An Agent System for Application Initialization in an Integrated Manufacturing Environment

Yun Peng, Tim Finin, Harry Chen, Ling Wang, Yannis Labrou, R. Scott Cost
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County, Baltimore, MD 21250, USA

Bill Chu, Valerie Cross, Mike Russell, Bill Tolone
Department of Computer Science
University of North Carolina, Charlotte, NC 28223, USA

Akram Boughannam, Jerry McCobb
IBM Corporation, Boca Baton, FL 33431, USA

ABSTRACT

A great deal of research and development effort has been undertaken in recent years to integrate otherwise disconnected manufacturing production and planning (P/E) application software systems so that the enterprises can react quickly and accurately to the ever changing market dynamics. In such an integrated P/E environment it is, at times, necessary to bring in a new application in order to replace an outmoded one or to provide functionality that is not available in the existing environment. The process of introducing a new application, *Application Initialization*, is very complicated and can be extremely costly in terms of time and resources.

In this paper we propose a multi-agent system (MAS) to assist with the application initialization process. A collection of eight agents with specialized expertise is assembled to carry out the operational steps of the initialization process. These agents interact with each other, with other applications in the environment, and with the human specialist in the process. All agents speak KQML language, supported by *Jackal*, a JAVA based, light-weighted and flexible agent communication infrastructure we have developed.

The MAS is tested with an integrated environment involving real P/E applications. The success of the experiment demonstrates that MAS is a technically viable approach for providing flexible and inexpensive solutions to difficult tasks (application initialization and others) in the integration of manufacturing planning and execution.

Keywords: Multi-agent Systems, Agent Communication, Manufacturing integration, Application Initialization.

1. INTRODUCTION

The production management system used by most of today's manufacturers consists of a set of separate planing and execution (P/E) application software [15, 17] such as Capacity Analyzer (CA), Enterprise Resource Planner (ERP), Finite Scheduler (FS), and Manufacturing Execution System (MES). Most of these P/E applications are legacy systems developed over years. Although performing well for their respective designated tasks, they are not equipped to automatically handle complex business scenarios in which several P/E applications are involved [10, 13]. A great deal of research and development effort has been undertaken in recent years in order to develop technologies for integrating otherwise disconnected (P/E) applications so that an enterprise can react quickly and accurately to the ever changing market dynamics.

We at the Consortium for Intelligent Integrated Manufacturing Planning-Execution (CIIMPLEX) have take the manufacturing integration our primary research goal, and have adopted as one of its key technologies the approach of intelligent software agents [3, 11]. In sharp contrast to traditional software programs, software agents are programs that help people to solve problems by collaborating with other software agents and other resources in the network [2, 7, 11]. A multi-agent system (MAS), as a society of autonomous agents, is inherently open and distributed, and the system's inter-agent communication capability provides the essential means for agent collaboration that aids interoperability of the system. [13]. A number of MAS prototypes have been constructed recently for enterprise integration. Examples of such systems include ADEPT [10] for business management, COOL for supply chain management [1], AMBEI for manufacturing integration [14], to mention just a few. In the past, we have also developed MAS for monitoring and management of exceptions in shop-floor [11] and in exchanges of transactional data [16].

Recently we have focused on another type of problem, namely, the problem of *Application Initialization*. The integrated P/E environment can be viewed as dynamically evolving, and it is, at times, necessary to introduce into the integrated environment a new application in order to replace an outmoded application or to provide function that is not available in the existing environment. Among other things, to ensure that the new application works properly with the environment, its state must be brought in sync with other, existing applications in the environment (the process is thus also called *Initial Sync*). This is a complex process involving dynamic interactions among the integration specialist, the new application, the existing applications, and the integration services provided by the integrated environment. It can be extremely costly in terms of time and resources. In some cases, it takes an experienced specialist weeks or even months to complete the initial sync of a particular application before it can be put into operation.

In this paper we report our research efforts and experiences in developing a MAS that assists human integration specialists for application initialization. The rest of the paper is organized as follows. To help motivate the work, we describe in Section 2 in detail what is involved in application initialization through an example scenario. Section 3 presents the design of the MAS, including the functionality of individual agents in the system.

Section 4 describes the communication aspects of the system and the supporting infrastructures. Section 5 briefly describes the operation of the system and experiments with a real-world integration environment. Finally, we discuss issues and directions for future research in Section 6.

2. AN INITIAL SYNC SCENARIO

In this section we describe an example scenario of application initialization and the key steps of operations involved. As depicted in Figure 1, a new application, a finite scheduler (BOOM, a Berclain product) is to be added into an existing integrated environment, which includes a number of P/E applications, such as one or more enterprise resource planners (ERP), manufacturing execution systems (MES), etc. These applications exchange transactional data by sending messages to each other via CIIMPLEX Infrastructure [13]. To ease the problem caused by heterogeneity of native data representations in these applications, a neutral, common format, the Business Object Document (BOD) sponsored by Open Application Group (OAG), was adopted for the message contents when applications communicate with each other through the infrastructure. A BOD consists of a verb, a noun, and one or more data area that is a formatted encoding of a business transactional data record. For example, a *SyncItem* requests the recipient application to insert the item record into its database.

One property that distinguishes application initialization from the more familiar process of auto installation of software is that the database of the new application needs to be initialized with transactional data from the existing applications. For example, before BOOM can make a proper production schedule, it needs to know the current state of the production (what are the quantity and delivery date of each outstanding order, what items are in what operation stages with what machines, available machines and their capabilities, available human operators, etc.) Extracting this data from existing applications is very time consuming. It requires identifying the owner application(s) of each type of the data needed, interacting with the CIIMPLEX Infrastructure to obtain the actual data, mapping the imported BOD into BOOM's native representation etc. Moreover, the data import is often an iterative process involving tentative decisions of data mapping and mediation on a trial-and-error basis. This process thus needs to be closely monitored by a human operator or a monitoring program. In case an error occurs, the process should effectively restore the database to the latest error-free state (or the initial state) before another set of mappings is tried.

After the consulting with the BOOM specialists and the integration experts, it becomes clear that the application initialization process would include at least the following steps.

1) Notifying and registering the new application with the environment,
2) Obtaining relevant information of the existing applications in the environment,
3) Obtaining the data requirement (types of BODs) for BOOM database initialization,
4) Obtaining the ownership of each required type of data,
5) Contacting the CIIMPLEX Infrastructure to request a specific number of BOD instances of specific BOD types,
6) Importing the required BOD instances to BOOM,
7) Converting BOD into BOOM's native representation based on a tentative mapping table,
8) Checking BOOM's database for any inconsistency or other types of error,
9) Notifying the system the completion of the initialization process, and registering with the system the types of BOD BOOM can provide (to others).

Figure 1 at the end of the paper depicts the functional components and the sequence of actions involved in application initialization. The *System Management Service* mainly provides registration service of applications, including registering their functionality and status. The *Data Retrieval Service* handles all actions related to transferring data from the environment to BOOM. It obtains the BOD ownership from the *System Data Registry,* interacts with CIIMPLEX Infrastructure, and imports BOD instances from the Infrastructure to BOOM. The *Coordination Service* coordinates all these components, and interacts with BOOM's human operator. Step 0 indicates the human operator can start the entire initialization process. Steps 7 and 8, not shown on the figure, will not be handled by the agent system in the current prototype but by the operator using other software tools. As mentioned earlier, the BOD importing is an iterative process, the steps involving actual BOD importing (5 and 6) thus may be aborted, stopped and resumed later.

3. MAS DESIGN

In this section we present our design of the MAS for application initialization, including the individual agents and their functionality. All agents in the MAS use KQML as the communication language and protocol, and use a subset of KIF that supports Horn clause deductive inference as the content language [5, 6, 12].

3.1. The agents

The following eight agents are employed to carry out the initial sync scenario described in Section 2.

1) *Agent Name Server* (ANS). This agent provides "white page" service for agent name registration and name-address mapping.
2) *Broker Agent* (BA). This agent provides "yellow page" service to help the agents in the system to find the service provider.
3) *Application Initialization Assistant* (AIA). This agent plays a dual role. It acts as a proxy for the new application and interacts with the rest of the MAS. It also serves as the interface, via its GUI, for the human operator to communicate with the agent world. AIA roughly corresponds to the Coordination Service component in Figure 1, and all initialization operations are initiated from AIA.
4) *Initialization Monitoring and Management Agent* (IMMA). This agent manages BOD transfers during initialization. IMMA roughly corresponds to the Data Retrieval Service component in Figure 1.
5) *BOD Service Agent* (BSA). This agent can build BOD's when asked by others. During application initialization, it will build special BODs that IMMA uses to access the CIIMPLEX Infrastructure.
6) *Process Monitoring and Management Agent* (PRMMA). This agent monitors the BOD transactions to the new application during the initialization process. In the case of an error or an exception, detail information is logged and reported to human operator via AIA's GUI.

7) *System Management Service Agent* (SMA). This agent imitates a small subset of the functions of the System Management Service component in Figure 1. In particular, SMA provides registration service to applications in the integrated environment, and handles inquiries about the status of the existing applications in the environment.
8) *BOD Registry Agent* (BODRA). This agent maintains the BOD registration, including ownership and other relevant information of different types of BODs. It also handles inquiries for information regarding the BOD ownership and other properties. BODRA corresponds to the System Data Registry component in Figure 1.

3.2. Functionality and design of individual agents

ANS provides agent name and address mapping service. With the help of ANS, two agents can communicate with each other without knowing the other party's physical network address. ANS maintains an address table of all registered agents, accessible through the agents' symbolic names. Newly created agents must register with the ANS their symbolic names, physical address and possibly other information by sending to the ANS a message with KQML performative *register*. (As a presumption, every agent in the system must know how to contact the ANS. This is achieved in the current implementation by including the physical address of ANS in the resource files of all other agents.) The ANS maps the symbolic name of a registered agent to its contact address when requested by other agents.

BA provides a "yellow page" type service. Agents register their available services by sending BA messages with the performative *advertise*. When received an *advertise* message, BA analyzes the content of the message, and stores the analyzed result in its database. An agent that is looking for a service provider can send BA a message with the performative *recommend-one* or *recommend-all*, with the message content describing the specific service that it is looking for. BA tries to find a match between a service request and advertised services and replies the match result to the requester. For example, SMA sends the following message advertising it can insert the registration record of any application into its database

```
  (advertise
     sender:    SMA
     receiver:   BA
     language: KQML
     content:
       (insert
         language: KIF
         content: (Application   ?name   ?id   ?description   ?status)
 ))
```

And AIA sends the following message to BA asking who can register a new application before it proceeds to the actual registration:

```
 (recommend-one
     sender:    AIA
     receiver:   BA
     language: KQML
     content:
        (ask-all
           language: KIF
           content: (Application   ?name  ?id   ?description   ?status))
  ))
```

(In the current implementation of BA, the matching process is solely based on the syntactic match and variable unification, no inference mechanism is employed.) After a match is found, BA will send back a *reply* message. The content of the message is the content of the matched *advertise* message sent by the service provider. If there were more than one matches, BA will reply with the first service provider it finds. In a reply to a *recommend-all*, BA will send a *reply* message similar to the *reply* message for *recommend-one*, except that the content of the message is a conjunction of the contents of all of the matched *advertise* messages. If there is no match, BA will send back a *sorry* message with null message content.

AIA. The primary task for AIA is to assist the operator during the application initialization by reducing the human involvement to the minimum. AIA can be viewed as a mediator between the agent world and the human operator. It is composed of two components, a GUI interface component that interacts with the human operator and an agent communication component that interacts with other agents. The GUI interface is design to have "installation wizard" like look and feel that will guide the initialization process through a pre-determined (default) sequence or branches of sequences of operation steps, similar to that described in Section 2. The order of the steps is embedded in this "wizard" like GUI interface. With such design the operator is not obligated to remember all possible steps, or the order of these steps, that are involved in the initialization process. He is required only to provide operation specific parameters when a step is prompted, or occasionally make a selection from a list of alternatives. In addition, the operator is allowed, via GUI, to stop or abort, at any time, an ongoing initialization process, and to resume a stopped process. This provision is crucial since, as mentioned earlier, application initialization is often an iterative process involving tentative mapping/mediation decisions that may have to be changed.

The operation steps initiated from GUI will be converted to instructions to other agents, in the form of KQML messages, and transmitted to other agents. The communication component is responsible for taking communication requests from the GUI, and delivering these requests to the appropriate receivers. It is also responsible for receiving incoming messages from other agents, delivering the message to the GUI component for display if it deems that the content is of interest to the operator. For example, when registering the new application, AIA will send SMA the message with the content

```
      (Application   BOOM  FS-1   some-description   InitialSync)
```

meaning a new application named BOOM, with id FS-1 and the described functions and features is to be registered with status "InitialSync". Other outgoing messages from AIA include those that request IMMA to start (stop, abort, and resume) importing particular types of BODs, e.g.,

```
  (achieve
      sender:   AIA
      receiver: IMMA
      content: (StartImportBOD    Synch  Item   FS-1   API
                  the-parameter-list   start-point   10   InitialSync-001)
  )
```

The content of the message specifies the type of operation (StartImport), the verb and noun of the BOD to be imported (Sync and Item), the destination to send the imported BOD, the API to be called at the destination (FA-1 and API), and the number of BOD instances to import for the time being (10). Besides replies to its outgoing messages, AIA also receives incoming messages that include error messages from PRMMA when the new application fails to receive BODs sent from

CIIMPLEX. The error message (including BOD type and the error code) will be displayed and evaluated by the operator.

SMA is responsible for accepting registration requests from new applications as well as answering queries regarding the status of the existing applications in the system. SMA has a database component, which, implemented by MSQL, stores the statuses of all applications in the systems. In the current implementation, an application can register to SMA (when it is brought into the system or it wishes to change its status), or withdraw its registration (when it is removed from the system). These requests are sent to SMA as KQML messages (using performative *insert* for registration or changing registration, and *delete-one* for removal of registration). The message content contains the name, the identification number, and the status of the application described. All incoming KQML messages are parsed, evaluated, and converted to appropriated database operations. If the operation is successful, SMA sends back an acknowledgement with performative *reply* to the requestor. To query the status about an application, an *ask-one* message is send to SMA. The content of the message can be either a partial or a full description of the application. In generating an answer to the inquiry, the variable unification process is employed to fill any unspecified field in the original query. If the unification fails, then SMA will send back a message with performative *sorry*. If an error occurs at any time during any conversation described above, SMA will send back a message with performative *error*.

BODRA has a structure similar to SMA. The difference is that BODRA handles registration for BOD's, not applications. It stores registration information, especially the ownership, of all types of BOD that existing applications can produce. BODRA will be queried by IMMA when the latter is asked by AIA to import a particular type of BOD but does not know which application(s) owns (and thus can provide instances of) this BOD. At the end of initialization, the new application needs to register to BODRA all types of BODs it can produce for others to consume. The conversation protocols between BODRA and other agents are the same as that for SMA.

IMMA plays a central role in importing the actual existing transaction data (in BOD format) to the new application. During the process of data importing, IMMA follows the following steps.

1) Processing the message from AIA that requires importing a list of BODs required for new application to initialize its database.
2) Querying BODRA for the owners of these required BODs.
3) Asking BSA to build special, inquisitory BOD
4) Using these inquisitory BOD's to request the CIIMPLEX infrastructure to send BOD instances of interest.
5) Forwarding received BOD to the new application.

In addition, when requested by AIA, IMMA can stop or abort the ongoing BOD importing, and resume a stopped importing process.

PRMMA. While IMMA focuses on moving individual BOD instances from existing applications to the new application, PRMMA helps to manage BOD transfers by handling errors and anomalies during the process. For example, for each BOD instance IMMA obtains and sends to the new application, it receives a confirmation message back from the new application. If the message is ConfirmFail (with a specific error code indicating the reason of the failure), IMMA will notify PRMMA, which will then either process and correct the error or notify the operator via a message to AIA. Another type of possible error that PRMMA can monitor and report is called unit of work violation, which occurs when the number of BOD instances the new application receives is different from what is originally required or these BODs arrive in an incorrect order. In addition, PRMMA is also responsible to notify AIA when the transfer of a group of BOD instances (a unit of work) is correctly completed.

4. COMMUNICATION

In this section, we briefly discuss the infrastructures that support the two types of communication involved in our MAS. They are Jackal for communication between agents, and CABS for communication between components within an agent.

4.1. Inter-agent communication.

Jackal is a Java-based, multi-threaded infrastructure for communication and interaction between KQML speaking agents [4, 13]. It is also well in sync with the recent international standards effort of ACL carried by the Foundation for Intelligent Physical Agents [8]. Jackal supports all of the important functionality for agent communication and interaction. In particular

- It supports ANS transparently through an automatic registration to ANS of each newly created agent and intelligent address caches at individual agents.
- It facilitates processing incoming and outgoing messages (e.g., message composition and parsing). Moreover, it automatically assigns and matches communication related message parameters (such as reply-to, in-reply-with fields).
- Its *Conversation Interpreter* enforces conversation policies which, based on the semantics of performatives, specify the sequence of message exchanges for a meaningful agent-agent conversation.
- Its *Distributor* guarantees correct message delivery within an agent (between separate threads for individual conversations).

Jackal is very easy to use when developing and implementing software agents. Although it consists of roughly seventy distinct classes, all user interactions are channeled through one class, the *Intercom*, hiding most details of the implementation. In addition, Jackal is flexible to support multiple, different transport mechanisms or protocols. Additional protocols can be easily added into Jackal. Similarly, specifications of additional conversation policies can be added easily.

4.2. Intra-agent communication.

A complex agent is often composed of multiple functional components, which are relatively independent of, and loosely connected to each other. This is the case for some agents in our MAS. For example, AIA has two major components, the GUI and the KQML Communication module. These two components interact with each other only when an initialization step initiated from GUI needs to be transmitted to a destination agent or an incoming message to AIA is of interest to the operator and needs to be displayed on GUI's screen. This is similarly the case for IMMA, which has, among other things, a

KQML communication module, a CIIMPLEX Infrastructure Adapter, and a component that can directly communicate with BSA via remote method call rather than KQML messages. Instead of hard coding the inter-component interaction within an agent, we have developed a blackboard-based framework for intra-agent communication. This framework, called CIIMPLEX Agent Builder Shell (CABS) was used to develop AIA and IMMA. The benefits of using a blackboard system like CABS to support the interactions between loosely coupled components within an agent include flexibility, extensibility, and reusability.

CABS is composed of a blackboard that is used to store *publish* and *subscribes* events, as well as a Java-based rules engine that interprets CABS rules, JESS [9]. A component can publish an event on the board by simply calling a CABS method ("AddCBASItem"). Other components can subscribe to the blackboard a set of events that it would like to be notified when any of these events is posted onto the blackboard. When a given rule is triggered by a newly published event, CABS will notify the subscriber by calling a callback method. In the following is a partial String representation of a CABS rule:

```
Trigger condition: (AddedAIA.events.RegAppEvent ?itemId)"
Pre-condition:     (RegAppEvent (itemId ?itemId)
                         (attention \"AIACommunicator\"))
Action:            subscribe
```

This rule is added to CABS by AIA's communication component to subscribe RegAppEvent, which is to be posted by AIA's GUI component to request registering a new application. When an application registration request is generated by the GUI component, the registration information is wrapped within the event object and posted onto the blackboard. When this rule is triggered, that is, when the newly published event object contains the attribute values matched the pre-condition of a subscribe event, the callback method of the subscriber is called. In our example, the communication component is notified after a RegAppEvent is posted. Within the communication component, a sequence of KQML messages is exchanged with other agents in the system(with BA to obtain SMA, and then with SMA for the actual registration) until the registration is done. In the reverse direction, the final result of the registration will be posted onto the blackboard by the communication component and passed to the GUI component (for display) with the similar publish and subscribe mechanism.

5. MAS OPERATION AND EXPERIMENT

In this section, we describe how these agents work together to achieve the application initialization scenario through an experiment in which a new application (BOOM, a finite scheduler) is added into an existing integrated environment. In the environment some real P/E applications have already been up and ruing. In particular, we demonstrate how our MAS helps BOOM to obtain BODs from an existing application (MfgPro, an ERP product of IBM) to populate its database. In the experiment, instances of only one type of BOD, namely SyncItem, were imported from MfgPro to BOOM.

In a normal operation mode, the MAS undergoes the following major steps in the process.

1) All agents advertising their services to BA by sending BA *advertise* messages
2) Registering BOOM to SMA. This is the first step encoded in the initialization wizard. When prompted by AIA's GUI, the operator types in the registration information (see the screen dump in Figure 2 (a) at the end of the paper). Also included in this step is the inquiry by AIA about the status of existing applications in the environment.
3) Issuing commands for importing BODs. When prompted by GUI, the operator types in the types of BODs (SyncItem) and the number of BOD instances he wishes to import (see Figure 2 (b)). This command is then sent to IMMA as a KQML message.
4) Preparing BOD import. When received from AIA the message for importing BOD, IMMA takes several steps to prepare the actual BOD transfer. First, IMMA queries BODRA to obtain the ownership of SyncItem BOD. Second, it asks BSA to build "GetlistItem" BOD. This inquisitory BOD is then sent to the source application (MfgPro) via CIIMPLEX Infrastructure, and a list of ids of all instances of SyncItem is returned to IMMA.
5) Importing BOD instances. IMMA asks BSA to build "Get Item" BOD for each instance on the returned list, and then uses it to request the CIIMPLEX Infrastructure to send the instances of SyncItem one at the time. Each received BOD instance is then forwarded to BOOM. PRMMA notifies AIA when all instances required are correctly imported into BOOM,
6) Final registration. When the operator determines the initialization of BOOM's database is completed, he will first register to BOBRA all types of BOD it can produce for the consumption of other applications (see Figure 2 (c)). Later the operator will change the status of its registration in SMA from "InitialSync" to "Ready", indicating BOOM is ready to start working in the integrated environment.

All the above steps were successfully carried out by the agent system, and the required SyncItem instances were correctly imported from MfgPro to BOOM. The system worked properly when the operator, via AIA, issued commands to stop or to abort the ongoing BOD importing process, and it was able to restart the process at an earlier point of time (specified by the operator) after the process was stopped.

Finally, we tested the system's error handling capability. A ConfirmFail message was generated at BOOM in the middle of BOD importing and sent to PRMMA. This triggers PRMMA to send an error message (in KQML) to AIA, whose content was then displayed on latter's GUI window.

6. CONCLUSION

In this paper we presented our design of a multi-agent system of eight agents that can be used to assist the application initialization process in an integrated manufacturing P/E environment. The agents in this system interact with each other, with other applications in the environment, and with the human specialist to carry out the operations needed for the process. The MAS is tested with an integrated environment involving real P/E applications. The success of the experiment demonstrates that MAS is a technically viable approach to provide flexible and inexpensive solutions to difficult tasks (application initialization and others) in the integration of manufacturing planning and execution.

This work represents our first attempt of applying agent technology for application initialization. The MAS we developed can be extended easily to provide more advanced features. First of all, the system can be extended to cover more initialization tasks. One example of major tasks not covered by the current system is mapping and mediation between the representation of the imported data (in BOD format) and the native representation in the new application's database. This task can be helped by additional agents that understand not only the syntax but also the semantics of these representations and be able to directly interact with the new application, i.e., to serve as their true proxies. Another direction of future work is to provide more powerful automatic error handling capability. This may be achieved by incorporating into the initial sync the agent system for managing BOD transfer exceptions we developed earlier [16]. Finally, the way the human operators interacting with the agent system can be expanded. For example, besides accepting the operator's commands via GUI, AIA can be extended to take operation requirement from a script written, and notify the operator when errors occur. This feature is especially important in situations where the data importing takes a long time (a few days) to complete and the operator cannot (and need not to) monitor the process constantly.

7. ACKNOWLEDGEMENT

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

8. REFERENCES

1. M. Barbuceanu and M.S. Fox, The Architecture of an Agent Building Shell. In *Intelligent Agents II*, **1037**, Berling: Springer Verlag, 1996, 235-250.
2. J. Bradshaw, S. Dutfield, P. Benoit, and J. Woolley, KAoS: Toward An Industrial-Strength Open Agent Architecture. *Software Agents* J.M. Bradshaw (Ed), Boston: MIT Press, 1998, pp. 375-418.
3. B. Chu, 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, Integrating Manufacturing Software for Intelligent Planning-Execution: A CIIMPLEX Perspective. In *Plug and Play Software for Agile Manufacturing, Proceedings of SPIE* , Vol. 2913. Boston, MA, 1996, pp. 96-108.
4. R.S. Cost, T. Finin, Y. Labrou, X Luan, Y. Peng, I. Soboroff, J. Mayfield, and A. Boughannam, Jackal: A JAVA-based tool for agent development, in *Working Notes of the Workshop on Tools for Developing Agents (AAAI Technical Report)*, AAAI 1998. http://jackal.cs.umbc.edu/cost/cv/pub/aaai98.pdf.
5. T. Finin, Y. Labrou, and J. Mayfield, KQML as an agent communication language. in *Software Agents*. Bradshaw, J.M. (Ed.). Boston: MIT Press, 1998, pp. 291-316.
6. M. Genesereth, *et al.* 1992. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report, Computer Science Department, Stanford University.
7. M. Genesereth and S. Katchpel, Software Agents. *Communication of the ACM*. 37(7), 1994, pp. 48-53.
8. http://fipa.comtec.com.jp/fipa/index.htm.
9. http://www-cia.mty.itesm.mx/~escamila/software/Jess.
10. N.R. Jennings, *et al,* ADEPT: Managing Business Processes Using Intelligent Agents. In *Proceedings of BCS Expert Systems Conference* (ISIP Track). Cambridge, UK. 1996.
11. H.S. Nwana, Software Agents: An Overview. *The Knowledge Engineering Review,* Vol 11 (3), 1996.
12. R. Patil, *et al,* 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)*, Dan Mateo, CA: Morgan Kaufmann, 1992.
13. Y. Peng, T. Finin, Y. Labrou, B. Chu, J. Long, W.J. Tolone, A. Boughannam, Agetn-Based Approach for Manufacturing Integration: the Ciimplex Experience. *Applied Artificial Intelligence*, 13(1-2), 1998. pp. 39-64.
14. W. Shen, D. Xue, and D.H. Norrie, Agent-Based Manufacturing Enterprise Infrastructure for Distributed Integrated Intelligent Manufacturing Systems. In *Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems*. London, UK. 1998, pp. 533-550.
15. M. Tennenbaum, J. Weber, and T. Gruber, Enterprise Integration: Lessons from Shade and Pact. In *Enterprise Integration Modeling*, C. Peter (ed.). Boston: MIT Press, 1993.
16. W.J. Tolone, B. Chu, J. Long, T. Finin, T., and Y. Peng, Supporting Human Interactions within Integrated Manufacturing Systems. *International Journal of Agile Manufacturing*, **1**(2), 1998, pp. 221-234.
17. T. Vollmann, W. Berry, and D. Whybark, *Manufacturing Planning and Control Systems*. Irwin: New York, 1992.

FIGURES

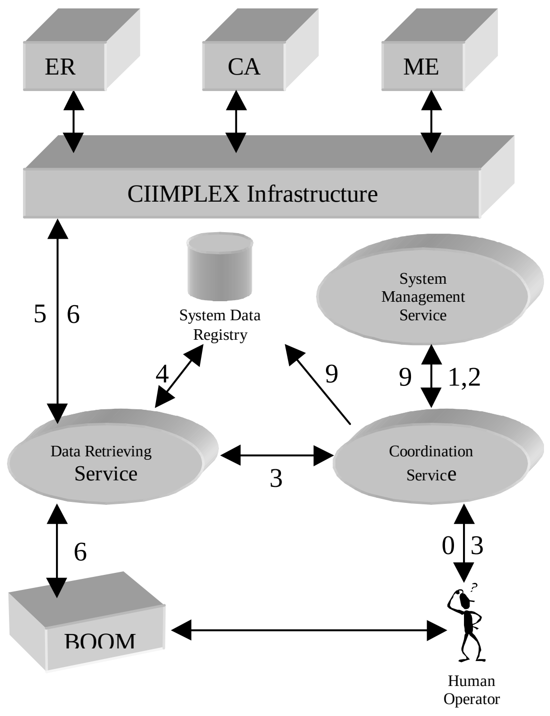
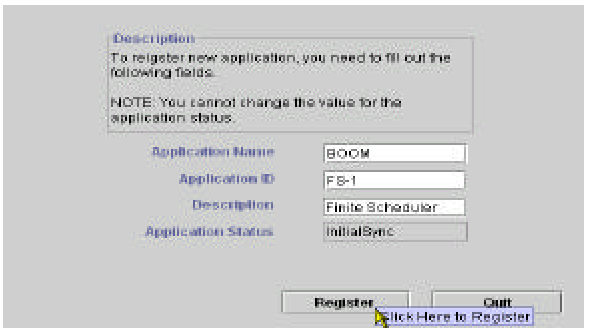


Figure 1 Application Initialization Scenario



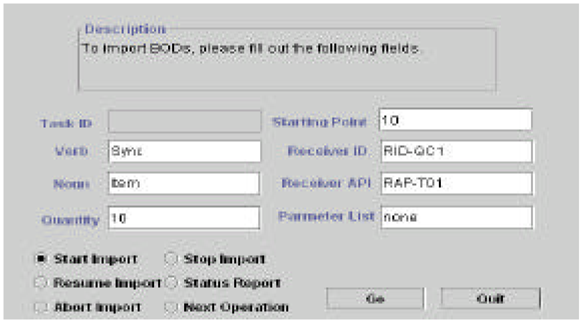
Description

To reigster new application, you need to fill out the following fields.

NOTE: You cannot change the value for the application status.

Application Name	BOOM
Application ID	FS-1
Description	Finite Scheduler
Application Status	InitialSync

(a) Registering new application



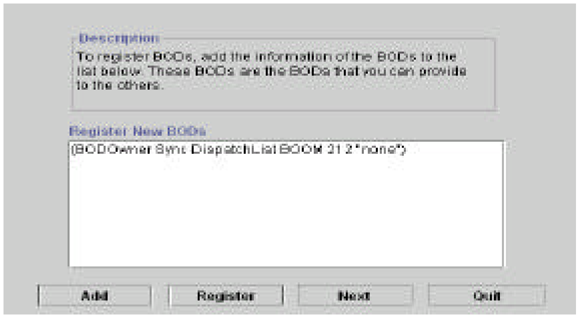
Description

To import BODs, please fill out the following fields.

Task ID		Starting Point	10
Verb	Sync	Receiver ID	RID-QC1
Noun	Item	Receiver API	RAP-T01
Quantity	10	Parameter List	none

[● Start Import] ○ Stop Import
○ Resume Import ○ Status Report
○ Abort Import ○ Next Operation

(b) Importing BODs



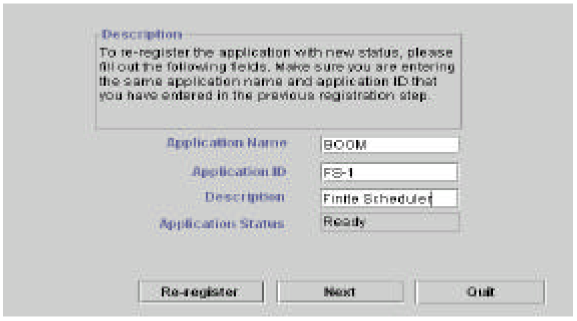
Description

To register BODs, add the information of the BODs to the list below. These BODs are the BODs that you can provide to the others.

Register New BODs

```
(BODOwner Sync DispatchList BOOM 21 2 "none")
```

(c) Registering new BOD's



Description

To re-register the application with new status, please fill out the following fields. Make sure you are entering the same application name and application ID that you have entered in the previous registration step.

Application Name	BOOM
Application ID	FS-1
Description	Finite Scheduler
Application Status	Ready

(d) Final registration of new application

Figure 2. AIA's GUI screen dumps