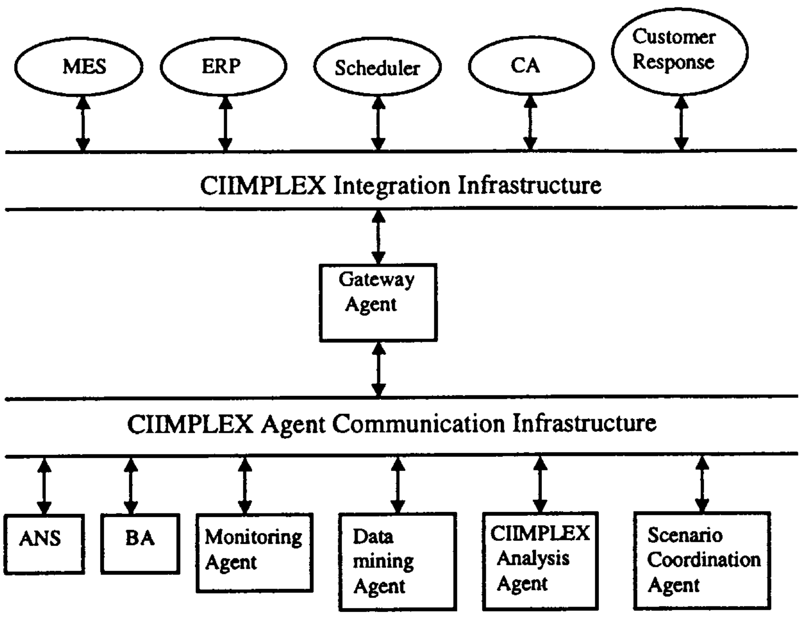

FIGURE 2. CIIMPLEX integration architecture.

activities, and then to make this aggregated information available for higher level analyses by other agents. 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. The Scenario Coordination Agents (SCA) assist human decision making for specific business scenarios by providing the relevant context, including filtered information, actions, as well as workflow charts.

All these 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. 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 P/E 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 among P/E applications such as MES and ERP, and between agents and applications. 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 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).

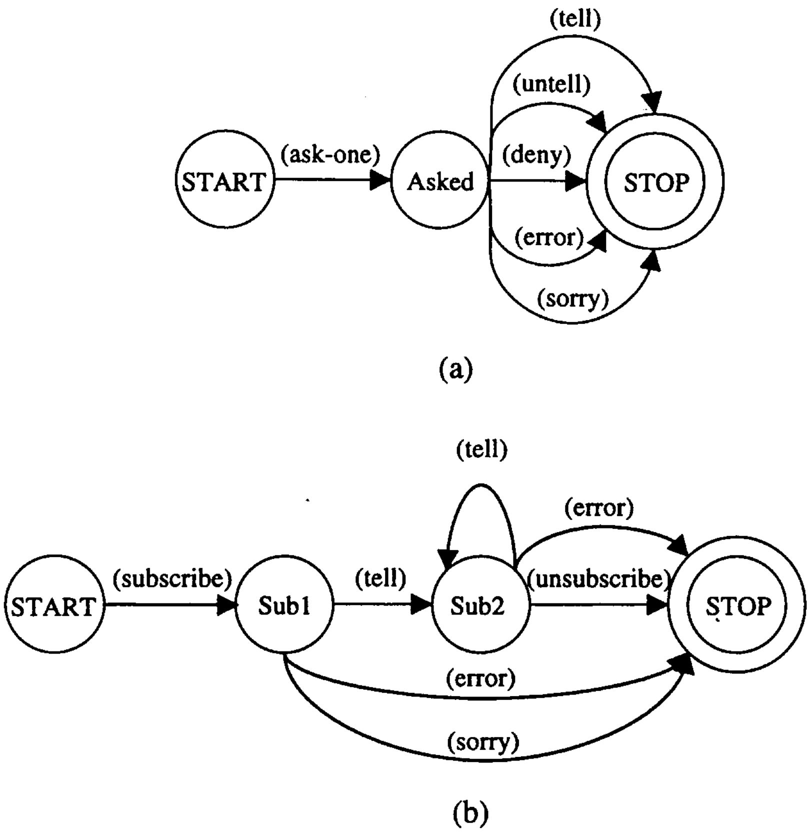

FIGURE 3. DFA state-diagrams for selected KQML conversations: (*a*) *ask-one* conversation and (*b*) *subscribe* conversation.

The agent system architecture outlined above is supported by the agent communication infrastructure called JACKAL developed by the consortium (Cost et al., 1998). The acronym JACKAL indicates that JACKAL is written in *J*AVA to support agent communication using the *KQ*ML agent communication language. The decision to select JAVA as the implementation language was based mainly on its interplatform portability, its networking facilities, and its support for multithread programming. The next two subsections provide a detailed description of JACKAL.

Conversation Policies in JACKAL

KQML itself only defines the syntax of the language. However, a good, workable semantics is imperative for conducting coherent conversation among agents. To support both syntactic and semantic aspects of the language, JACKAL takes a semantic interpretation of KQML for Labrou

(1996) and Labrou and Finin (1997) and realizes part of it as a set of conversation policies.

The conversation policies are procedures that, based on the performatives involved, specify how a conversation (consisting of a sequence of messages) 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 reply. (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; a *tell* message, whose content contains an answer to the given question; or an *untell* message, which retracts a previous *tell* message). A conversion 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* continuing to listen for new messages from *B* that satisfy the subscription criterion.

Conversion policies chosen for JACKAL can be described using a Deterministic Finite Automata (DFA) model. 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. Figure 3 shows examples of the DFAs for *ask-one* and *subscribe* conversations.

Overview of JACKAL's Design

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

Figure 4 presents the principal JACKAL components and the basic message path through the system. We will first discuss each of the components and then, to illustrate their interaction, trace the path of a message through the system.

JACKAL Components

The **Intercom** class is the bridge between the agent application and JACKAL. It controls startup and shutdown of JACKAL, provides the application with access to internal methods, houses common data structures, and plays a supervisory role to the communications infrastructure. Agents create and use an instance of **Intercom** rather than extending an agent shell. This gives us great flexibility for the design and implementation of other parts of the agent.

JACKAL runs a Transport Module for each communication protocol it uses. The current JACKAL implementation includes a module for TCP/IP,

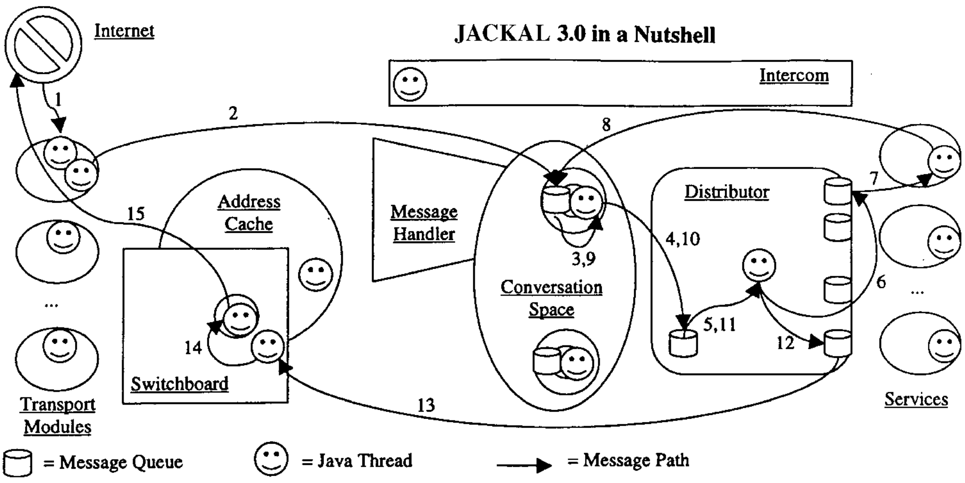

FIGURE 4. Overview of JACKAL.

and users can create additional modules for other protocols. The Transport Module is responsible for message transmission and receipt.

Messages received by the Switchboard must be directed to the appropriate places in the Conversation Space. This is the role of the Message Handler. Messages are associated with current (logical) threads based on their ID (the value of the "reply-with" or "in-reply-to" field). This directs their assignment to ongoing conversations when possible. If no such assignment can be made, a new conversation appropriate to the message is started.

Conversation protocols, described above, are "run" as independent threads for all ongoing conversations. This allows for easy context management, while providing constraints on language use and a framework for low-level conversation management.

The Distributor is a Linda-like (Carriero & Gelertner, 1989) shared memory space (conceptually like a 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 permit actions such as removing an acquired message from the blackboard or marking it as read-only. A priority setting determines the order of matching. Requests can be set to persist indefinitely or to terminate after a certain number of matches.

A service here is any thread; this could be a JACKAL service or threads within the agent itself. The only thing that distinguishes among threads is the request priority they use. System, or JACKAL, threads choose from a set of higher priorities than agent threads, but each chooses a level within its own pool. JACKAL reserves the highest and lowest priorities for services directing messages out of the agent and for those cleaning the blackboard, respectively.

The Switchboard acts as an interface between the Transport Modules and the rest of JACKAL. It must facilitate the intake of new messages, which it gathers from the Transport Modules, and carry out send requests from the application. The latter is a fairly complicated procedure, since it has multiple protocols at its disposal. The Switchboard must formulate a plan for the delivery of a message, with the aid of the Address Cache, and pursue it for an unspecified period of time, without creating a bottleneck to message traffic.

The problem of agent naming arises in any multiagent system. A naming scheme, called KNS is developed in JACKAL. KNS is a hierarchical scheme similar in spirit to DNS. A fully qualified agent name (FQAN) is a sequence of period-separated agent names (e.g., phil.cs.edu). The sequence denotes registrations between agents. Agents can register together to form teams and can maintain multiple identities to represent roles in the multiagent system. Multiple registrations for an agent become a network of aliases for that agent. If one name becomes inaccessible, another can be identified to fill the gap. Moreover, since agents can maintain multiple contact information for each name, agents can change locations and leave forwarding arrangements for messages while they migrate. In this way, dynamic group formation is supported. KNS can easily be extended to support collaborating, mobile KQML-speaking agents using a variety of transport protocols. In such case, the final segment in a FQAN is always a URL (e.g., phil.cs.http://www.umbc.edu), providing unique, static location information for the base of an agent registration chain.

The Address Cache holds agent addresses in order to defray look-up costs. It is a multilayered cache supporting various levels of locking, allowing it to provide high availability. Unsuccessful address queries trigger underlying KNS look-up mechanisms, while blocking access to only one individual listing. JACKAL supports KNS transparently through an intelligent address cache.

Message Path

Having described the various components of JACKAL, we will trace the path of a received message and the corresponding reply, using the numbered arcs in Figure 4 for reference.

The message is first received by a connection thread within a Transport Module (labeled 1 in Figure 4), is processed in the Message Handler, and is transferred directly to the input queue of either a waiting or a new conversation (2). A unique thread manages each conversation. The target conversa-

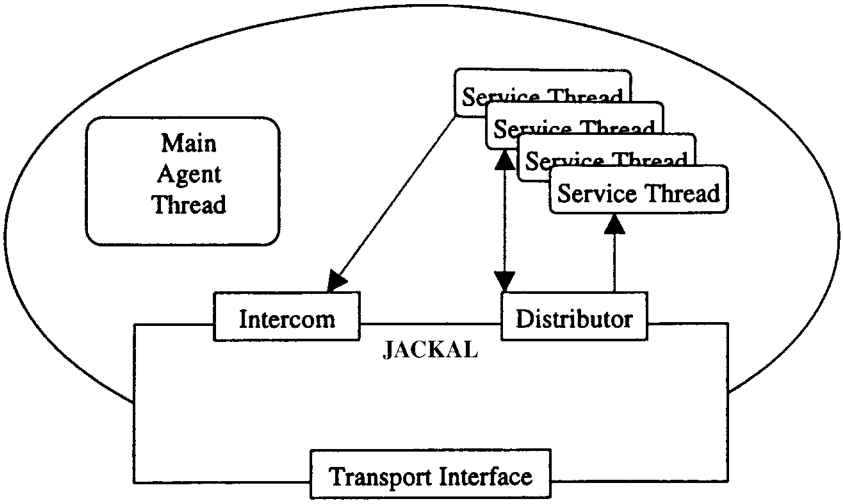

FIGURE 5. JACKAL agent skeleton.

tion awakens, takes the message from its input queue (3), and tries to advance its DFA accordingly. If accepted, the message is entered into the Distributor (4), an internal blackboard for message distribution. The Distributor (5), in turn, tries to match the message with any pending requests in order of a specified priority. Ideally, a match is found, and the message is placed in the appropriate queue or queues (6). This is the point at which the agent gains access to the message flow, through services attending to the blackboard.

Once the requesting service thread picks the message out of its queue (7), it presumably performs some action, and may send (8) a reply or a new message; we assume it does. JACKAL arranges for reply requests to be placed into the Distributor before messages are sent, when appropriate. The message is directed into the Conversation Space and traces the same path as the previous incoming message (9,10) to the Distributor. The message is captured by the Switchboard's outbound message request (11). The Switchboard removes the new message from its queue and assigns it to an idle send thread (12); this results in some overhead but allows sends to proceed concurrently, avoiding bottlenecks due to variation in delivery times. The send thread uses the appropriate Transport Module to transmit the message.

Figure 5 depicts the typical design of an agent using JACKAL for agent communication. A main thread serves primarily to start and direct JACKAL and a collection of service threads. Each service thread interacts with the Distributor to send and receive messages, and potentially with the main thread and other service threads as well. Each service thread takes some basic role within the agent, such as processing all broker requests or logging all message traffic to an archival agent.

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.

Scenario

The scenario selected, called "process rate change" and depicted in Figure 6, 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, and some others may be complicated and expensive such as requesting a rescheduling based on the changed operation rate. Two real P/E application programs, namely, FactoryOp (an MES by IBM) and MOOPI (a Finite Scheduler by Berclain), are used in this scenario.

Agents

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

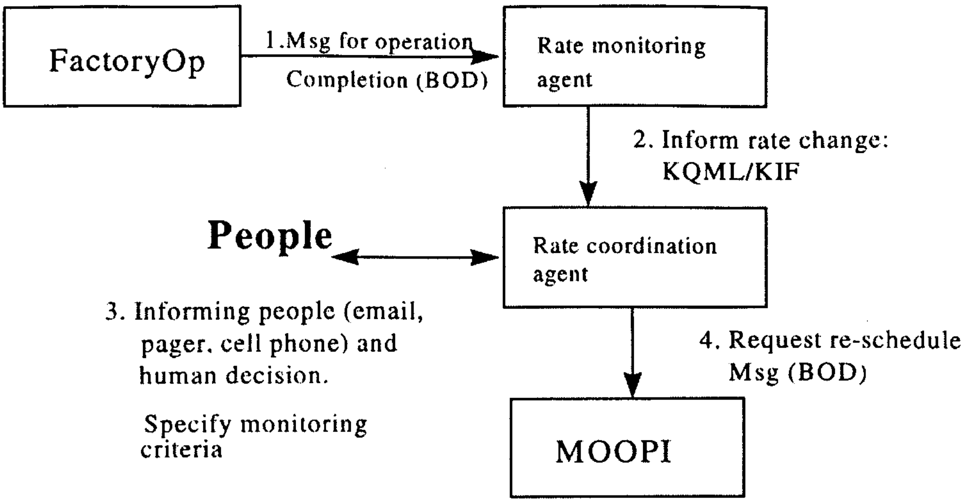
FactoryOp
1.Msg for operation Completion (BOD)
Rate monitoring agent
2. Inform rate change: KQML/KIF
People
Rate coordination agent
3. Informing people (email, pager, cell phone) and human decision.
Specify monitoring criteria
4. Request re-schedule Msg (BOD)
MOOPI

FIGURE 6. "Process rate change" scenario.

- collect in real-time 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,
- decide appropriate actions to handle the rate change, and
- carry out the actions

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 multiagent 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 originate from FactoryOp in the BOD format and are 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.
- The Scenario Coordination Agent (SCA) sets the rate-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 BOD format. Details of the internal logic and algorithms of SCA that handle the "rate change" scenario are reported elsewhere (Tolone et al., 1998).
- 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

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 process time for this operation can then be computed by the difference between the finishing and starting time stamps, divided by the 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 operation rate is represented by its mean and standard deviation over a period of time. RATE instances are computed and constantly updated by PRA, based on a stream of instances of predicate OP-COMPLETE obtained from GA.

The RATE-CHANGE predicate contains all the information needed to construct a BOD that tells MOOPI a significant rate change has occurred and a reschedule based on the new rate is called for. In particular, it contains the operation rate used to compute the current schedule and the new rate. It is the responsibility of the rate change SCA to compose an instance of the RATE-CHANGE predicate and send it to GA when it deems necessary to request MOOPI for a reschedule, 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.

Agent Collaboration and Message Flow in the Agent System

Figure 7 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 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. Since there is no predetermined stationery connection among agents, the broker agent (BA) plays a crucial role in dynamically establishing communication channels for agents' information exchange.

Advertising to BA

GA advertises that it can provide OP-COMPLETE predicate. It also advertises to be able to accept RATE-CHANGE predicate. PRA advertises

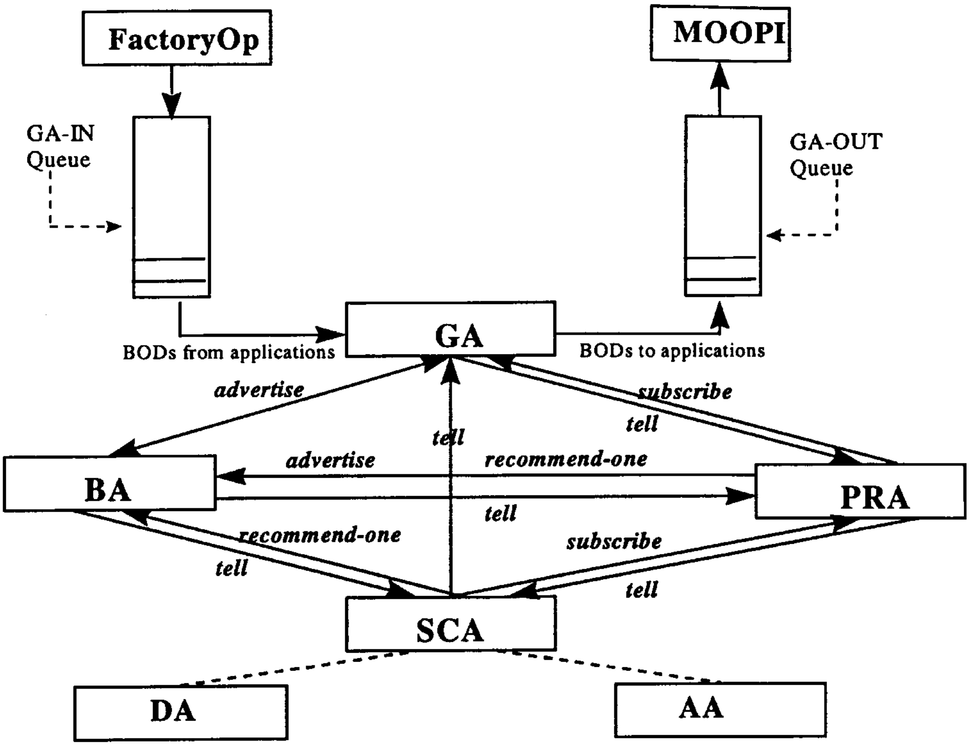

FIGURE 7. Agent system for "process rate change" scenario.

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 to BA:

```
(advertise
   :sender       GA
   :receiver     BA
   :reply-with   ⟨ a unique string as the tag of this message⟩
   :content      (subscribe: content (ask-one: content
                                  (OP-COMPLETE ?× 1 ... ?× n))))
```

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 that can provide RATE predicate and receives PRA in response. It also asks BA to recommend an agent that 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    ⟨ a unique string as the tag of this meassage⟩
   :content       (subscribe :content (ask-one :content
(OP-COMPLETE ?× 1 ... ?× n))))
```

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

```
(tell
   :sender:     BA
   :receiver    PRA
   :in-reply-to ⟨ the tag of the message to which this message responds⟩
   :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 that it can provide the current rate for a certain operation, it may send PRA the following *subscribe* message:

```
(subscribe
   :sender      SCA
   :receiver    PRA
   :reply-with  ⟨ a unique string as the tag of this message⟩
   :language    KQML
   :content     (ask-one
                      :language KIF
                      :language KIF
                      :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 less than fifty. 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.

EVALUATION

The prototype agent system outlined in the previous section has been installed at IBM's Boca Raton CIIMPLEX integration center, where a host of P/E application systems, including FactoryOp and MOOPI, is running. The system has been tested successfully. The architecture of this system can easily be applied to handle other types of P/E exception scenarios. For example, we have recently assembled another agent system to manage a different exception scenario in which some BODs sent to an application are out of sequence and need to be resynchronized.

Additional experiments have been conducted to test the reliability and scalability of JACKAL. In the experiments, JACKAL is seen to be able to support up to forty-two agents in a ring in a single NT machine (until the machine runs out of memory), and a message takes an average of 0.3 second to traverse the entire ring. JACKAL is also see to easily handle messages of large size. The largest ones we have tested are binary image data of half megabytes.

In summary, the prototype system achieves the following, which as discussed in the beginning of this article are essential for manufacturing planing and execution integration:

1. Specialized agents (e.g., PRA, MA, and SCA) are built to provide the functionality needed to manage P/E exceptions in manufacturing. These agents fill the white space in between legacy P/E application systems.
2. JACKAL, the interagent communication infrastructure, is reliable and scalable. It is also easy to use in developing individual agents because it imposes minimum interference between communication and other agent functions.
3. Conversation policies implemented in JACKAL realize, to some extent, the belief-desire-intention (BDI) model intended by KQML. These policies can be used to enforce semantically correct agent-to-agent conversations.

4. The brokering agent (BA) provides necessary support for flexible agent-to-agent collaboration. The gateway agent (GA) provides the interface between the agent world and the application world. They together make runtime collaboration among all modules possible.
5. Ontological support (i.e., the BOD definitions), although very limited at this moment, enables collaboration between agents and legacy P/E systems.

In comparison with some existing work of others, our work is more focused on particular needs of the CIIMPLEX integration environment. Works such as ZEUS (Nwana et al., 1998), dMARS based on the Procedural Reasoning System (Georgeff & Lansky, 1987), ADEPT (Jennings et al., 1996), and RETSINA (Sycara et al., 1996) attempt to provide generic agent construction environments and toolkits or general agent architecture. Our work can be seen as more of a point solution, although some techniques from this work, such as JACKAL, can be used as a general-purpose communication infrastructure for KQML-speaking agents.

We do not impose a unified architecture for all of our agents. Besides the interface to JACKAL, each agent is free to choose its own internal structure. Currently, we are considering Linda-like shared memory or lightweight blackboard architecture for intraagent (component-to-component) interaction.

There is great deal of research activity in developing agent systems for manufacturing and other business applications. Examples of such systems include ADEPT (Jennings et al., 1996) for business management, COOL for supply chain management (Barbuceanu & Fox, 1996), and ABMEI for manufacturing integration (Shen et al., 1998), to mention just a few. A distinguishing characteristic of our work is that we explicitly deal with real legacy P/E systems. Although others discuss legacy systems in principle (Jennings & Wooldridge, 1998; Shen et al., 1998), these legacy systems are seldom included in the actual implementation.

Another major difference between our work and some others is in the way the agent collaboration is achieved. COOL (Barbuceanu & Fox, 1996) emphasizes the high-level coordination by negotiation and extends KQML to support the specification of conversations for negotiation.* ABMEI (Shen et al., 1998) uses a network of mediators whose main purpose is to resolve heterogeneity between subsystems. Agents in their systems presumably have the knowledge of what agents to contact when certain needs arise, while in

* Like the conversation policies in our system, COOL's conversations are also defined using DFA's. The difference is that their conversations specify negotiation conventions, and our conversations implement intended semantics for KQML performatives based on speech act theory. In other words, they are at different levels of abstraction, and for different purposes: theirs is for specifying what it will be, and ours for ensuring what it should be.

our system, agents do not assume knowledge of what others can do. Each agent advertises services it can provide and announces what services it needs. The service requests match advertises by the BAs, and the communication between the matched pairs then follows. The Open Agent Architecture (OAA) (Martin et al., 1998) goes further to use a powerful facilitator to coordinate agents' activities in a system. The facilitator works like a planner. Based on the knowledge stored, it is able to decompose a task received from an agent into subtasks, delegate subtasks to appropriate agents, and monitor and coordinate the executions of these subtasks. A drawback of OAA is that the facilitator is also the communication center. All agents must communicate via the facilitator, while in our system, agents can communicate directly to each other after the brokering matches.

CONCLUSION

In this article, we presented a multiagent system that is capable of supporting intelligent integration of manufacturing planning and execution, especially in managing the exceptions in business scenarios. 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, in order to arrive at timely decisions in dealing with various enterprise scenarios. This system has been tested successfully with a real manufacturing scenarios involving real legacy MES and schedulers.

The work presented here represents only the first step of our effort toward agent-based enterprise integration for manufacturing planning and execution. Further research and experiments are needed to extend the current work and to address its shortcomings. Although KQML does not impose many constraints and requirements on the internal structure of agents, it may 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 reusability, and plug-and-play. Another research direction under consideration is to increase the functionality of the BA 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. The current ontological support is very limited. It only provides definitions of various BODs and constraints of BOD fields to ensure data consistency. We are currently extending the ontology to include deductive rules for additional interrelations between different BODs and BOD fields to support more complicated business scenarios. 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.

REFERENCES

Barbuceanu, M., and M. S. Fox. 1996. The architecture of an agent building shell. *Intelligent Agents II* 1037:235–250.

Bermudez, J. 1996. Advanced planning and scheduling systems: Just a fad or a breakthrough in manufacturing and supply chain management? Report on Manufacturing, Advanced Manufacturing Research, Boston, Massachusetts, December.

Bradshaw, J., S. Dutfield, P. Benoit, and J. Woolley. 1997. KAoS: Toward an industrial-strength open agent architecture. In *Software agents*, ed. J. M. Bradshaw. Boston, Massachusetts: MIT Press, 375–418.

Carriero, N., and D. Gelertner. 1989. Linda in context. *CACM* 32(4):444–458.

Chu, B., J. Long, M. Matthews, J. Barnes, J. Sims, M. Hamilton, and R. Lambert. 1998. FAIME: An object-oriented methodology for application plug-and-play. To appear in *Journal of Object-Oriented Programming*.

Chu, B., W. J. Tolone, R. Wilhelm, M. Hegedus, J. Fesko, T. Finin, Y. Peng, C. Jones, J. Long, M. Matthews, J. Mayfield, J. Shimp, and S. Su. 1996. Integrating manufacturing softwares for intelligent planning-execution: A CIIMPLEX perspective. Plug and play software for agile manufacturing. *Proceedings of SPIE* 2913:96–108.

Composition Research Group. 1996. Caltech Infosheres Project. Available at http://www.infospheres.caltech.edu

Cost, R. S., T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soloroff, J. Mayfield, and A. Broughamnam. 1998. Jackal: A JAVA-based tool for agent development. In *Working Notes of the Workshop on Tools for Developing Agents* (*AAAI Technical Report*), *AAAI-98. http://jackal.cs.umbc.edu/cost/cv/pub/aaai98.pdf*

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

Finin, T., et al. 1993. Draft specification of the KQML agent communication language. Available at http://www.cs.umbc.edu/kqml/kqmlspec/spec.html

Finin, T., Y. Labrou, and J. Mayfield. 1997. KQML as an agent communication language. In *Software agents*, ed. J. M. Bradshaw. Boston, Massachusetts: MIT Press, 291–316.

Genesereth, M., and S. Katchpel. 1994. Software agents. *Communication of the ACM* 37(7):48–53.

Genesereth, M., et al. 1992. Knowledge interchange format, version 3.0 reference manual. Technical report, Computer Science Department, Stanford University, Stanford, California.

Georgeff, M. R., and A. L. Lansky. 1987. Reactive reasoning and planning. *Proc. of AAAI* 677–682.

Hammer, M. 1996. *Beyond reengineering: How the process-centered organization is changing our work and our lives*. New York: HarperCollins.

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

Jennings, N. R., and M. L. Wooldridge. 1998. Applications of intelligent agents. In *Agent technologies: Foundations, applications, and markers*, eds. N. R. Jennings and M. J. Wooldridge, 3–28.

Labrou, Y. 1996. Semantics for an agent communication language. Ph.D. dissertation, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, August.

Labrou, Y., and T. Finin. 1997. Semantics and conversations for an agent communication language. In *Proc. of the 15th Int. Joint Conf. on Artificial Intelligence* (*IJCAI-97*). San Mateo, CA: Morgan Kaufmann. *http://umbc.edu/~finin/papers/atal97.pdf*

Martin, D. L., A. J. Cheyer, and D. B. Moran. 1998. Building distributed software systems with the open agent architecture. In *Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems*, 355–376. London, UK.

Nwana, H. S. 1996. Software agents: An overview. *Knowledge Engineering Review* 11(3).

Nwana, H. S., D. T. Ndumu, and L. C. Lee. 1998. ZEUS: An advanced tool-kit for engineering distributed multi-agent systems. In *Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems*, 377–392. London, UK.

Parunak, H. V. D., A. Baker, and S. Clark. 1997. AARIA agent architecture: An example of requirements-driven agent-based system design. Available at http://www.aaria.uc.edu

Patil, R., R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. 1992. The DAPA knowledge sharing effort: Progress report. In *Principles of knowledge representation and reasoning: Proc. of the Third International Conference on Knowledge Representation* (*KR '92*), eds. B. Neches, C. Rich, and W. Swartout. San Mateo, California: Morgan Kaufmann.

Shen, W., D. Xue, and D. H. Norrie. 1998. Agent-based manufacturing enterprise infrastructure for distributed integrated intelligent manufacturing systems. In *Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems*, 533–550. London, UK.

Sycara, K., A. Pannu, M. Williamson, and D. Zeng. 1996. Distributed intelligent agents. *IEEE Expert* 11(6):36–46.

Tennenbaum, M., J. Weber, and T. Gruber. 1993. Enterprise integration: Lessons from shade and pact. In *Enterprise integration modeling*, ed. C. Peter. Boston, Massachusetts: MIT Press.

Tolone, W. J., B. Chu, J. Long, T. Finin, and Y. Peng. 1998. Supporting human interactions within integrated manufacturing systems. To appear in *International Journal of Agile Manufacturing* .

Vollmann, T., W. Berry, and D. Whybark. 1992. *Manufacturing planning and control systems*. New York: Irwin.