An Architecture for Multi-Agent COTS Software Integration Systems

Guo-Ming Fang, Jim-Min Lin*, Zeng-Wei Hong, and Kai-Yi Chin

Department of Information Engineering and Computer Science, Feng Chia University

Taichung City 40724, Taiwan

*jimmy@fcu.edu.tw

Received 3January 2007; Revised 25 March 2007; Accepted 31March 2007

Abstract. Commercial Off-The-Shelf (COTS) software products are increasingly used as software components in large-scale systems. We had proposed an approach for distributed COTS software integration by using the concepts of multi-agent system and distributed scripting mechanism. To describe the experience in the COTS software integration and facilitate the reuse of the software integration procedure, this paper presents a multi-agent architecture for the COTS software integration systems. This architecture is of a three-layered structure and is described with the Agent UML (AUML). Since the interaction and internal processing of agents is clearly described in the proposed architecture, programmers may have a guide to build a software system and implement the protocols and behaviors of agents according to the three-layered description. To illustrate the use of the proposed architecture, an example system is also experimented in our study.

Keywords: COTS Software Reuse/Integration, Agent UML (AUML), Multi-Agent Distributed Scripting System (MADSS), Software Architecture

1 Introduction

Software applications are increasingly built with distributed object-oriented technique, such as OMG CORBA [1], Microsoft DCOM [2], and J2EE [3]. These middleware systems [4] provide well-designed component/object models, and well integration mechanisms supporting interfaces to link components/objects together. In addition to the features of an object-oriented system, a multi-agent system could have several advantages, like:

1. A software agent has well social ability [5]. An agent could communicate with human users and accept the delegated tasks. Furthermore, it is also a communicative program that interacts with other programs/agents in speech-acts [6], which means the communication likes human's talk. A complex task could be completed through the cooperation of software agents.
2. A software agent could have mobility. This feature enables a task to be completed remotely. Moreover, some studies [7] have shown that mobile agents could reduce the network traffic in some applications.
3. A software agent with intelligent abilities is potentially suitable for handling sophisticated distributed computations. Some studies [8] indicated that large-scale systems are becoming more and more complex. The systems might consist of lots of software components that interact with others. Object-oriented software development is not the only efficient paradigm for constructing a large-scale software system. Software agents are actually software objects having autonomy and intelligence. Software agents could have better interaction ability than traditional objects and thus suit for building distributed software systems.

Our previous study proposed a multi-agent system, named as Multi-Agent Distributed Scripting System (MADSS) [9], to integrating software applications into a distributed software system. MADSS is aimed to achieve the goal of integrating COTS [10-13] software products or legacy software systems under the distributed heterogeneous environment through the cooperation and interaction of multiple agents. The social ability of agents provides the communication among agents. The mobile agent technology is used to support the remote access. A scripting language was also proposed to help the users to control the behaviors of agents.

This paper is a continued work of MADSS project. MADSS did achieve the goal of COTS software integration with multi-agent paradigm, but have only less discussion on how software integration could be achieved by agents' cooperation. Therefore, the purpose of this paper is to further identify and describe the multi-agent architecture of MADSS. This multi-agent architecture is in a form of 3-layers approach and is represented using AUML (Agent Unified Modeling Language) [14].

* Correspondence author

The rest of this paper is organized as follows. We will briefly describe MADSS project in Section 2. Section 3 will describe the multi-agent architecture of MADSS, which is based on design considerations from the responsibilities of agents and the interactions between agents. Section 4 gives a case study as our demonstration and a guide of using the architecture. Finally, we conclude this research and describe our future works in Section 5.

2 MADSS project

MADSS is basically a distributed software integration system in which software were integrated through agents' cooperation. An MADSS script language was developed as an interface through which a software engineer could drive an agent's behavior. By using this scripting language, a software integrator could perform an application project rapidly through the typeless and command-level attributes [15, 16].

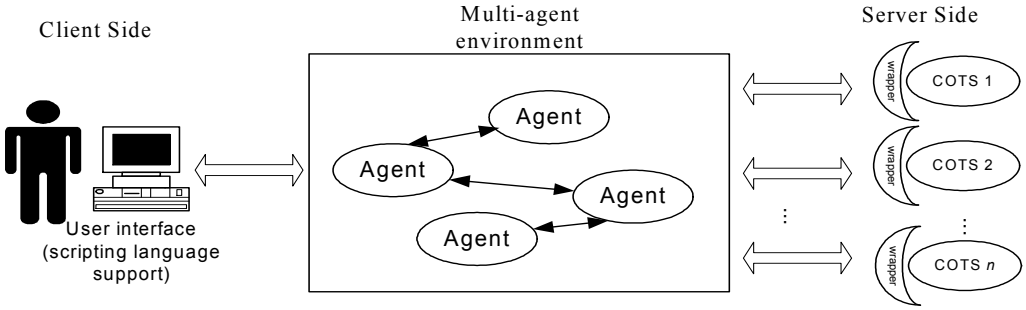


Fig. 1. Conceptual Model of MADSS

MADSS is a typical 3-tier distributed system (see Fig. 1). At the server-side, there exists several distributed wrapped COTS software applications. Through the software wrapper, each COTS software application could expose its services for agent's call. A *service agent* is designated to maintain the interfaces of the wrapped COTS software applications, which reside on the same host. Whenever a new wrapped software application is integrated into the host, the service agent will be responsible for advertising the new services on the facilitator. A facilitator is actually a software agent responsible for interoperating MADSS agents.

At the client side, MADSS uses a client software agent to support a user interface to interact with the user and to receive jobs written in MADSS scripting language. The client agent generates one or more mobile slave agents to perform the integration job. During the execution phase, a slave agent will communicate with a service agent via KQML (Knowledge Query and Manipulation Language) [17] messages. A KQML message encapsulates input parameters to a service agent and then translates them into proper data types of the service agent. Finally, service agent requests the corresponding software applications to execute this job. If a host is overloaded, the service agent would be responsible for suggesting the slave agent to move to other route.

MADSS project overcame two critical issues in agent-based COTS software integration:

1. The black-box-like COTS software applications under MS-Windows and UNIX-like systems were successfully wrapped as programmable and reusable software components [18-20].
2. MADSS successfully demonstrates the feasibility of integrating software by mobile agents.

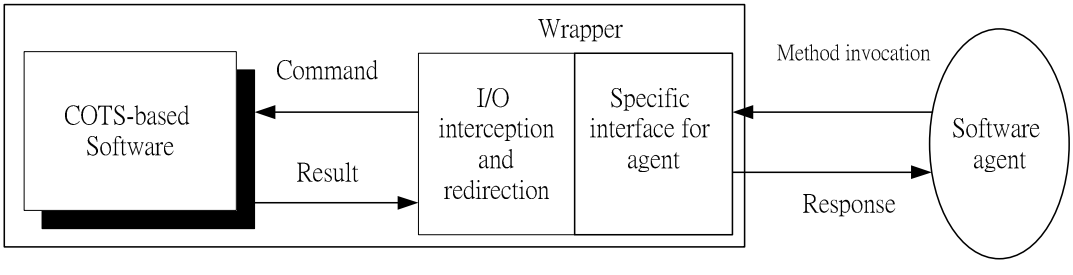


Fig. 2. Wrapper for Reengineering COTS Software Applications

Reengineering COTS software such as MS-Windows applications may suffer from the seldom-available source code. Software wrapper (in Fig. 2) in MADSS uses *I/O interception and redirection* to simulate a COTS application's a sequence of operations, such as command or key events. Therefore, software wrapper could operate a MS-Windows application by sending keyboard events to it or passing input data to the Windows clipboard space. The result could also be captured by clipboard space or other output channels. In Fig. 3, the example program code (a), (b) and (c) represents respectively Win APIs for simulating key events, getting and setting the data in clipboard space. To input key events, the wrapper program must let MS-Windows application get the focus first by using FindWindow() and SetFocus(). After setting the focus to the application, Wrapper program adapted keybd_event() to send keyboard events by assigning the virtual code and scan code of keys. Besides, Wrapper program uses OpenClipboard() and CloseClipboard() to handle the clipboard space in MS-Windows. With the use of GetClipboardData() and SetClipboardData(), Wrapper program can read and write the clipboard space. Moreover, to migrate a MS-Windows application to an agent framework, this MS-Windows application was encapsulated a specific interface the agent could access.

```
(a)
    HWND appHWND = FindWindow ("Title of App", NULL);
    SetFocus (appHWND);
    keybd_event ((BYTE) v_code, (BYTE) s_code,0,0);
    keybd_event ((BYTE) v_code, (BYTE) s_code, KEYEVENTF_KEYUP, 0);
(b)
    HGLOBAL memHND = GlobalAlloc(GHND, strlen(input_str)+1);
    VOID* memPtr = GlobalLock(memHND);
    MoveMemory(memPtr, input_str, strlen(input_str)+1);
    GlobalUnlock(memHND);
    OpenClipboard(NULL);
    EmptyClipboard();
    SetClipboardData(CF_TEXT, memHND);
    CloseClipboard();
(c)
    char *out_buffer=(char*) malloc(BUFFER_SIZE);
    OpenClipboard(NULL);
    HANDLE clipHND = GetClipboardData(CF_TEXT);
    VOID* clipPtr = GlobalLock(clipHND);
    strcpy(out_buffer, (char*)clipPtr);
    GlobalUnlock(clipHND);
    CloseClipboard();
```

Fig. 3. Example Code in Wrapper of MS-Windows Applications

MADSS has been successfully implemented by referring to the concept of software integration through mobile agents. However, more detail description to this experimental multi-agent system is needed to formally represent the design experience. This study will describe the interaction between agents and each agent's internal state uses AUML in next section.

3 Multi-Agent Architecture for MADSS

In order to define the agents and interactions between agents in detail, a three-layer approach of AUML proposed by Odell is adopted as the description language in the proposed multi-agent architecture. At the first layer, the overall interaction protocols of the multi-agent architecture in MADSS are defined as reusable packages. The interactions among agents in protocols are described at the second layer. Finally, the internal agent processing are represented at the third layer.

According to the conceptual model of MADSS, three interaction protocols between agents could be defined. These interaction protocols are also indicated in the first layer description (see Fig. 4).

1. Delegating package. The Delegating package expresses a protocol between a client agent, Facilitator and slave agent. This protocol describes how a client agent delegates tasks to a slave agent and handle the results from the slave agent.

2. Publishing package. The Publishing package expresses a protocol between a service agent and the Facilitator. This protocol describes how a service agent publishes the information of services supported by a wrapped COTS software to Facilitator.
3. Binding package. The Binding package expresses a protocol between a slave agent and service agents. This protocol describes how a slave agent requests for service to a service agent.

The detail interactions among agents in the first layer description will be represented in the second layer description. Here, the Delegating, Publishing and Binding interaction protocols are expressed with extended sequence diagram and depicted correspondingly in Fig. 5, Fig. 6, and Fig. 7 respectively.

In the delegating protocol, a client agent may query a Facilitator about a service. If the query is available, service results will be replied. On retrieving the location information of the service, the client agent will initialize a slave agent and delegate tasks to it for execution. The client agent will not only delegate tasks to the slave agent but also combine these results from the slave agent. If an error happened, the slave agent will response them to the client agent.

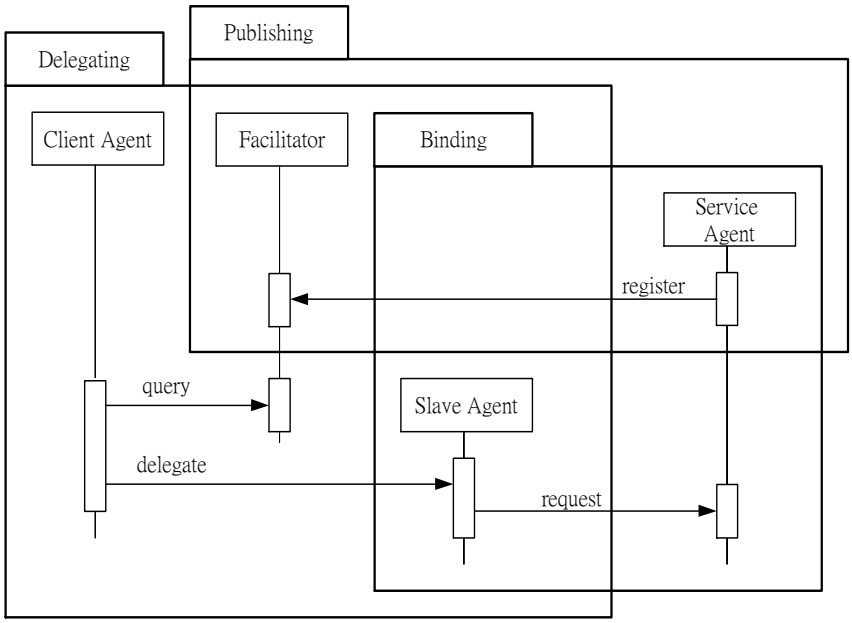


Fig. 4. First Layer Description of the Multi-Agent Architecture in MADSS

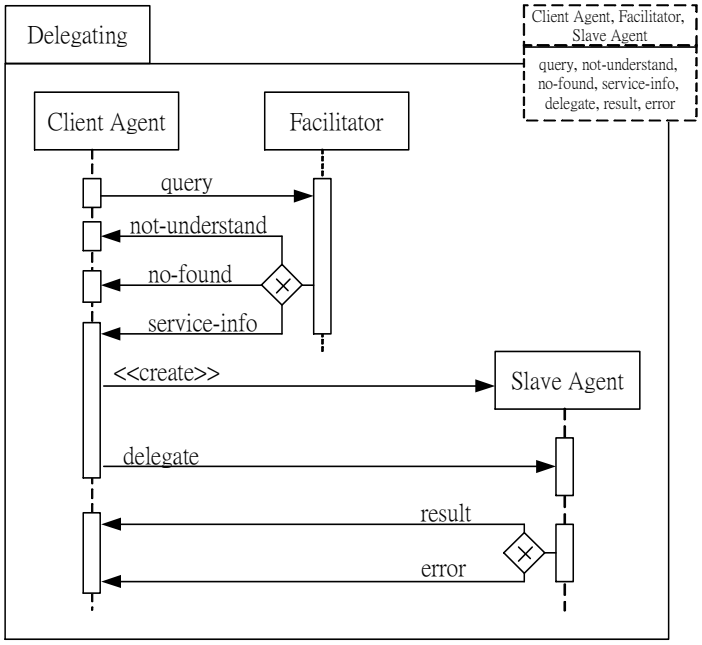


Fig. 5. Delegating Interaction Protocol

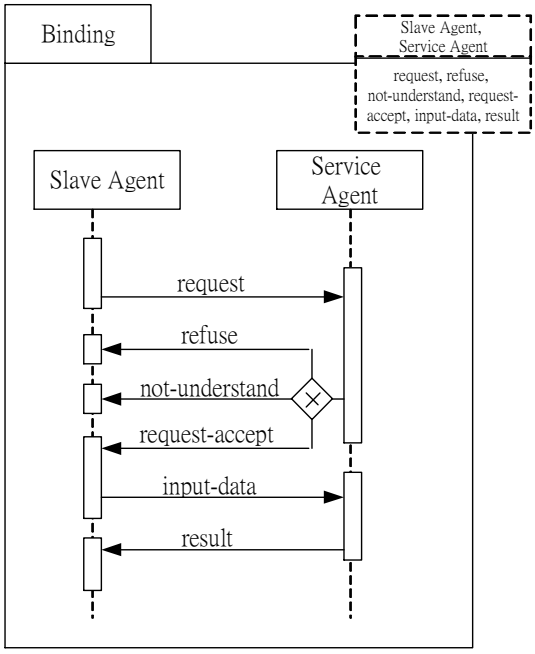


Fig. 6. Binding Interaction Protocol

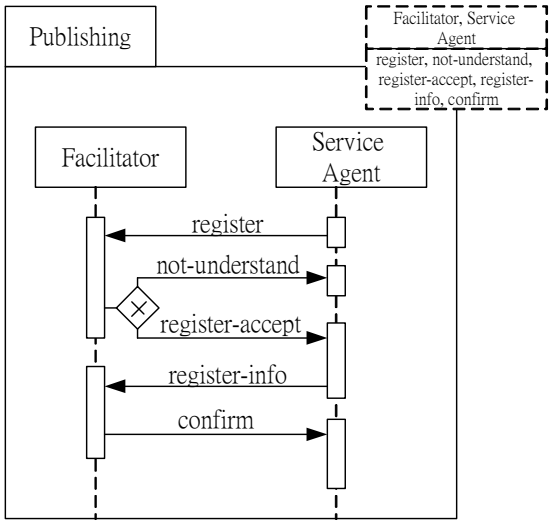


Fig. 7. Publishing Interaction Protocol

In the Binding protocol, a slave agent sends a service request to a service agent. If the service agent accepts the request, the slave agent will send some data to the service agent as the input of the service. After the service is finished, the result will then be replied to the slave agent by the service agent.

In the Publishing protocol, a service agent sends the register request of a service to a Facilitator. If the Facilitator accepts the request, the service agent will send the registration information to the Facilitator. After the processing of registration is finished, a confirmation from the Facilitator will be replied to the service agent.

In the third layer description, activity diagram is adopted to express the internal processing of agents according to related interactions. Hence, the internal processing of the client agent, slave agent, Facilitator, and service agent are depicted correspondingly in Fig. 8, Fig. 9, Fig. 10, and Fig. 11 respectively.

The client agent, which is a stationary agent, serves users at client side. In the internal processing of a client agent, the client agent deals with the content of tasks after a user input the tasks through the user interface. However, some of these tasks may not be handled by the ability of the client agent. Thus, the client agent may make a query to a Facilitator about a service that could serve these tasks. If the service result is replied to the client agent, the client will analyze the result to get the location information of the service agent that provides the service. The location information of the service and the content of these tasks would be delegated to a slave agent. Finally, the client agent may process the resulting message or error message from the slave agent.

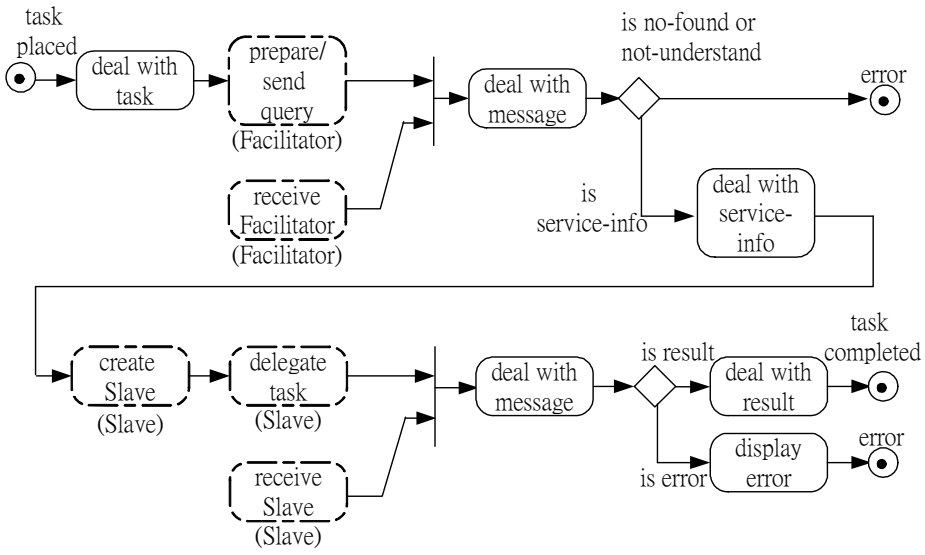


Fig. 8. Internal Processing of Client Agent

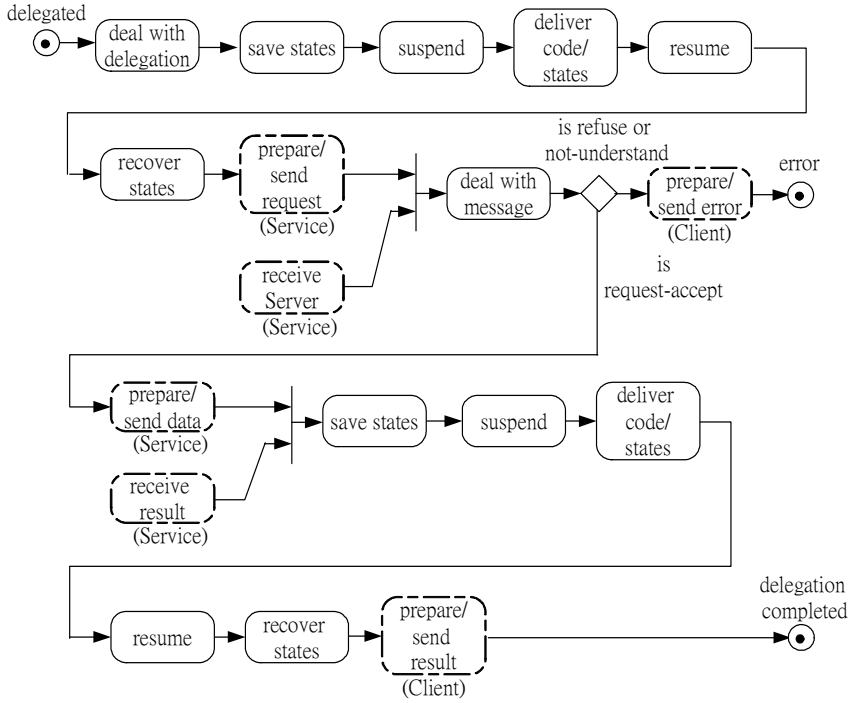
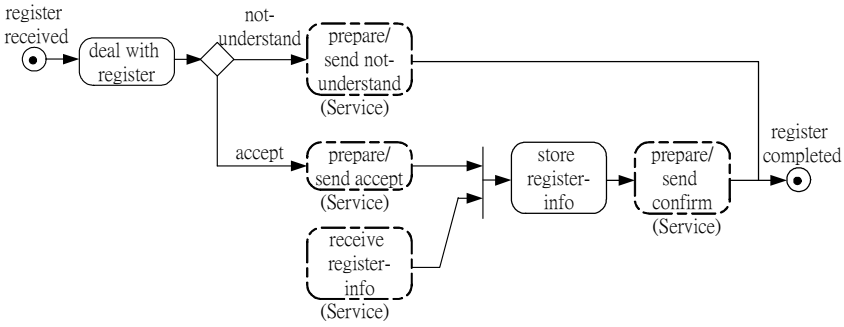


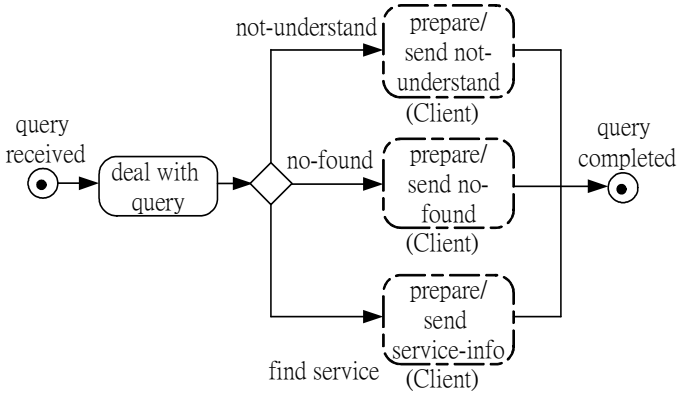
Fig. 9. Internal Processing of Slave Agent

The slave agent, which is a mobile agent, processes the tasks from the client agent. In the internal processing of a slave agent, the slave agent may analyze the delegation from a client agent to get the information of a service, such as service name, service agent name, and server location. After the slave agent migrates to the server side, the slave agent will send a service request to the service agent. If the service agent accepts the request, the slave agent will send input data to the service agent, else an error message will be sent back to the client agent. After getting the result form the service agent, the slave agent migrates back and sends the result to the client agent.

The Facilitator, which is a public stationary agent, provides the directory service. It may handle the service register from a service agent (see Fig. 10 (a)) or query from a client agent (see Fig. 10 (b)). On receiving a request of service registration from a service agent, the Facilitator will analyze the message. If the registration service is available, the Facilitator will send the acceptation of registration to the service agent and receive the service information from the service agent. After the information of the service is stored in the registry, the Facilitator will send a registration confirmation to the service agent. Additionally, on receiving a request of service query from a client agent, the Facilitator will analyze the message also. If the service information is stored in the registry of the Facilitator, the Facilitator will then send the service information back to the client agent.

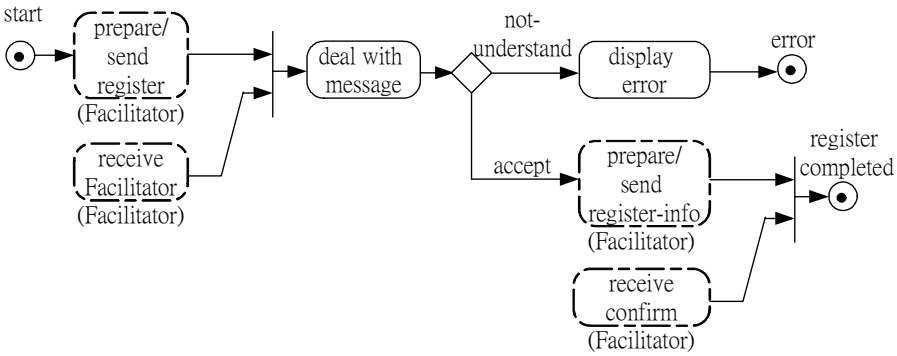


(a) Handle Register of Service from Service Agent

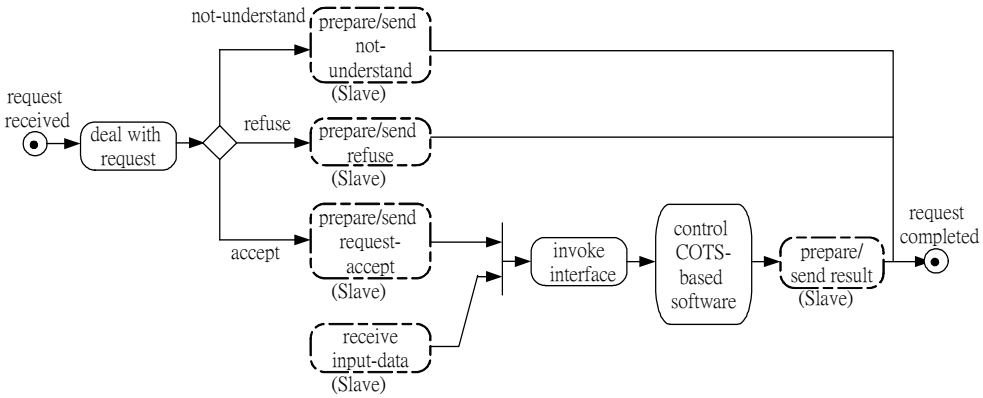


(b) Handle Query from Client Agent

Fig. 10. Internal Processing of Facilitator



(a) Publish the Information of Service



(b) Handle the Request from Slave Agent

Fig. 11. Internal Processing of Service Agent

The service agent, which is a stationary agent at the remote side, manages the COTS software. The internal processing of a service agent could be addressed as follows. The service agent may either publish the service information of a wrapped COTS software as this agent startups (see Fig. 11 (a)) or handle the request from a slave agent (see Fig. 11 (b)). To publish a service, a service agent will send a request of service registration to a Facilitator and waits for the reply from the Facilitator. If the Facilitator accepts the request, the service agent will send the service information supported by a wrapped COTS software back to the Facilitator. In addition, the service agent also handles the service requests from a slave agent. If the request is available and acceptable, the service agent will receive the input data from the slave agent. The input data are the parameters for accessing the programmable interface of the wrapped COTS software by using procedure call or message passing.

4 An Example System: A Graphical Mechanical Part Management System

To demonstrate the feasibility of our proposed architecture, we report on implementing an experimental integrated software system—Graphical Mechanical Part Management System (GMPMS) in this section. There are numerous mechanical parts in various categories and types but they might be alike in name, shape, model, size, and so on. Many non-expert buyers might thus make incorrect orders without assistance from a dealer for more mechanical part information, like precise specification, picture, and layout. Therefore, the purpose of GMPMS is to assist dealers in efficiently managing and querying the mechanical part database with graphical displays of the part layout.

To carry out a GMPMS, two major subsystems are necessary: a database subsystem for storing the mechanical parts data and a graphical display subsystem for displaying the part drawing. To develop these two subsystems from scratch would be a time consuming and costly endeavor. An economic and rapid way is through the software reuse approach. Therefore, AutoCAD 2000 and dBase III Plus are selected to be integrated in the GMPMS. AutoCAD 2000 is a drawing software tool and commonly used to design a mechanical part. The dBase III Plus is a simple database tool for providing the management of mechanical parts and store the command stream of mechanical parts. According to the agents defined in our multi-agent architecture, the GMPMS consists of following five agents.

1. Mechanical part management agent. The mechanical part management agent is a stationary agent at the user side and plays the role of the client agent. It provides a user interface to manage mechanical parts and shows the 2D/3D shape of mechanical parts.
2. Slave agent. Slave agent is a mobile agent and responsible for getting the query conditions of mechanical parts from the mechanical part management agent. It could migrate to the location of the mechanical part information agent to get the information of mechanical parts. It could also be migrated to the location of the shape generation agent to generate the 2D/3D shape of mechanical parts.
3. Mechanical part information agent. The mechanical part information agent is a kind of service agent. It maintains the *queryPart* service to query the information of mechanical parts. In order to provide the service, it handles the operations of dBase III Plus through the simulation of keying a sequence of dBase III commands.
4. Shape generation agent. The shape generation agent is a kind of service agent. It maintains the *generate2dShape* and *generate3dShape* services to generate the 2D/3D shape of mechanical parts. In order to provide these services, it handles the operations of AutoCAD 2000 by inputting the AutoCAD command stream of mechanical parts.
5. Facilitator. The Facilitator provides directory service and maintains the information about the *queryPart*, *generate2dShape* and *generate3dShape* services.

To show the application of our architecture on the system, we give some interaction and internal processing diagrams about the use of *queryPart* service. For invoking *queryPart* service, the interactions among agents in GMPMS are based on the protocols of Delegating, Publishing, and Binding. However, to apply these protocols to our application, some modifications are necessary. Fig. 12, Fig. 13, and Fig. 14 represent the Delegating, Publishing and Binding protocols about *queryPart* service in the GMPMS respectively. In the second layer descriptions about *queryPart* service, some messages in the aforementioned protocols should be modified to fit our application requirements. For example, the message *register-info* in Fig. 7 is replaced with *queryPart-info* in Fig. 13. The message *input-data* in Fig. 6 is replaced with *input-query conditions* in Fig. 14.

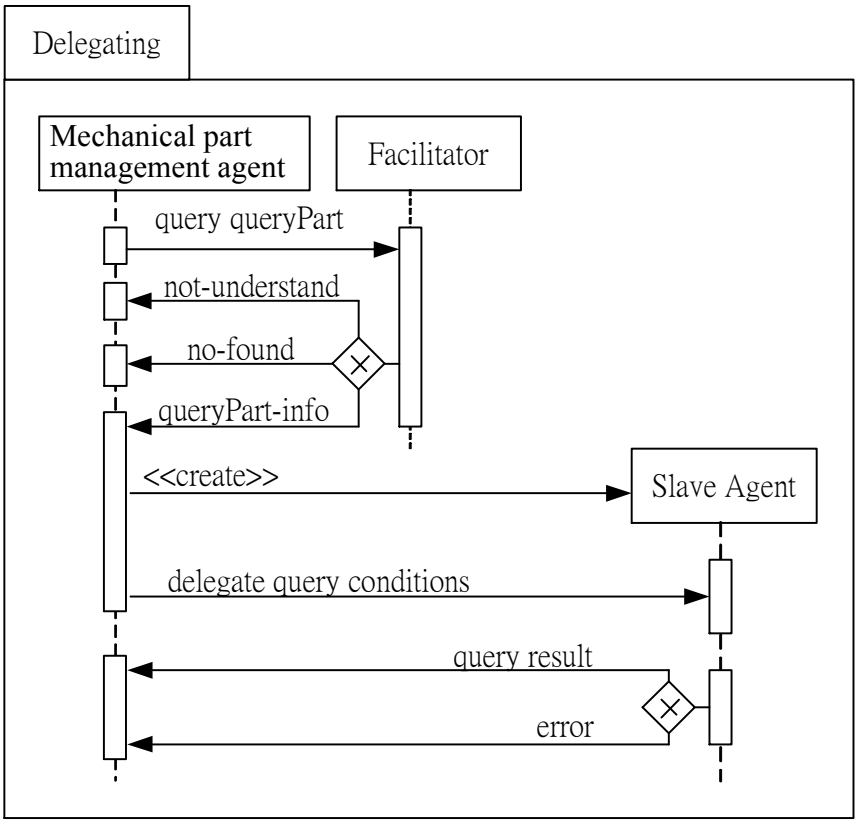


Fig. 12. Delegating Interaction Protocol about *queryPart* Service

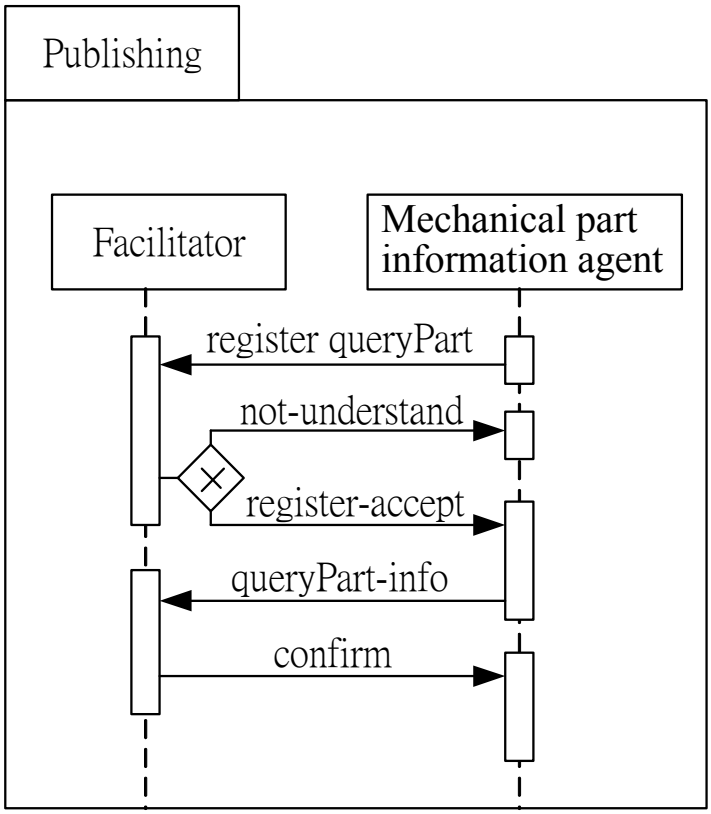


Fig. 13. Publishing Interaction Protocol about *queryPart* Service

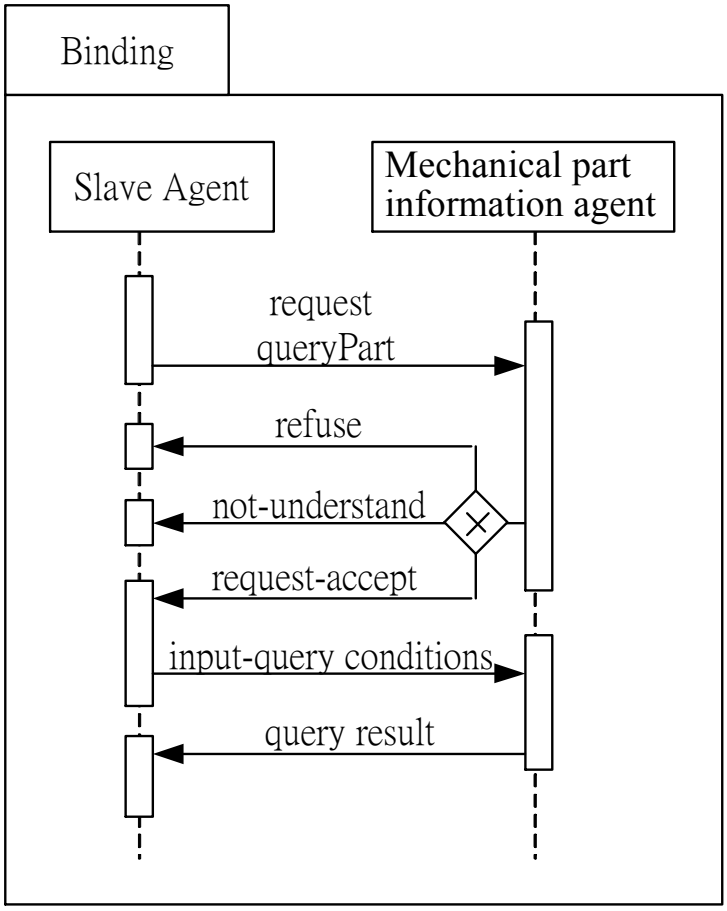


Fig. 14. Binding Interaction Protocol about *queryPart* Service

ask-one
:sender *Slave agent*
:receiver *Mechanical part information agent*
:in-reply-to *S1*
:ontology *queryPart*
:language *String*
:content *model=`WS-1201`*

Fig. 15. Example of KQML message for binding queryPart service

Each message in these protocols may be equal to a KQML messages. For example, when the slave agent arrives the remote side, it communicates with the service agent to get the *queryPart* service through some KQML messages. The Fig. 15 shows the content of the messages *input-query conditions* between the slave agent and the mechanical part information agent through the binding interaction protocol.

The internal processing of these agents about *queryPart* service is based on the third layer description in our multi-agent architecture. The behaviors of mechanical part management agent, slave agent, Facilitator, and mechanical part information agent are illustrated in Fig. 16, Fig. 17, Fig. 18 and Fig. 19 respectively. Some processing blocks in the agent state diagrams should be modified to fit the functions of *queryPart* service also. For instance, the *invoke interface* in Fig. 11 is replaced with *invoke dBase III Plus Wrapper* (see Fig. 19).

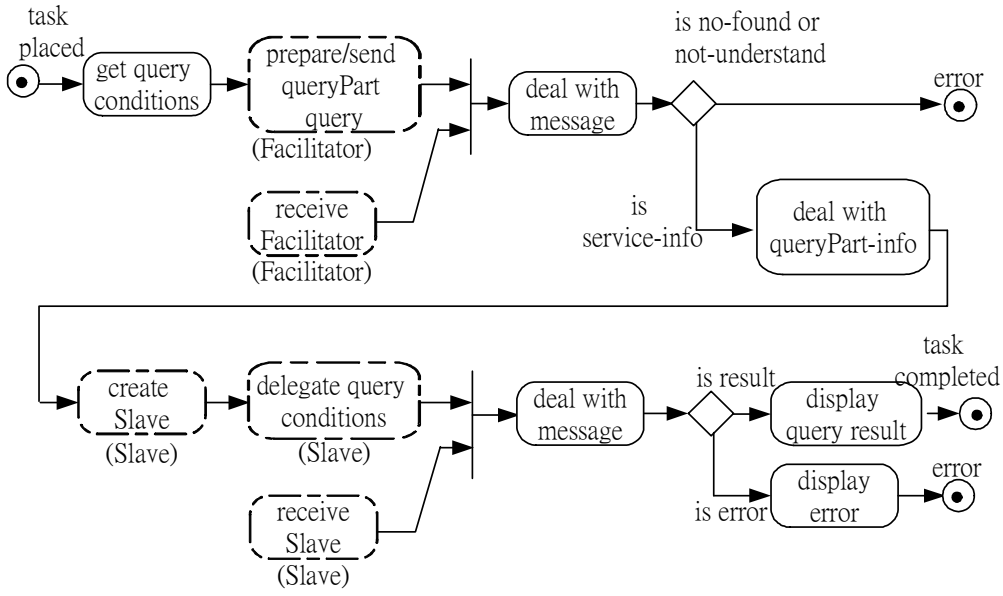


Fig. 16. Internal Processing of Mechanical Part Management Agent about *queryPart* Service

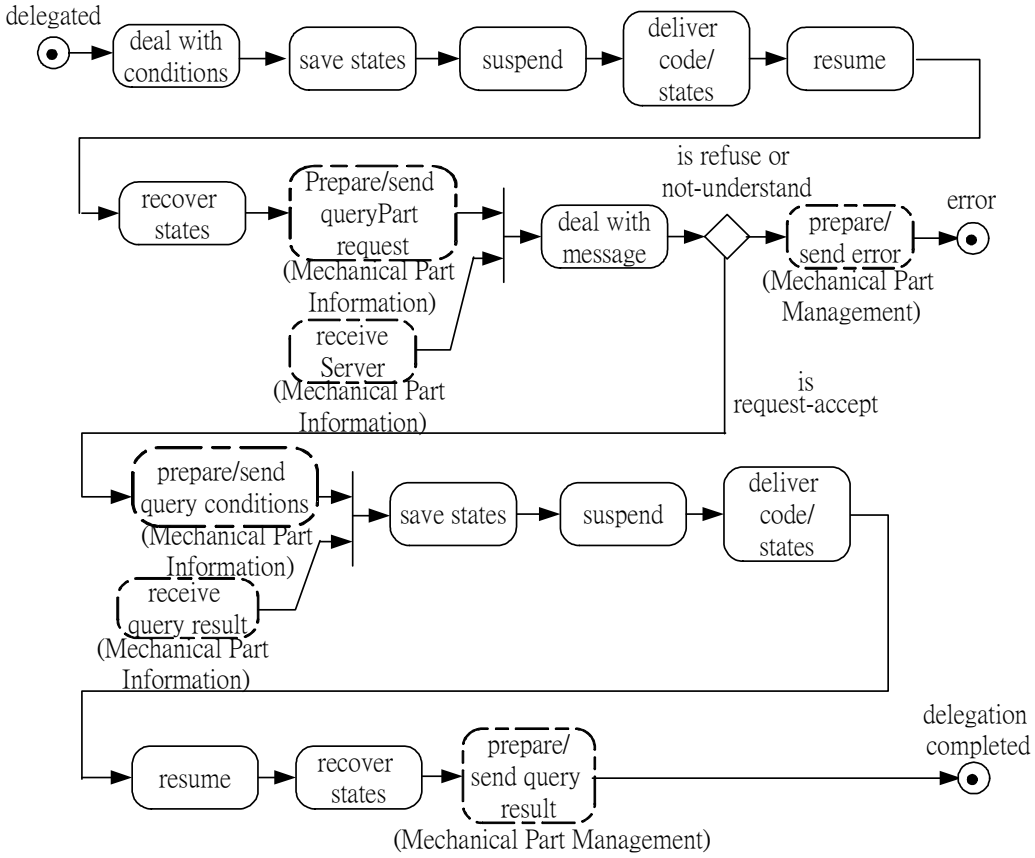
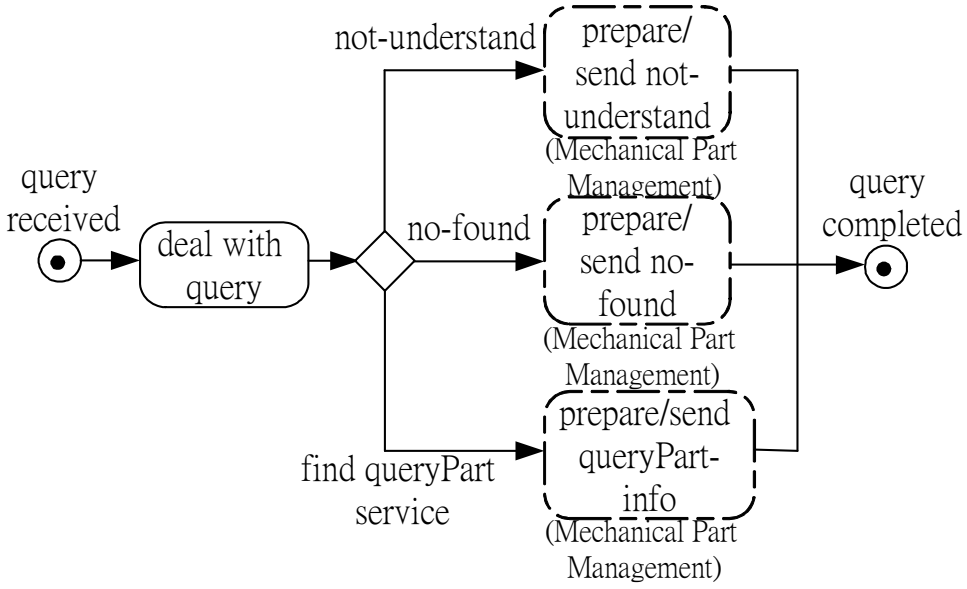
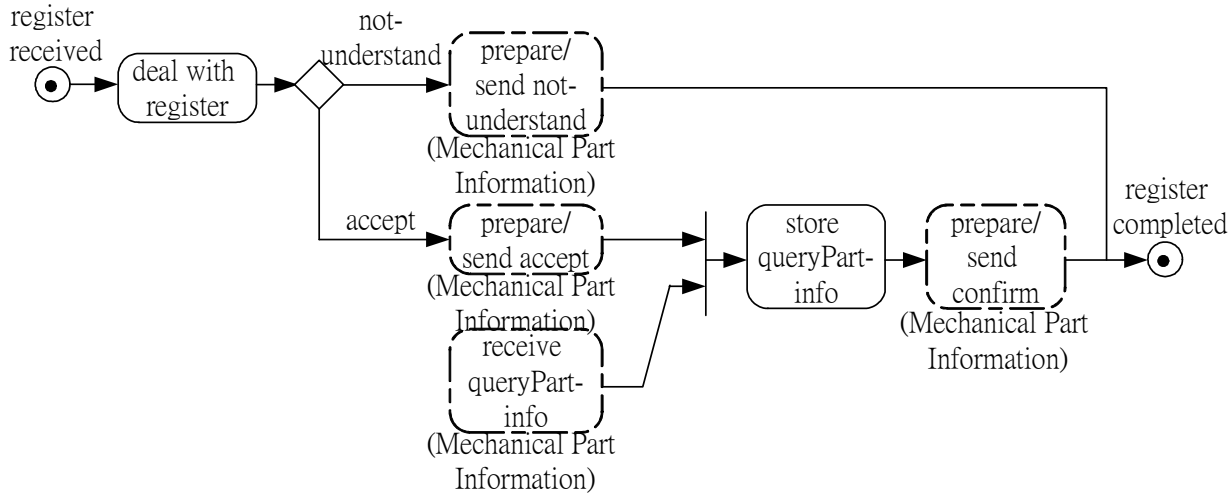


Fig. 17. Internal Processing of Slave Agent about *queryPart* Service

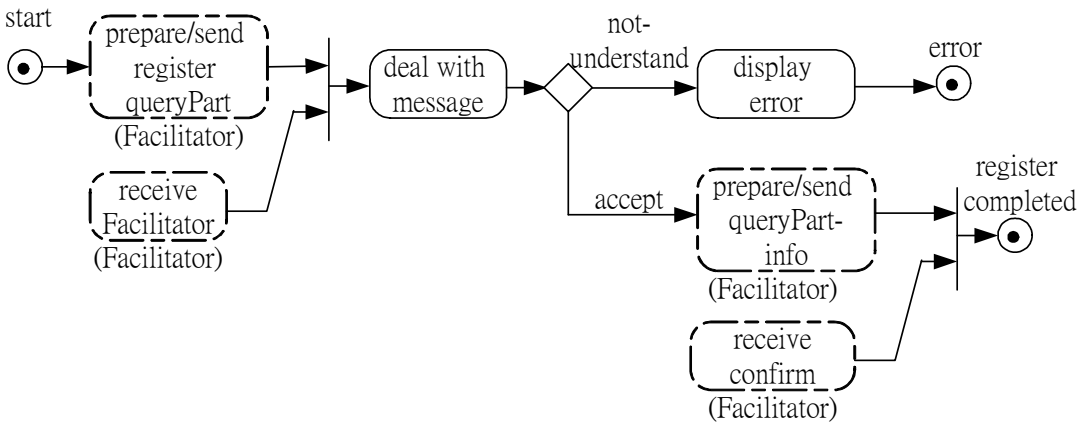


(a) Handle Query from Mechanical Part Management Agent

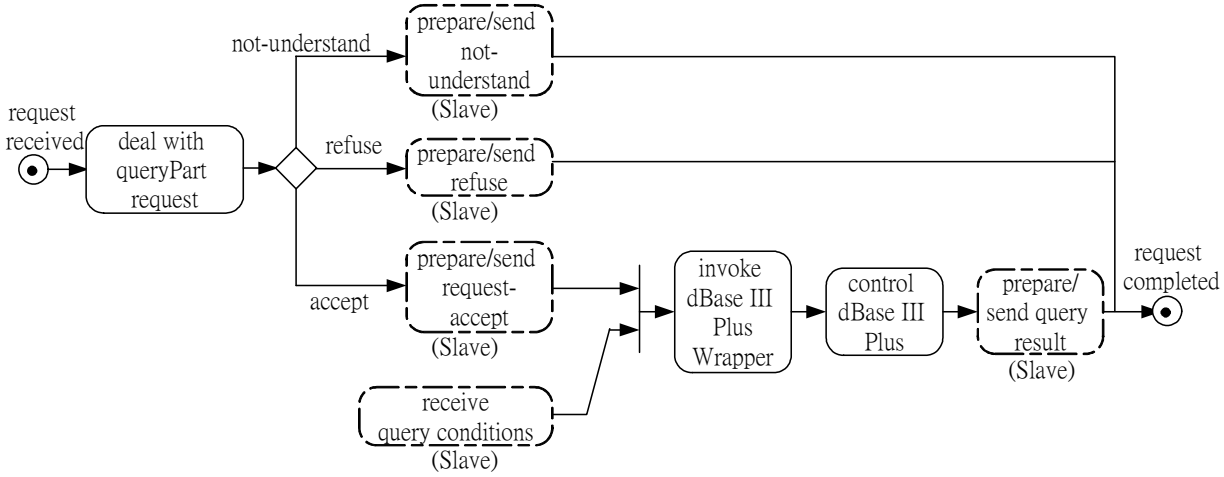


(b) Handle Register of *queryPart* from Mechanical Part Information Agent

Fig. 18. Internal Processing of Facilitator about *queryPart* Service



(a) Publish the Information of *queryPart* Service



(b) Handle the Request from Slave Agent

Fig. 19. Internal Processing of Mechanical Part Information Agent about *queryPart* Service

Each blocks in these internal processing may be implemented to some program codes. For instance, the mechanical part information agent at the remote side may involve some implementation to operate the dBase III. Fig. 20 shows the example codes of controlling dBase III Plus to handle queryPart service. It simulates the key events to input data to dBase III by using the functions of inputString(), hotkey() and key() provided by dBaseWrapper.

```
void queryPart(String conditions) {
  dBaseWrapper.inputString( "use parts" );   //copy data to clipboard
  dBaseWrapper.hotkey(17, 29, 86, 47)   //Ctrl +v
  dBaseWrapper.key(13, 28)   //Enter
  dBaseWrapper.inputString( "copy to result.txt for" + conditions +  "type sdf" );
  dBaseWrapper.hotkey(17, 29, 86, 47)   //Ctrl +v
  dBaseWrapper.key(13, 28)   //Enter
}
```

Fig. 20. Example of queryPart in Mechanical Part Information Agent

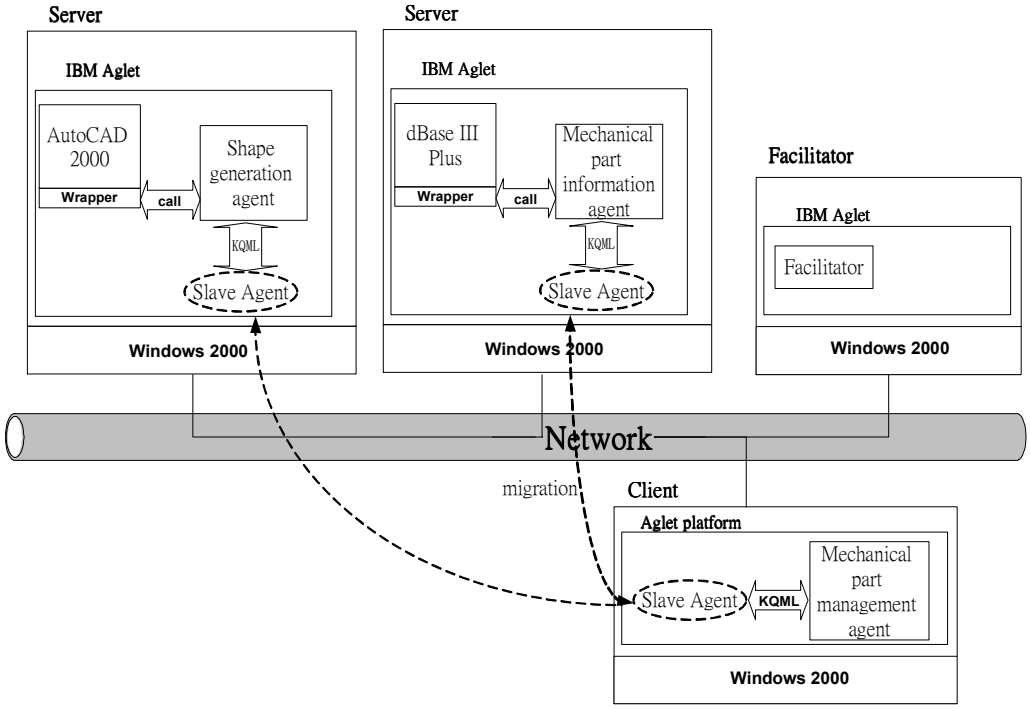


Fig. 21. Overall structure of GMPMS

Fig. 21 illustrates the overall structure of GMPMS. In GMPMS, the mechanical part management agent provides a user interface to interact with the user and queries the Facilitator about the information of related services. By delegating the slave agent to interact with the mechanical part information agent and shape generation agent, the mechanical part management agent could get the information and shape of mechanical parts. Here, the slave agent retracts the AutoCAD command stream in the information of mechanical parts gotten from the mechanical part information agent and uses it as the input of the shape generation agent to generate the shape of mechanical parts.

In the implementation of the GMPMS, all the agents are written in Java language. In order to provide the communication mechanism and the mobility of agents, the IBM Aglet [21] is adopted as the execution environment.

In Fig. 22, the information of mechanical parts in dBase III Plus can be queried by inputting some query string. In Fig. 23, the user interface of querying mechanical parts in GMPMS is shown. By inputting some mechanical part attributes, dealers could get needed mechanical part information in the dBase III Plus. The 2D shape of the indicated mechanical part is shown at the view of top. By using the AutoCAD command stream stored in dBase III Plus as the input, the AutoCAD 2000 would be able to generate the shape of mechanical part. Therefore, dealers would find and show the mechanical parts for customers.

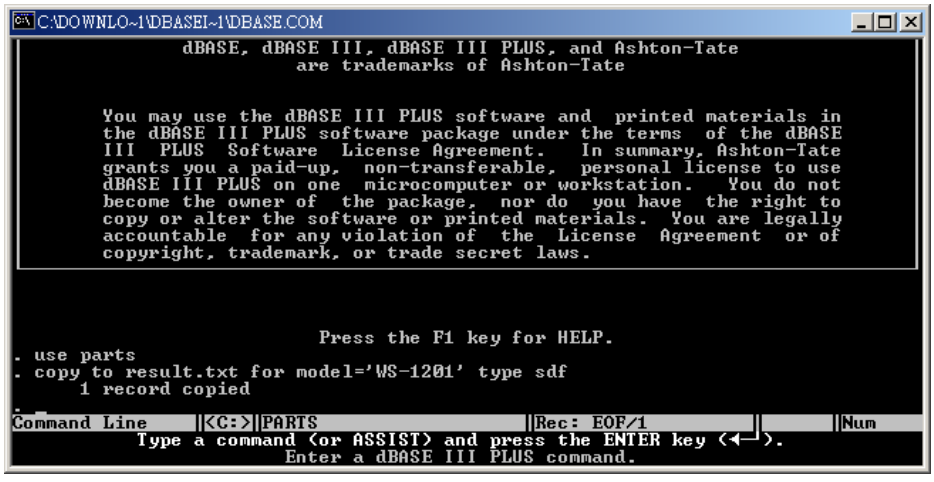


Fig. 22. Wrapped dBase III Plus in GMPMS

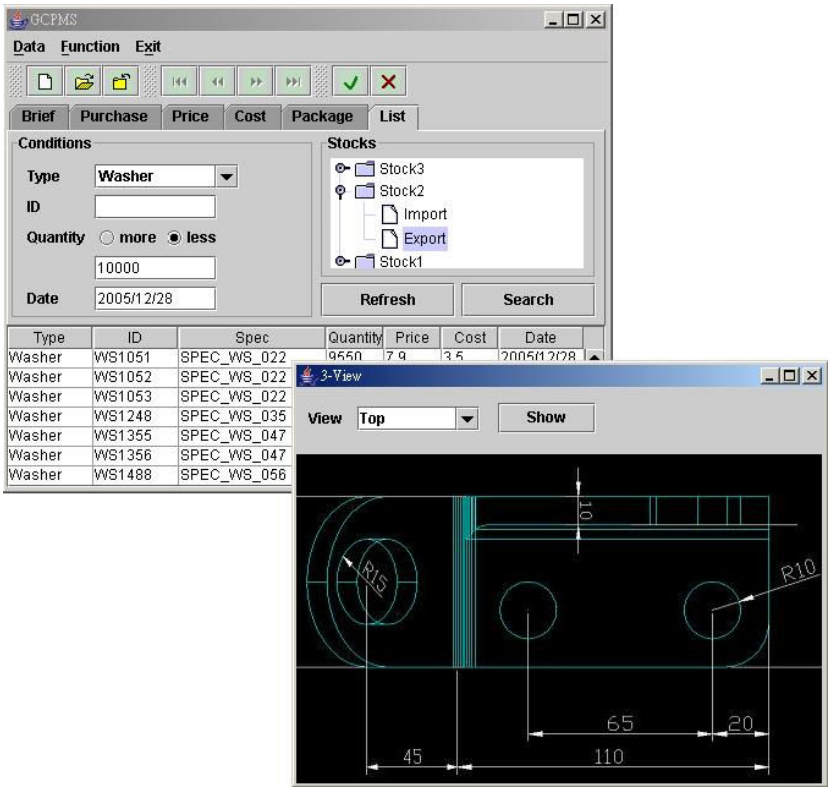


Fig. 23. Screenshot of GMPMS

5 Conclusions

A multi-agent architecture of MADSS is proposed in this paper to incorporate several COTS software by using software agent technology within an integrated software system. To describe the architecture in detail, the three-layer approach of Odell's AUML is adopted as the description language. Therefore, the specification of implementation, including protocol and internal processing, are described. A Graphical Mechanical Part Management System (GMPMS) is also experimented in our study to illustrate the use of the proposed architecture. It integrates the services provided by wrapping dBase III Plus and AutoCAD 2000. The proposed architecture would be helpful to programmers with the knowledge of UML or AUML in having a guide to construct distributed COTS software integration systems with the multi-agent paradigm.

Finally, we conclude several future works to improve the architecture:

1. To define the description of exported COTS service.
2. To enhance a mechanism for discovering the suitable and desired services advertised in the community.
3. To handle the problem of agent's ontology.

References

[1] P. Felber, P. Narasimhan, "Experiences, strategies, and challenges in building fault-tolerant CORBA systems," *IEEE Transactions on Computers*, Vol.53, No.5, pp.497-511, May 2004.

[2] A. Davis, D. Zhang, "A comparative study of DCOM and SOAP," *Proceedings of 4th International Symposium on Multimedia Software Engineering*, pp.48-55, Dec. 2002.

[3] E. Altendorf, M. Hohman, R. Zabicki, "Using J2EE on a large, Web-based project," *IEEE Software*, Vol.19, No.2, pp.81-89, March-April 2002.

[4] S. Vinoski, "Where is middleware," *IEEE Internet Computing*, Vol.6, No.2, pp.83-85, March-April 2002.

[5] Y. B. Peng, J. Gao, J. Hu, B. S. Liao, "Policy-Driven Agent Social," *Proceedings of International Conference on Machine Learning and Cybernetics*, Vol.1, pp.345-350, Aug 2005.

[6] J. R. Koo, "Social commitments in MAS," *Proceedings of the 8th Russian-Korean International Symposium on Science and Technology (KORUS-2004)*, Vol.1, pp.72-74, July 2004.

[7] D. X. Xu, J. W. Yin, Y. Deng, J. H. Ding, "A Formal Architecture Model for Logical Agent Mobility," *IEEE Transactions on Software Engineering*, pp.31-45, 2003.

[8] R. Nicholas. M. W. Jennigns, "Agent-Oriented Software Engineering," *Proceeding of 9th European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, 2000.

[9] J. M. Lin, Z. W. Hong, and G. M. Fang, "MADSS: a Multi-Agent Based Distributed Scripting System, " *Proceedings of 26th Annual International Computer Software and Application Conference (COMPSAC-2002)*, Oxford, UK, pp.581-586, Aug 2002.

[10] T. G. Baker, "Lessons Learned Integrating COTS into Systems," *Proceedings of 1st International Conference on COTS-Based Software System (ICCBSS 2002)*, Orlando, FL, USA, pp.21-30, Feb. 2002.

[11] L. Davis and R. Gamble, "Identifying Evolvability for Integration," *Proceedings of 1st International Conference on COTS-Based Software System (ICCBSS 2002)*, Orlando, FL, USA, pp.65-75, Feb. 2002.

[12] T. Pfarr and J. E. Reis, "The Integration of COTS/GOTS within NASA's HST Command and Control System," *Proceedings of 1st International Conference on COTS-Based Software System (ICCBSS 2002)*, Orlando, FL, USA, pp.209-221, Feb. 2002.

[13] A. Egyed and R. Balzer, "Unfriendly COTS integration – instrumentation and interfaces for improved plugability," *Proceedings of 16th International Conference on Automated Software Engineering (ASE 2001)*, pp.223-231, Nov. 2001.

[14] J. Odell, H. V. Dyke Parunak, Bernhard Bauer, "Extending UML for Agents," *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pp.3-17, 2000.

[15] J. Ousterhout, "Scripting: higher lever programming for the 21st Century," *IEEE Computer*, Volume 31, pp.23-30, March 1998.

[16] J. Ousterhout, "Additional Information for Scripting White Papter," in URL: *http://www.sunlabs.com/people/john.ousterhout/scriptextra.html.*

[17] Y. Liu, C. Xu, W. D. Chen, Y. Pan, "KQML Realization Algorithms for Agent Communication," *Fifth World Congress on Intelligent Control and Automation (WCICA-2004)*, Vol.3, pp.2376-2379, June 2004.

[18] J. M. Lin, Z. W. Hong, G. M. Fang, H. C. Jiau, and W. C. Chu, "Reengineering Windows software applications into reusable CORBA objects," *Journal of Information and Software Technology*, Vol. 46, No.6, pp.403-413, May 2004.

[19] Z. W. Hong, J. M. Lin, H. C. Jiau, G. M. Fang, and C. W. Chiou, "Reengineering Windows-Based Software Applications into Reusable Components using Pattern Language," *Journal of Information and Software Technology*, Vol.48, No.7, pp.619-629, July 2006.

[20] J. M. Lin, Z. W. Hong, G. M. Fang, and C. T. Lee, "A Style for Migrating MS-Windows Software Applications to Client-Server Systems Using Java Technology," *SOFTWARE - Practice & Experience*, Vol.37, No.4, pp.417-440, April 2007.

[21] IBM Aglet, in URL: *http://www.trl.ibm.com/aglets/index_e.htm*.