플러그 앤드 플래이(Plug-and-Play)개념을 이용한 이형 응용 프로그램의 통합 기법

백 순 철[†] 최 중 민[††] 장 명 욱[†††]
박 상 규[†] 민 병 의[††††] 임 영 환[†††††]

요 약

본 논문에서는 한 응용 프로그램의 간단한 접속 및 수행을 통하여, 그 프로그램이 다른 프로그램과 서비스와 정보를 공유할 수 있도록 하는 다중 에이전트 시스템의 개발에 관하여 다룬다. 이를 위하여 3가지 요소를 만들어 사용하였다. 첫번째 요소인 메타 정보는 새로 접속되는 응용 프로그램을 알아보고 이용하기 위해 필요한 정보를 자동으로 구성한 것이다. 이 메타정보는 모양이 다른 응용 프로그램들을 직접 다룰 때 발생하는 복잡성을 줄여준다. 두번째 요소인 에이전트 모듈은 한 응용 프로그램이 다른 응용 프로그램들과 동일한 행동 양식으로 교류하기 위한 제어를 제공한다. 마지막으로 세번째 요소인 에이전트간 교류 메시지는 모양이 다른 응용 프로그램들일 지라도 서로 교류할 수 있도록, 교류 메시지에 대한 공통의 표현 양식을 제공한다. 이러한 세가지 요소를 사용하여, 임의의 응용 프로그램의 간단한 접속 및 수행을 통한 새로운 시스템의 확장 요구를 만족시킨다.

The Integration of heterogeneous applications through Plug-and-Play

Sooncheol Baeg,[†] Joong-min Choi[††], Myoeng-Wuk Jang[†††]
Sang-Kyu Park,[†] Byung-eui Min[††††] and Young-Hwan Lim[†††††]

ABSTRACT

In this paper, we discuss an effort to develop a multi-agent architecture through which heterogeneous applications communicate and cooperate by means of plug-and-play mechanism. Three components are created in order to challenge the plug-and-play mechanism: meta-information, PnP agent module, and ICM. The meta-information is used to automatically set up a suitable configuration for a new plugged-in application, eliminating the need for direct addressing among heterogeneous applications. The PnP agent module is a homogeneous controller that operates on an application to ensure that its activities are coordinated with those of the others within the community, providing a uniform control mechanism. The ICM is a high level communication message that provides a homogeneous communication envelope for all heterogeneous applications. The combination of these three components is used to meet the desire for implementing the plug-and-play mechanism. In this distributed, open architecture, one should be able to simply plug in a new application and it should work.

1. Introduction

There are many thousands of software applications available to computer users today, providing a variety of services and information. Although most of these applications provide their users significant value when used in stand

† 정 회 원 : 한국전자통신연구소 인공지능연구실 선임연구원
†† 정 회 원 : 한양대학교 전자계산학과 조교수
††† 정 회 원 : 한국전자통신연구소 연구원
†††† 정 회 원 : 한국전자통신연구소 인공지능연구실 책임연구원
††††† 정 회 원 : 한국전자통신연구소 멀티미디어연구부 책임연구원
논문접수 : 1995년 8월 7일, 심사완료:1995년11월 1일

-alone applications, there is increasing demand for applications that can cooperate[10]. This cooperation offers expanded opportunities for software reuse and extensibility.

One of the major issues surrounding the software cooperation has been the integration of applications. The integration is not so easy since applications are not constructed with the aim of integration in mind. They are written by different people, at different times, in different languages, and with different interfaces.

A simple way for the integration would be to modify the source codes of the application and re-compile them into a single application. This is too static : Integrated systems should be configurable, that is, it should be able to add, move or remove applications as necessary. However the integrated application would have to be modified and re-compiled whenever one of these changes occurs. This is an aspect of transparency ; applications should be independent of the changes of other applications.

Object-oriented programming is an important programming technology that offers expanded opportunities for software reuse and extensibility. Object-oriented programming shifts the emphasis of software development away from functional decomposition and toward the recognition of units (called objects) that encapsulate both code and data. As a result, programs become easier to maintain and enhance. Despite its promise, penetration of object-oriented technology to major commercial software products has progressed slowly because of certain obstacles. This is particularly true of products that offer only a binary programming interface to their internal object classes (i.e., products that do not allow access to source code).

The first obstacle that developers must confront is the choice of an object-oriented programming language. No binary standard exists for C++ objects, so the C++ class libraries produced by one C++ compiler cannot (in general) be used from C++ programs built with a different C++ compiler. The second obstacle is that, because different object-oriented languages and tool kits embrace incompatible of what objects are and how they work, software developed using a particular language or tool kit is naturally limited in scope. Classes implemented in one language cannot be readily used from another. A C++ programmer, for example, cannot easily use classes developed in Smalltalk, nor can a Smalltalk programmer make effective use of C++ classes. Object-oriented language and tool kit boundaries become, in effect, barriers to inter-operability.

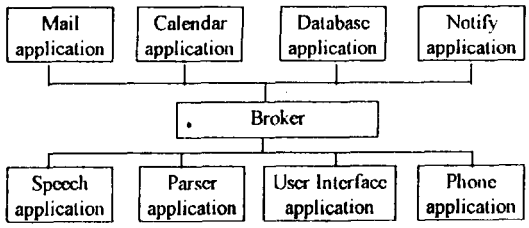
We present an approach to the integration of software applications through a plug-and-play framework. With this framework, we should be able to simply plug in a new application and it should work. The framework automatically sets up a suitable configuration for the new agent. In the framework, a server application named broker waits for requests from client software applications. The requests involve the connection to the broker (plug-in), the fulfillment of goals, and several services. A client application requests a connection to the broker, posts a query to the broker and performs a task assigned by the broker. For every request posted to the broker, the broker determines which application is capable of performing the request, and then delegates it to the application.

The determination of the broker is based on an internal database, called meta-information. The meta-information (information about applications) contains all the associated data such as the location, the name and the capabilities of the applications that are currently connected. In addition to this meta-information, a homogeneous controller and a high level inter-agent communication message are essential to support the plug-and-play (or PnP) mechanism. The homo-

geneous controller, named PnP agent module, operates on an application to ensure that its activities are coordinated with those of the others within the community, providing a uniform control mechanism. The inter-agent communication message, called ICM, provides a homogeneous communication envelope for all heterogeneous client applications. The framework, named PMAF(Plug-and-Play based Multi-Agent Framework), employs three components (the meta-information, the PnP agent module and the ICM) to support dynamic plug-and-play of many heterogeneous applications.

2. Plug-and-Play Example

To illustrate how multiple stand-alone applications can be cooperatively used, we now consider an example. Let's assume that a system administrator wants to be notified when his system security is broken. This scenario involves the cooperation of the eight stand-alone applications in the network in (Fig. 1). Through the user interface application, the query "When mail about security alert arrives, get it to me by telephone" is posted to the broker, asking for the resolution of speech recognition. The user interface application accepts spoken or handwritten natural language queries from the user. The query will be answered by the speech application, and then a new query for the translation of strings into inter-application expressions. The broker will then route this type of query to the natural language parser application that is responsible for the translation. When the translated query from the user interface application has been posted to the broker, "when mail about security alert arrives", the broker knows that the mail application should be the one to respond to this type of question. The mail application constantly monitors incoming mails, testing the condition.



(Fig. 1) An example of the cooperation among applications

Once the condition has been met, the mail application posts the following actions to the broker:"get the message to the user by telephone." The notification application will read this and respond. In order to send this information to the user, it must first know where the user is. A goal "whereis(user, Location)" is posted to the broker, and the broker will route this query to the calendar application. Once the location has been returned, the notification application then asks the broker to find out the phone number for that given location. This information is answered by the database application. Finally once the notification application knows where and what phone number the user is. A query is posted to the broker to contact the user,"send particular message by telephone", and of course the phone application can response this type of queries.

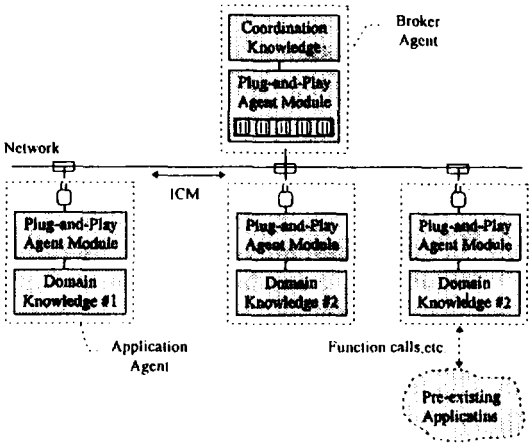
A variety of systems can be designed for this scenario. We could build a stand-alone application from scratch including all the capabilities of the eight applications rebuild all the existing software in a common environment, so that they all share common representations, reasoning mechanisms and knowledge semantics construct a framework into which these existing applications can be incorporated (with minimal modifications) which allows them to communicate and cooperate with each other.

We chose the final option and in this paper we address some issues of the framework. In the next section, we describe the framework ar-

chitecture to support this plug-and-play mechanism.

3. Plug-And-Play Architecture

An agent is the smallest unit that can plug into the PMAF. The agent communicates with external applications via an inter-agent communication message, sharing services and information with other applications (or agents). By converting applications into agents, we can have them cooperate in the PMAF. An agent is able to respond to all messages defined by the ICM, and uses the message to invoke the services of other agents. Agents monitor local and remote events, such as messages from other agent, database updates, OS, or network activities, determining for themselves the appropriate time to execute.



(Fig. 2) A plug-and-play architecture

The PMAF is a blackboard-based model [6, 7]. This model allows individual "client" agents (or application agents) to communicate by means of messages posted on a blackboard controlled by a "server" blackboard agent (or broker) as shown in (Fig. 2).

There are two kinds of agents in the PMAF: a broker agent and an application agent. The broker agent (or simply, broker) is an agent that coordinates the cooperation and the communication among application agents. An application agent is an entity that plugs into the broker and then plays. The application agent performs domain specific operations, which are eventually mapped into an application's capabilities. An application agent can readily plug into the broker or unplug from the broker. Plugging into the broker, the application agent sends its name and capabilities. The broker then constructs the meta-information with the data just sent from the application agent. The meta-information contains useful hints for the broker to figure out what application agents are currently plugged in and what goals they are capable of solving.

When attempting to solve a goal, an application agent can either post a general request to the broker expecting some application agents to resolve it or specify an application agent to solve the goal. In the former case, the broker determines which application agent(s) should be responsible for the request and route it to the most appropriate application agent(s). Application agents respond to requests delegated by the broker that eventually originate from another application agent or the user's request. Communication and interaction among application agents take place solely through the broker.

3.1 Components

The architecture to support plug-and-play agents consists of the following four components: Plug-and-Play Agent Module (PnP agent module): the PnP agent module is a homogeneous controller that operates on a domain knowledge to ensure that its activities are coordinated with those of the others within the community, providing a uniform control mechanism. This module establishes connections among agents. The PnP agent module in the broker side

creates a socket into which application agents are plugged. The PnP agent module in the application agent side then requests connection to the broker. When the broker accepts the connection request, plug-in process is completed. The PnP agent module interprets incoming ICM, passes extracted mission to the domain knowledge and constructs outgoing ICM with the mission's result.

Inter-agent Communication Message (ICM): the ICM provides a homogeneous communication envelope for all heterogeneous application agents. Agents use the ICM to invoke other agent's services or provide their own services.

Coordination Knowledge: This knowledge is designed to coordinate the interactions among application agents. According to the interpreted ICM, part of the coordination knowledge is used to perform appropriate actions. These actions involve constructing the information about currently connected application agent, determining the proper recipient of a task and returning solutions to the originating application agent. The information about application agent is essential to the plug-and-play mechanism. The meta-information is used to determine the proper recipient of a task, eliminating the need for direct addressing among heterogeneous applications.

Domain Knowledge: This knowledge contains application agent's domain specific operations. This knowledge can be constructed from scratch, by interrogating applications data files, or through function calls provided by the applications. In addition, other application agents capabilities can be used in building the domain knowledge.

The block diagram in (Fig. 2) shows how the four components of the PnP architecture work together. The broker must at least be executed in advance providing a socket into which application agents plug. The PnP agent module completes the plug-in process performing the sequence of network connection, and advertising the application agent's name and its capabilities to the broker. The broker constructs the meta-information through the combination of the application agent's name, its network address and its capabilities. The meta-information will be consulted to determine proper recipient of a task.

Incoming messages from the broker are interpreted by the PnP agent module. The PnP agent module maps the interpreted message into the application agent's local operation contained in the domain knowledge. After executing the local operation, the PnP agent module returns the results if any to the broker by using the ICM.

3.2 Plug-and-Play Algorithm

The following steps are performed by two agents: the broker and an application agent:

a) The broker first builds a socket into which application agents will plug.

b) An application agent requests the connection to the broker. When the request is accepted by the broker, the application agent sends its name and capabilities to the broker.

c) The broker constructs the meta-information by using the data sent from the connected application agent. Whenever a new application agent plugs into the broker, the broker updates the meta-information. Whenever a disconnection of an application agent happens, the broker deletes all the associated meta-information.

d) Both the broker and the application agent iterate following steps.

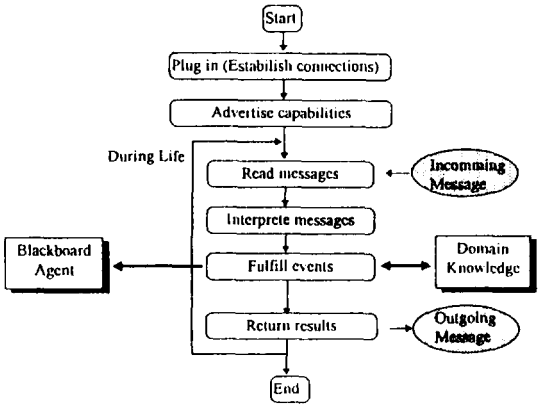
  i) interpret incoming messages from other agents, extracting goals or command to execute.

  ii) examine the domain knowledge or coordination knowledge to perform appropriate actions.

iii) execute appropriate actions and return the results to the originating agent.

4. Convention

So long as an application program abides by the convention of the PMAF, it does not matter how the agent is implemented: the application can readily plug into the agent community sharing its services and information with other applications. The convention includes the PnP agent module's control flow (agent behavior), the description of the meta-information and the ICM.



(Fig. 3) The behavior of an agent

4.1 PnP Agent Module

Because all the agents are independent processes, they go through birth, life, and death (Fig. 3). At birth, application agents are instantiated with specific capabilities and plug into the broker. During their lives, they go through a continuous cycle of reading messages (inter-agent communication), unwrapping the messages to extract events to perform, examining the events against triggers to produce new events, fulfilling events, and returning the results.

While fulfilling the events, application agents utilize their domain knowledge or take advantage of other application agents capabilities by posting subgoals to the blackboard. The PnP agent module is responsible for these control tasks.

So long as a program abides by the details of the communication convention implemented in the PnP agent module, it does not matter how it is implemented. For example, electronic mail programs from different platform, data format, and capabilities can inter-operate with other agents through the PnP agent module.

The behavior of agents is, in principle, quit simple. After connecting to the broker and reporting the application agents capabilities, each PnP agent module iterates the following steps at regular intervals:

(1) read the current ICM, and unwrap the wrapper layer to extract a message content.
(2) examine the trigger base. Execute the capability corresponding to the message content. The capability can be found either in the performer or in the domain knowledge.
(3) construct and send the outgoing message.

The PnP agent module is at the heart of the PMAF and has been used in the development of cooperating multi-agent systems. This module is based upon the philosophy of providing generic interface with multi-agent systems. By including the PnP agent module into their applications, developers can make their applications agent-aware. The separation of the domain and control knowledge into the domain knowledge and the PnP agent module respectively, allows preexisting systems to be incorporated into the multi-agent community with relatively few modifications, and allows the control knowledge to be reused in a number of applications.

4.2 Meta-information

The first step for the plug-and-play is to make individual application agents visible to the broker. Agent names should be assigned to individual agents. Agents use their names as identi-

fiers when connecting with the broker. Messages are sent to named agents. Individual agents are responsible for notifying the broker of their own capabilities so that the broker can find out which application agent is able to handle a task. The broker must store information about the application agent capabilities associated with the agent's name. This meta-information (or information about agents) is used to determine the proper recipient of a task, eliminating the need for direct addressing among heterogeneous application agents.

A BNF description of the meta-information is as follows: (In accordance with standard conventions, { } denotes repetition of zero or more times. [ ] denotes repetition of zero or one time.)

```
⟨meta-informations⟩ ::= '['⟨m-knowledge⟩
                        {, ⟨m-knowledge⟩}']'
⟨m-knowledge⟩       ::= (⟨agentName⟩, '['⟨capability⟩
                        {, ⟨capability⟩}']')
⟨agentName⟩         ::= ⟨alphanumeric-str⟩
⟨capability⟩        ::= ⟨capabilityName⟩[(⟨parame
                        ter⟩{,⟨parameter⟩})]
⟨capabilityName⟩    ::= ⟨alphabet⟩{⟨alphanumeric⟩}
⟨parameter⟩         ::= ⟨alphanumeric-str⟩
                        | ⟨variable⟩
⟨alphanumeric-str⟩  ::= ⟨alphabet⟩
                        {⟨alphanumeric⟩}
⟨alphanumeric⟩      ::= ⟨alphabet⟩ | ⟨digit⟩
⟨variable⟩          ::= ⟨capital⟩⟨alphanumeric⟩
⟨alphabet⟩          ::= a | b | c | ··· | y | z
⟨digit⟩             ::= 0 | 1 | 2 | ··· | 8 | 9 | '—' |
                        '(' | ')' | ','
⟨capital⟩           ::= A | B | C | ··· | Y | Z
```

For example, the intuitive reading of the meta-information (schedule, whereis(User, Location)) is "the broker knows that schedule agent is able to answer where the user is".

The capabilities known to the broker are high level logical operations that are mapped by the application agents to one or more its own local operations. From the programmer's point of view, the capabilities can be seen as APIs through which programmers can make use of other application's operation. Mapping an application's operations into the capabilities is the basic step of making the application an application agent. Then, it is required that its meta-information be public to the broker. The meta-information is a uniform representation of applications' various forms of the operations. This uniformity resolves the heterogeneity among applications.

4.3 Inter-agent Communication Message

The PMAF agents share information and services with other agents by communicating. Agents use the ICM, a high level communication message that provides a homogeneous communication envelope for all heterogeneous agents. When using the ICM, an agent transmits messages composed in Prolog-like syntax, providing the familiar syntax and rich semantics of first order predicate logic, wrapped in an ICM.

The ICM is conceptually a layered message consisting of the wrapper layer, the primitive layer and the message layer.

The wrapper layer encodes a set of features to the message that describe the lower communication parameters, such as the identity of the sender agent, and a wrapper indicating the message is for inter-agent communication.

Agents communicating with each other require a well-known set of conventions. This set of conventions comprises a protocol that must be implemented at both ends of a connection. This protocol is implemented in the primitive layer that determines the kinds of interactions one can have with another agent. This primitive signifies that the message is a query, a command, or any other mutually agreed upon speech act [11, 12]. It also signifies how the sender would like any reply to be delivered (i.e., what primitive will be followed).

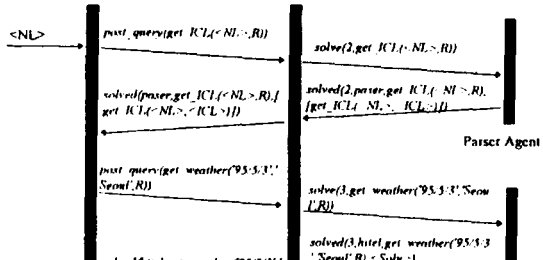
The message layer is the actual content of the

ICM. This layer may contain a goal to solve or a command for controlling agents.

The following is the syntax of the ICM:

```
ICM              ::=term(⟨sender-agent⟩,  ⟨protocol-
                     msg⟩ | ⟨msg-cnt⟩)
⟨sender-agent⟩::=⟨agentName⟩
⟨protocol-msg⟩::=post-query '('[⟨agentName⟩
                     ','](⟨goal⟩')' | solve'('⟨id⟩','⟨goal⟩')'
                     | solved '('⟨id⟩',' ⟨solver⟩','⟨goal⟩','
                     ⟨results⟩')'
⟨goal⟩           ::=⟨capability⟩
⟨results⟩        ::='['⟨msg-cnt⟩']'
⟨msg-cnt⟩        ::=⟨capability⟩ | ⟨commands⟩
                     | ⟨message⟩
⟨commands⟩::=⟨alphanumeric-str⟩
⟨message⟩::=⟨alphanumeric⟩⟨alphanumeric⟩
```

The ICM supports seven basic primitives: post-query, solve, solved, add-trigger, register-solvable-goals, read-bb, write-bb. Post-query is used to post a query to the broker. This primitive causes the broker to determine appropriate application agents that may handle this query, to send the agents a new ICM composed of solve primitive, and to add a trigger. The application agents receiving the ICM with the solve primitive perform the goal and return the results. All the results are conveyed using solved primitive. This primitive denotes that the content of the message layer is the solution to a query. Add-trigger is used to let an agent do



<NL> = "get the weather in Seoul on May 3 1995"
<ICL> = get_weather('95/5/3', 'Seoul', S1)
<Solu> = get_weather('95/5/3', 'Seoul', fine)
<NL_O> = "The weather is fine"

(Fig. 4) An example of interactions among agents

something whenever a condition is satisfied. Register-solvable-goals allows agents to send their capabilities to the broker. Finally, read-bb is for reading data from the broker's blackboard and write-bb for writing data to the blackboard. (Fig. 4) shows an example of an agent's interactions by means of these primitives.

Communicating through the ICM instead of direct function calls allows agents to resolve the heterogeneity. The ICM is constructed and interpreted by the PnP agent module.

4.4 Trigger

To the activities of control mechanism, we add trigger handler to enhance the desired characteristic of dynamic control. Triggers that eventually direct the behavior of agents can be added and removed dynamically by a local agent or remote agents. Fundamental to the notion of triggers is the occurrence of events. Triggers are defined to correspond to specific events. The occurrence of an event that matches a trigger of event type suggests that some agent should perform some specific action responding to the event. The action part of the matching trigger becomes a new event to be processed. For example, lets assume that an application agent posted the query to the broker "what is johns phone number?". The broker would determine appropriate application agents and route the request to them. Not knowing when the answer will be arriving, the broker sets a trigger indication that when solved message arrives, the solutions should be sent back to the asking agent:

trigger(event, when, solved(id-1, phone(john,N), Solutions),
forward(askingAgent, id-1, phone(john, N), Solutions)

Trigger base contains this trigger. Each incoming ICM is examined against the triggers of event group in the trigger base. The message,

solved(id-1, phone(john,N), [phone(john, 593 2)]) matches the trigger and causes the broker to forward the solutions to the originating agent. This trigger handling eliminates the need for waiting all the time for a reply and allows the broker to perform some other tasks. The triggers of test group execute periodically their conditional part to monitor changes such as mail spools or databases, and the action part of the matching trigger is posted to the broker as an event. Incoming events can take an arbitrary form. The only caution is that for an event to be useful for triggers there must be a corresponding trigger.

The following is the syntax of the trigger:

```
⟨trigger base⟩ ::='['⟨trigger⟩, ⟨trigger⟩']'
⟨trigger⟩      ::=trigger'('⟨group⟩','  ⟨type⟩','⟨condi-
                   tion⟩','⟨new-event⟩')'
⟨group⟩        ::=event | test | data
⟨type⟩         ::=when | whenever
⟨condition⟩    ::=⟨msg-cnt⟩
⟨new-event⟩    ::=⟨msg-cnt⟩
```

The triggers of when types are removed from The trigger base whenever used, but those of whenever types are not removed.

5. Experience with the Prototype

The development of the PMAF encompasses two distinct efforts: development of the PMAF core and constructing a new application agent either from scratch or by making pre-existing application the PMAF-aware. All the PMAF core components have been implemented. The implementation of the core has been developed in Prolog on Sun Sparcstations running Sun OS 4.1. We have another version implemented in C on PCs running Microsoft's WindowsNTTM. We have demonstrated the scenario described in section 2. All the application agents have been developed in Quintus Prolog or C on Sun Sparcstations and PCs, except for the pen/voice user interface agent, which is implemented in Basic programming language on a PC and a PDA platform. The communication is based on TCP/IP. Almost every system provides an interface for network communication based on this protocol and it therefore provides a good basis for communication. The blackboard architecture has been ported both to UNIX and WindowsNT-TM.

Dynamic plug-and-play was one of the fundamental design principle of the PMAF. Not all system components of multi-agent systems are known at design time. We can expect a developer to provide a new application agent capable of new functionality in the future. The PMAF architecture supports a flexible system in which application agents may be added, removed, or replaced at any time without making changes to other programs. The broker keeps all the information concerned with the application agents currently connected, and deletes the associated information when the disconnection of an application agent happens. If the broker receives a request for an application agent's capability, it only has to look for the application agent of the required capability and send the request to it. Further, the addition or deletion of an application agent only requires a simple update to the broker's database once the required local method has been implemented.

User interface is not "hard-wired" in our implementation. The PMAF is able to link to a variety of user interfaces that use the ICM protocols and have the PnP agent module's behavior. Therefore, not only the user interface agent can be connected and disconnected dynamically like other application agents across a network, but a variety of interfaces such as X windowsTM-based or WindowsTM-based can coexist. Users can carry user interfaces and get other application agents services at any place where he can use phone line or TCP/IP network.

A broker may itself be a client in a hierarchy of brokers; if none of its applications can solve a particular goal, this goal may be passed further along in the hierarchy [8]. Following Gelerntner's LINDA model [9], blackboard systems themselves can be structured in a hierarchy, which could be distributed over a network.

The creation of an application agent should be simple. We present an example of an application agent program in Prolog (Fig. 4). The agent's name and its capabilities are a prerequisite. The PnP agent module extracts these data from the predicates agentName and solvable respectively, and uses them in connecting to the broker and in advertising the agent's capabilities. The do-event routines contain real program codes for the capabilities defined in the solvable predicate. These routines can be constructed from scratch, by interrogating applications data files, or through function calls provided by the applications. Furthermore, other application agent's capabilities can be used in building the do-event routines. In (Fig. 5), the predicate others make this possible. This predicate allows the application agent to post a request and wait until the solution arrives. This feature is useful since an application agent's capabilities can be reused by other application agents.

```
:-[agent].     /*Load the PnP agent module*/
agentName(calendar).  /*Define the name of this
            agent*/
solvable([     /*Define the capabilities of this agent*/
        get-free-time-slot(-Date, -Person, -FreeTime),
        where(-Person, -Place, -WithPerson)]).
/*Application agent Specific functionality*/
do-event(-, get-free-time-slot(Date,
          Person, FreeTime)):-
          get-appointment-file(Person, CalendarFile),
          get-free-time-C(CalendarFile, Date, FreeTime)
          , !.
do-event(-, where(Person, Place, WithPerson)):-
          get-appointment-file(Person, CalendarFile),
          others(get - current - date - and - time(Date,
          Time)),
          get-appointment(CalendarFile, Date, Time, Ap-
          pointment),
          parse-appointment(Appointment,   WithPerson,
          Place).
.....
/*Interface with C programming language*/
foreign-file(cal.o, [get-free-time-C]).
foreign(get-free-time-C, c, get-free-time-C(+string,
+string, -string)).
```

(Fig. 5) An example of an application agent program

6. Related Work

Although the issues underlying the design of software integration based on the plug-and-play remain largely unexplored, there is an increasing awareness that applications in the future will be communicate with others to share services and information by means of relatively simple plug-and-play.

One of these related bodies of research is KQML [3, 2], a new language and protocol for exchanging information and knowledge. This wok is part of a large effort, the ARPA Knowledge Sharing Effort that is aimed at developing techniques and methodology for building large-scale knowledge bases that are sharable and reusable. KQML can be used as a language for an application program to interact with an intelligent system or for two or intelligent systems to share knowledge in support of cooperative problem solving.

However, KQML was not defined by a single research group for a particular project. It was created by a committee of representatives from different projects, all of which were concerned with managing distributed implementations of systems. The representatives did not share a common communication architecture. As a result, KQML does not dictate a particular system architecture, and several different systems have evolved.

ARCHON [1, 5, 4] project has developed a general purpose architecture that can be used to facilitate cooperative problem solving in in-

dustrial applications. By representing skills, interests and goals of its acquaintances an agent is able to specifically involve others in its own problem solving objective. An agent decides which skills should be executed locally and which should be delegated to others.

Each agent in the ARCHON contains representations of other agents in the community in terms of their skills, interests, current status of workload etc. However, this approach does not address the communication overload that might happen whenever an agent connects or disconnects. Since an agent contains representations of other agents, the communication could increase as agents plugged into or unplugged from the community.

7. Conclusions and Future Work

Three components were employed in order to challenge the plug-and-play architecture: meta-information, PnP agent module, and ICM. The meta-information is used to determine the proper recipient of a task, eliminating the need for direct addressing among heterogeneous agents. The PnP agent module is a homogeneous controller that operates on an application agent to ensure that its activities are coordinated with those of the others within the community, providing a uniform control mechanism. The ICM is a high level communication message that provides a homogeneous communication envelope for all heterogeneous agents. The combination of these three components is used to meet the desire for implementing plug-and-play architecture.

The PnP agent module is a domain-independent reusable module. It is based upon the philosophy of providing generic interface with agent systems. By including the PnP agent module into their applications, developers can make them agent-aware. Since the PnP agent module assumes the burden of inter-operation, application programmers are relieved of this responsibility and can construct their programs without having to learn the details of other programs in the runtime environment. As a result, applications become easier to maintain and enhance.

A variety of applications are available to computer users today. These applications can be used as the domain knowledge of agents. The domain knowledge of an agent may be the interface between the existing applications and the agent community. There are three ways of accessing the functionality of the underlying application. The first is through the manipulation of files (for example, mail spool, calendar data files) and the second is through calls to an application's API interface (e.g., SDK for Microsoft MailTM). Finally, the third is through a scripting language, or through interpretation of an operating system's message events (Apple Event or Microsoft Windows Messages).

In order that an application agent be activated, its capabilities need to be mapped into terms understood by ensemble of application agents, and also by users. However, this advertising knowledge representation can lead to conflicts among definitions. We intend to develop API description Tool, with which the application agent developer describes the services provided by that application agent. The tool will produce mappings of expressions in the ICM into representations that can be merged into a common whole.

References

[1] T.Wittig, N.R. Jennings and E.H. Mamdani. ARCHON - A Framework for Intelligent Co-operation, In IEE-BCS Journal of Intelligent Systems Engineering - Special Issue on Real-time Intelligent

Systems in ESPRIT, 1994.

[2] D.Kuokka, L.Harada. On Using KQML for Matchmaking. In Proceedings of the First International Conference on Multi-Agent Systems, pages 239-245, San Francisco, California, June 12-14, 1995.

[3] T. Finin, D. McKay, R. Fritzson, and R. McEntire. KQML: an information and knowledge exchange protocol. In International Conference on Building and Sharing Very Large-Scale Knowledge Bases. December 1993.

[4] C. Roda, N. R. Jennings, and E. H. Mamdani. The Impact of Heterogeneity on Cooperating Agents. In Proceedings of AAAI Workshop on Cooperation among Heterogeneous Intelligent Systems. Anaheim, Los Angeles, 1991.

[5] N. R. Jennings and T. Wittig. ARCHON: Theory and Practice. In N. M. Avouris & L. Gasser (eds.), Distributed Artificial Intelligence: Theory and Praxis, pages 179-196. Kluwer Academic Publishers, 1992.

[6] R. S. Engelmore, A. J. Morgan, and H. P. Nii. Introduction. In R. Engelmore, T. Morgan, editors, Blackboard Systems, pages 1-22. Addison-Wesley Publishing Co., Menlo Park, CA, 1988.

[7] Victor R. Lesser, Rovert C. Whitehair, Daniel D. Corkill, and Joseph A. Hernandez. Goal Relationships and Their Use in a Blackboard Architecture. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors, Blackboard Architectures and Applications, pages 9-26. Academic Press Inc., San Diego, CA, 1989.

[8] Philip R. Cohen, A. Cheyer, M. Wang, and S. C. Baeg. An Open Agent Architecture. In Working Notes of the AAAI Spring Symposium on Software Agents, pages 1-8. Menlo Park, CA, March 1994.

[9] D. Gelernter. Mirror Worlds. Oxford University Press, New York, 1993.

[10] Michael R. Genesereth and Steven P. Ketchpel. Software agents. CACM, 37(7):48-53, July 1994.

[11] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. In Artificial Intelligence, 42:213-261, 1990.

[12] Philip R. Cohen and Hector J. Levesque. Persistence, intention and commitment. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, Intentions in Communication, pages 33-69. MIT Press, Cambridge MA, 1990.

백 순 철

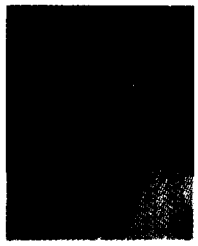
1986년 연세대학교 전자공학과 졸업(학사)
1988년 연세대학교 전자공학과 졸업 (석사)
1988년~현재 한국전자통신연구소 인공지능연구실 선임연구원
1993년~94년 미국 SRI International(International Fellow)
관심분야:에이전트 시스템, 전문가 시스템

최 중 민

1984년 서울대학교 컴퓨터공학과 졸업(학사)
1986년 서울대학교 컴퓨터공학과 졸업(석사)
1993년 State University of New York at Buffalo, Computer Science 졸업(박사)
1993년~95년 한국전자통신연구소 인공지능연구실 선임연구원
1995년~현재 한양대학교 전자계산학과 조교수
관심분야:인공지능, 에이전트 시스템, 지식 표현 및 추론

장 명 욱

1990년 고려대학교 전산과학과 졸업 (학사)
1992년 한국과학기술원 전산학과 졸업 (석사)
1992년~현재 한국전자통신연구소 연구원
관심분야:에이전트 시스템, 패턴 인식



박 상 규

1982년 서울대학교 컴퓨터공학과 졸업(학사)
1984년 한국과학기술원 전산학과 졸업(석사)
1984년~87년 대림산업 전산실 근무
1989년~현재 한국과학기술원 전산학과 박사과정 수료
1987년~현재 한국전자통신연구소 인공지능연구실 선임연구원
관심분야:에이전트 시스템, 정보 검색, HCI

민 병 의

1982년 한양대학교 졸업(학사)
1984년 한국과학기술원 전기 및 전자공학과 졸업(석사)
1992년 한국과학기술원 전기 및 전자공학과 졸업(박사)
1984년~1987년 대림산업기술연구소
1987년~현재 한국전자통신연구소, 인공지능연구실 실장
관심분야:멀티미디어시스템, 에이전트

임 영 환

1977년 경북대학교 수학과 졸업(학사)
1979년 한국과학기술원 전산학과 졸업(석사)
1985년 Northwestern 대학 전산학과 졸업(박사)
1979년~82년 한국전자기술연구소 선임연구원
1983년~85년 Argonne Lab. 연구원
1985년~현재 한국전자통신연구소 멀티미디어연구부 책임연구원
1993년~94년 SRI International(International Fellow)
관심분야:멀티미디어, 에이전트 시스템