Model-Based Development of Intelligent Communicating Agents for Multi-Agent Systems[1]

Haiping Xu and Sol M. Shatz

Department of Computer Science

The University of Illinois at Chicago

Chicago, IL 60607

Email: {hxu1, shatz}@cs.uic.edu

Abstract

An intelligent communicating agent has both the properties of intelligent agents and communicating agents, which are autonomy, reactivity, proactiveness and sociability. In this paper, we propose a model-based approach to designing and implementing intelligent communicating agents for multi-agent systems (MAS). As background knowledge for our approach, we first briefly introduce the agent-oriented G-net model, which is based on the G-net formalism. This formal agent model serves as both a specification and a high-level design for the tool kit called ADK (Agent Development Kit) that supports developing intelligent communicating agents for multi-agent systems. Then the architectural design and detailed design of intelligent communicating agents, which closely follow the agent formal model, are described. As a potential solution for automated software development, we summarize the procedure to generate a model-based design of intelligent communicating agents. Finally, to illustrate an application built on ADK, we present an air-ticket trading example.

Keywords: agent-oriented G-net model, intelligent communicating agent, multi-agent system, agent development kit (ADK), model-based development.

1. Introduction

[1] This material is based upon work supported by the U.S. Army Research Office under grant number DAAD19-01-1-0672, and the U.S. National Science Foundation under grant number CCR-9988168.

The development of agent-based systems offers a new and exciting paradigm for production of sophisticated programs in dynamic and open environments, particularly in distributed domains such as web-based systems and electronic commerce. Though there have been significant commercial and industrial research and development efforts underway for some time, developments based on formal agent frameworks are rare. In this paper, we present a development approach, including design and implementation, for intelligent communicating agents for multi-agent systems (MAS) [27]. The approach is based on a formal agent model introduced in earlier work [4][5] and subsequently described in more details, including examples of model checking for design properties [31]. The agent model serves as both a specification and a high-level design for agent implementation, and it supports design modularization and inheritance. To bridge the gap between modeling and implementation, we highlight a system that provides a full class-library for the domain of intelligent communicating agents for multi-agent systems. We call the development system ADK (Agent Development Kit). The proposed formal agent model, called agent-oriented G-net model, is based on the G-net formalism [2][3], which is a type of high-level Petri net [1]. The significance of this model is that it explicitly supports asynchronous message passing among agents [4], and it supports inheritance for functional units defined in its internal structure [5]. The functional units in this model not only include methods, as in the case of object-oriented paradigm, but also include *Message Processing Units (MPU)*, which are functional units defined for asynchronous message passing. In addition, the agent-oriented G-net model can be translated into more "standard" forms of a Petri net for design analysis, such as deadlock detection and model checking [31].

Previous efforts on narrowing the sizable gap between agent formal models and agent-based practical systems can be summarized into three categories. In the first case, some researchers aim at constructing directly executable formal agent models. For instance, Fisher's work on Concurrent METATEM has attempted to use temporal logic to represent individual agent behaviors where the representations can be executed directly, verified with respect to logical requirement, or transformed into some refined representation [6]. Vasconcelos and his colleagues have tried to provide a design pattern for skeleton-based agent development [23], which can be automatically extracted from a given electronic institution. The electronic institutions have been proposed as a formalism with which one can specify open agent organizations [24]. These types of work seem to be an ideal way for seaming the gap between theories

and implemented systems; however, an implementation automatically derived from a formal model tends to be not practical. This is because a formal model is an abstraction of a real system, and thus an executable formal model ignores most of the components and behaviors of a specific agent. Therefore, as stated in paper [28], executable models based on formalisms, such as temporal logic, are quite distant from agents that have actually been implemented, and at least for the time being, the gap between an executable formal model and a practical agent implementation is still very large. In the second case, researchers use agent theories or agent formal models as conceptual guidelines for agent implementation. Examples of such work are listed as follows. JAM is a hybrid intelligent agent architecture that draws upon the theories and ideas of the Procedural Reasoning System (PRS), Structured Circuit Semantics (SCS), and Act plan interlingua [7]. Based on the BDI theories [8], which models the concepts of beliefs, goals (desires), and intentions of an agent, JAM provides strong goal-achievement syntax and semantics with support for homeostatic goals and a much richer, more expressive set of procedural constructs. The JACK intelligent agent framework by Agent Oriented Software brings the concept of intelligent agents into the mainstream of commercial software engineering and Java [9]. JACK is designed as a set of lightweight components with high performance and strong data typing. Paradima has been implemented to support the development of agent-based systems [10]. It relies on a formal agent framework, i.e., Luck and d'Inverno's formal agent framework [11], and is implemented by using recent advances in Java technology. Though all of the above agent developments rely on formal agent models, the relationships between the implementations and their underlying formal agent models are loosely coupled.

As the third approach for bridging the gap between agent formal models and agent-based practical systems, we use a formal agent model as an agent specification and high-level agent design. In particular, we use the agent-oriented G-net model to define the agent structure, agent behavior, and agent functionality for intelligent communicating agents. A key concept in our work is that the agent-oriented G-net model itself serves as a design model for an agent implementation. We will see that our architectural design of intelligent communicating agents closely follows the agent-oriented G-net model, and the detailed design and implementation of ADK satisfies the requirements specified by the agent-oriented G-net model. By supporting design reuse, our approach follows the basic philosophy of *Model Driven Architecture (MDA)* [12] that is gaining popularity in many communities, for example UML.

The rest of this paper is organized as follows. In Section 2, we briefly review the agent-oriented G-net model, which has been previously proposed [5] and subsequently described in more details for design analysis [31]. We also discuss the role of ADK, in serving as a bridge between the formal agent model and the agent implementation platform. In Section 3, we describe the architectural design and detailed design of intelligent communicating agents. Section 4 summarizes the procedures to design and implement intelligent communicating agents for multi-agent systems, and uses an air ticket trading example to illustrate the derivation of an application using the ADK approach. The generality of the example supports the notion that our model-based approach is feasible and effective. In Section 5, we provide conclusions and our future work.

2. A Framework for Agent-Oriented Software

2.1 G-Net Model Background

A widely accepted software engineering principle is that a system should be composed of a set of independent modules, where each module hides the internal details of its processing activities and modules communicate through well-defined interfaces. The G-net model provides strong support for this principle [2][3]. G-nets are an object-based extension of Petri nets, which is a graphically defined model for concurrent systems. Petri nets have the strength of being visually appealing, while also being theoretically mature and supported by robust tools. We assume that the reader has a basic understanding of Petri nets [1]. But, as a general reminder, we note that Petri nets include three basic entities: place nodes (represented graphically by circles), transition nodes (represented graphically by solid bars), and directed arcs that can connect places to transitions or transitions to places. Furthermore, places can contain markers, called tokens, and tokens may move between place nodes by the “firing” of the associated transitions. The state of a Petri net refers to the distribution of tokens to place nodes at any particular point in time (this is sometimes called the marking of the net). We now proceed to discuss the basics of the G-net models.

A G-net system is composed of a number of G-nets, each of them representing a self-contained module or object. A G-net is composed of two parts: a special place called *Generic Switch Place (GSP)* and an *Internal Structure (IS)*. The *GSP* provides the abstraction of the module, and serves as the only interface between the G-net and other modules. The *IS*, a modified Petri net, represents the design of the module. An example of G-nets is shown in Figure 1. Here the G-net models represent two objects – a *Buyer* and a *Seller*. The generic switch places are represented by *GSP(Buyer)* and *GSP(Seller)* enclosed by ellipses, and the internal structures of these models are represented by round-cornered rectangles that contain four methods: *buyGoods()*, *askPrice()*, *returnPrice()* and *sellGoods()*. The functionality of these methods are defined as follows: *buyGoods()* invokes the method *sellGoods()* defined in G-net *Seller* to buy some goods; *askPrice()* invokes the method *returnPrice()* defined in G-net *Seller* to get the price of some goods*; returnPrice()* is defined in G-net *Seller* to calculate the latest price for some goods and *sellGoods()* is defined in G-net *Seller* to wait for the payment, ship the goods and generate the invoice. A *GSP* of a G-net *G* contains a set of methods *G.MS* specifying the services or interfaces provided by the module, and a set of attributes, *G.AS*, which are state variables. In *G.IS*, the internal structure of G-net *G*, Petri net places represent primitives, while transitions, together with arcs, represent connections or relations among those primitives. The primitives may define local actions or method calls. Method calls are represented by special places called *Instantiated Switch Places (ISP)*. A primitive becomes *enabled* if it receives a token, and an enabled primitive can be executed. Given a G-net *G*, an *ISP* of *G* is a 2-tuple *(G'.Nid, mtd)*, where *G'* could be the same G-net *G* or some other G-net, *Nid* is a unique identifier of G-net *G'*, and $mtd \in G'.MS$. Each *ISP(G'.Nid, mtd)* denotes a method call *mtd()* to G-net *G'*. An example *ISP* (denoted as an ellipsis in Figure 1) is shown in the method *askPrice()* defined in G-net *Buyer*, where the method *askPrice()* makes a method call *returnPrice()* to the G-net *Seller* to query about the price for some goods. Note that we have highlighted this call in Figure 1 by the dashed-arc, but such an arc is not actually a part of the static structure of G-net models. In addition, we have omitted all function parameters for simplicity.

From the above description, we can see that a G-net model essentially represents a module or an object rather than an abstraction of a set of similar objects. In a recent paper [25], we defined an approach to extend the G-net model to support class modeling. The idea of this extension is to generate a unique object identifier, *G.Oid*, and initialize the state variables when a G-net object is instantiated from a G-net

G. An *ISP* method invocation is no longer represented as the 2-tuple *(G'.Nid, mtd)*, instead it is the 2-tuple *(G'.Oid, mtd)*, where different object identifiers could be associated with the same G-net class model.

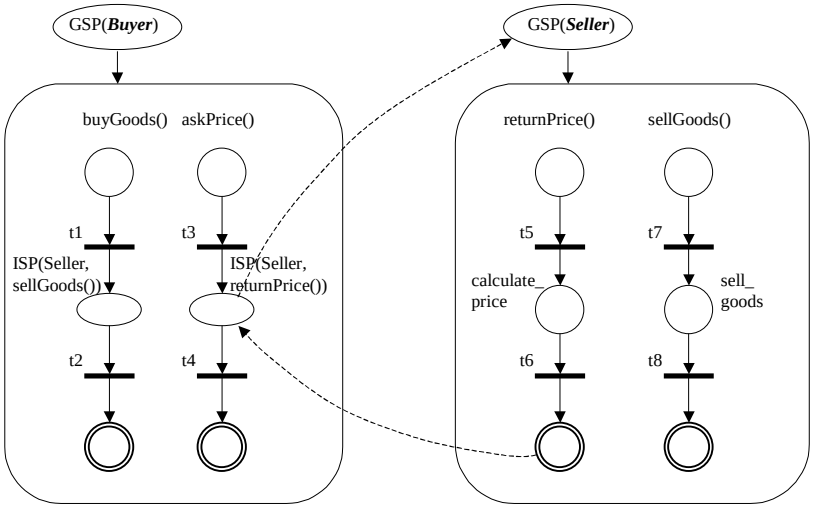


Figure 1. G-net models of buyer and seller objects

The token movement in a G-net object is similar to that of original G-nets [2][3]. A token *tkn* is a triple *(seq, sc, mtd)*, where *seq* is the propagation sequence of the token, *sc* ∈ {**before**, **after**} is the status color of the token and *mtd* is a triple *(mtd_name, para_list, result)*. For ordinary places, tokens are removed from input places and deposited into output places by firing transitions. However, for the special *ISP* places, the output transitions do not fire in the usual way. Recall that marking an *ISP* place corresponds to making a method call. So, whenever a method call is made to a G-net object, the token deposited in the *ISP* has the status of **before**. This prevents the enabling of associated output transitions. Instead the token is "processed" (by attaching information for the method call), and then removed from the *ISP*. Then an identical token is deposited into the *GSP* of the called G-net object. So, for example, in Figure 1, when the *Buyer* object calls the *returnPrice()* method of the *Seller* object, the token in place *ISP(Seller, returnPrice())* is removed and a token is deposited into the *GSP* place *GSP(Seller)*. Through the *GSP* of the called G-net object, the token is then dispatched into an *entry* place of the appropriate called method,

for the token contains the information to identify the called method. During "execution" of the method, the token will reach a *return* place (denoted by double circles) with the result attached to the token. As soon as this happens, the token will return to the *ISP* of the caller, and have the status changed from **before** to **after**. The information related to this completed method call is then detached. At this time, output transitions (e.g., *t4* in Figure 1) can become enabled and fire.

Notice that the example we provide in Figure 1 follows the *Client-Server* paradigm, in which a *Seller* object works as a server and a *Buyer* object is a client. Further details about G-net models can be found in references [2][3][25].

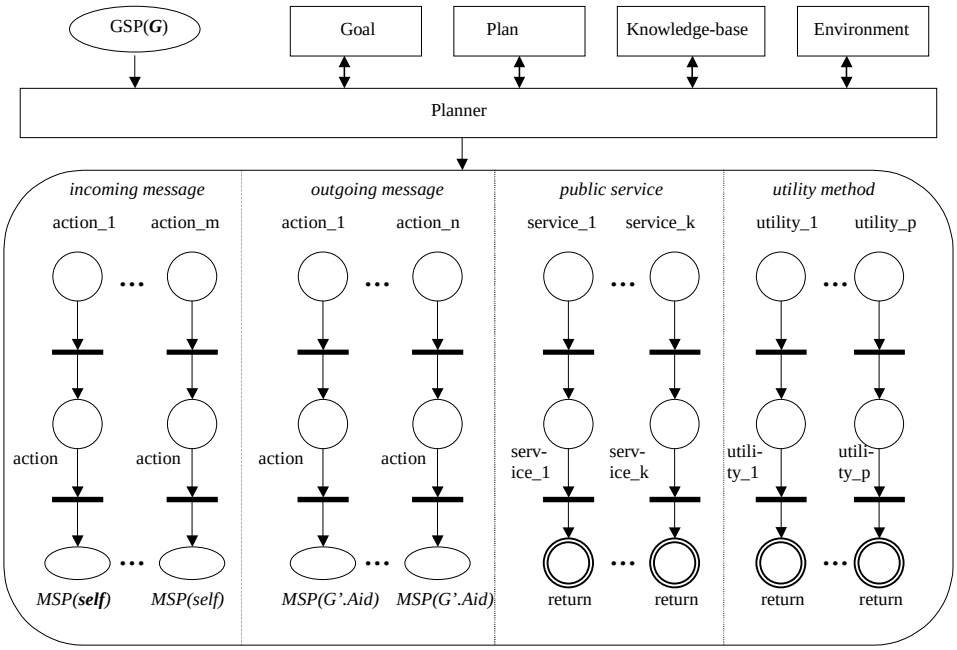
2.2 Agent-Oriented G-Net Model

Although the G-net model works well in object-based design, it is not sufficient in agent-oriented design for the following reasons. First, agents that form a multi-agent system may be developed by different vendors independently, and those agents may be widely distributed across large-scale networks such as the Internet. To make it possible for those agents to communicate with each other, it is desirable for them to have a common communication language and to follow common protocols. However the G-net model does not directly support protocol-based language communication between agents. Second, the underlying agent communication model is usually asynchronous, and an agent may decide whether to perform actions requested by some other agents. The G-net model does not directly support asynchronous message passing and decision-making, but only supports synchronous method invocations in the form of *ISP* places. Third, agents are commonly designed to determine their behavior based on individual goals, their knowledge and the environment. They may autonomously and spontaneously initiate internal or external behavior at any time. The G-net models can only directly support a predefined flow of control.

To support agent-oriented design, we need to extend a G-net to support modeling an agent class[2] [4][5]. This extension is made in three steps. First, we introduce five special modules to a G-net to make an agent autonomous and internally motivated. As shown in Figure 2, the five special modules are the *Goal*

[2] We view the abstract of a set of similar agents as an agent class, and we call an instance of an agent class an agent or an agent object.

module, the *Plan* module, the *Knowledge-base* module, the *Environment* module and the *Planner* module. The *Goal*, *Plan* and *Knowledge-base* module are based on the BDI agent model proposed by Kinny and his colleagues [8], while the *Environment* module is an abstract model of the environment, i.e., the model of the outside world of an agent. The *Planner* module represents the heart of an agent that may decide to ignore an incoming message, to start a new conversation, or to continue with the current conversation. In the *Planner* module, committed goals are achieved, and the *Goal*, *Plan* and *Knowledge-base* modules of an agent are updated after the processing of each communicative act that defines the type and the content of a message [29][30], or if the environment changes. Second, different from the semantic of a G-net as an object or a module, we view the extended G-net, we call it an *agent-oriented G-net*, as a class model, i.e., the abstract of a set of similar agent objects. Third, we define the instantiation of the agent-oriented G-net as follows: when an agent-oriented G-net *A* is instantiated, we generate an agent identifier *A.Aid* for the resulting agent object *AO*; meanwhile, the state of *AO*, i.e., any state variables defined in *A*, is initialized.



Notes: G'.Aid = messageToken.body.msg.receiver

Figure 2. A generic agent-oriented G-net model

The *Internal Structure (IS)* of an agent-oriented G-net consists of four sections: *incoming message*, *outgoing message*, *public service*, and *utility method*. The *incoming/outgoing message* section defines a set of *Message Processing Units (MPU)*, which corresponds to a subset of communicative acts. Each *MPU*, labeled as *action_i* in Figure 2, is used to process incoming/outgoing messages and execute any necessary actions before or after the message being processed. The *public service* section defines a set of methods that provide services to other agents, and it makes an agent work as a server. Note that this section is optional, and it is not necessary that an agent must provide public services to the outside world. Similarly, the *utility method* section defines a set of methods that can only be called by the agent itself.

Although both objects (passive objects) and agents use message-passing to communicate with each other, message-passing for objects is a unique form of method invocation, while agents distinguish different types of messages and model these messages frequently as speech-acts and use complex protocols to negotiate [13][14]. In particular, these messages must satisfy the format of the standardized communicative (speech) acts, e.g., the format of communicative acts defined in the FIPA agent communication language, or KQML [15][29][30]. Note that in Figure 2, each named *MPU action_i* refers to a communicative act, and the agent-oriented G-net model supports an agent communication interface through the *GSP* place. In addition, agents analyze these messages and can decide whether to execute the requested action. As stated before, agent communications are typically based on asynchronous message passing. Since asynchronous message passing is more fundamental than synchronous message passing, it is useful for us to introduce a new mechanism, called *Message-passing Switch Place (MSP)*, to directly support asynchronous message passing [4]. When a token reaches an *MSP* (represented as an ellipsis in Figure 2), the token is removed and deposited into the *GSP* of the called agent. But, unlike with the G-net *ISP* mechanism, the calling agent need not wait for the token to return before it can continue to execute its next step.

The *Planner* module has the functionality of message dispatching and decision-making. In addition, the *Planner* module also includes a sensor, which may capture internal or external events, and invoke certain plans correspondingly. To support agent-oriented design, the *Planner* module has been designed in such a

way that it supports inheritance for *MPU*s and methods defined in the internal structure of an agent-oriented G-net model. Due to the size limitation of this paper, we do not discuss further for the *Planner* module. A detailed description for this module can be found in earlier work [4][5]. It is worth noting that G-nets can be used to model the *Knowledge-base* module and the decision-making units in the *Planner* module due to their support for knowledge representation and reasoning [26].

2.3 From Formal Agent Model to Agent Implementation

ADK (Agent Development Kit) is intended to provide the necessary facilities for agent implementation based on the formal agent model described previously. Thus, the development of ADK is not ad hoc, but results from a model-based development process. The agent-oriented G-net model, as an operational model, provides the specification and high-level design for intelligent communicating agents. Specifically, the key components or mechanisms defined in the agent-oriented G-net model serve as building blocks of our agent development kit, and the agent-oriented G-net model itself becomes a high-level design model for intelligent communicating agents.

As Figure 3 shows, the role of ADK is to serve as a bridge between the formal agent model and the agent implementation platform. The key components and mechanisms defined for an intelligent communicating agent, as shown on the left hand side of the figure, are listed as follows. First, the modularization of the agent design provides the formal agent architecture that makes an agent autonomous, reactive, proactive and sociable. For instance, the *Goal*, *Plan*, and *Knowledge-base* modules are based on the BDI agent model [8] that is a conceptual model for intelligent agents. The *Planner* module is used for decision-making, message dispatching and event capturing. And the *Internal Structure* is a container for methods and *MPU*s, where methods are defined for method invocation, and *MPU*s support asynchronous message passing. Second, the message passing mechanisms are defined in two cases: the synchronous message passing and asynchronous message passing. Synchronous message passing is usually used for method invocation, and it is realized through the *ISP* mechanism, while asynchronous message passing is vital for agent communication, and it is achieved by the *MSP* mechanism [4]. Recall that in the case of asynchronous message passing, when a *MSP* is called, the agent does not need to wait for the result to

come back, and it may proceed to execute other functionality. Third, the formal agent model defines the functional units as inheritable components. As methods are defined as inherited units in object-oriented design, both methods and *MPU*s could be inherited from an agent superclass to an agent subclass.

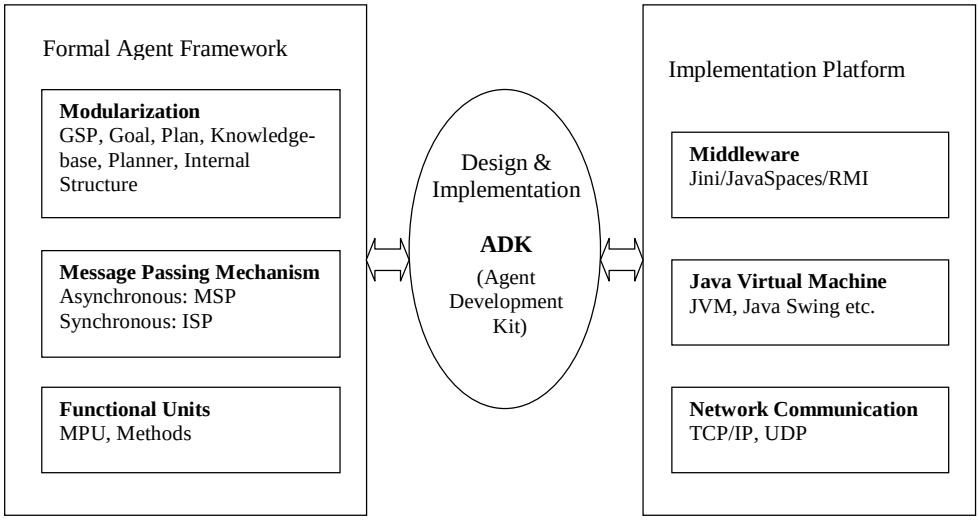


Figure 3. The role of ADK between formal agent model and implementation platform

As shown on the right hand side of Figure 3, the implementation platform provides the standard technologies, such as the Jini middleware [16][17] and the Java Virtual Machine (JVM), for agent implementation. We choose Java as our programming language because applications developed on JVM are platform independent, and they are suitable for web-based applications such as electronic commerce. In addition, we use the Jini middleware to simplify our development process for agent communication. In this case, we do not need to take care of the low-level communication protocols, such as the TCP/IP and UDP protocols, which can be automatically handled by the Jini middleware, and can concentrate on high-level communication protocols, such as price-negotiation protocol. Finally, In the middle of this figure, ADK represents the design and implementation, and it provides the framework and the class library for developing intelligent communicating agents in multi-agent systems. A multi-agent system built upon ADK can be realized by deriving the required specific agent classes from a template, which is the *Agent*

class defined in ADK. We will see that it is straightforward to augment the agent framework with application-specific functionality to meet system requirements.

3. Design of Intelligent Communicating Agents

3.1 Middleware Support for Agent Communication

As we mentioned before, the Jini middleware can be used to simplify the development process for agent communication. The Jini architecture is intended to resolve the problem of network administration by providing an interface where different components of the network can join or leave the network at any time [16][17]. Such a collection of services is called a Jini community, and the services within the Jini community represent service providers or service consumers. The heart of the Jini system is a trio of protocols called *discovery*, *join*, and *lookup*. *Discovery* occurs when a service is looking for a lookup service with which to register. *Join* occurs when a service has located a lookup service and wishes to join it. And *lookup* occurs when a client or user needs to locate and invoke a service described by its interface type and possibly, other attributes.

In designing the ADK, we use Jini as a middleware for agents to find each other. Each agent is designed as both a service provider and a service consumer. However the only service that an agent may provide is to let other agents send asynchronous messages to that agent.

In the agent-oriented G-net model, the *GSP (Generic Switch Place)* is defined as the only interface among agents [4][5]. Thus, we design the schema for an agent interface as follows:

```
public interface GSP extends Remote {
    public void asynMessagePassing(Message message) throws RemoteException;
}
public class MiddlewareSupport implements GSP {
    // agent interface
```

```
    public void asynMessagePassing(Message message) {
        System.err.println("This method should be overridden by an agent class!");
    }
    // find lookup services and join the Jini community
    public void setup(String[] groupsToJoin) {…}
    …
}
```

The class *MiddlewareSupport* implements the *GSP* interface, where an abstract method *asynMessagePassing()* is defined. However, in class *MiddlewareSupport*, the implementation of this method is again deferred to subclasses of the *MiddlewareSupport* class because we want that the class *MiddlewareSupport* only defines the functionality to deal with the Jini community, such as discovering lookup service on the network, registering with the Jini community and searching for other agents in the Jini community. Here the method *setup()* is defined to let the *GSP* find a lookup service and joins the Jini community. As we will see in Section 3.2, the *Agent* class, which is defined as a subclass of the *MiddlewareSupport* class, actually implements the method *asynMessagePassing()*, and inherits all the functionality defined in class *MiddlewareSupport*.

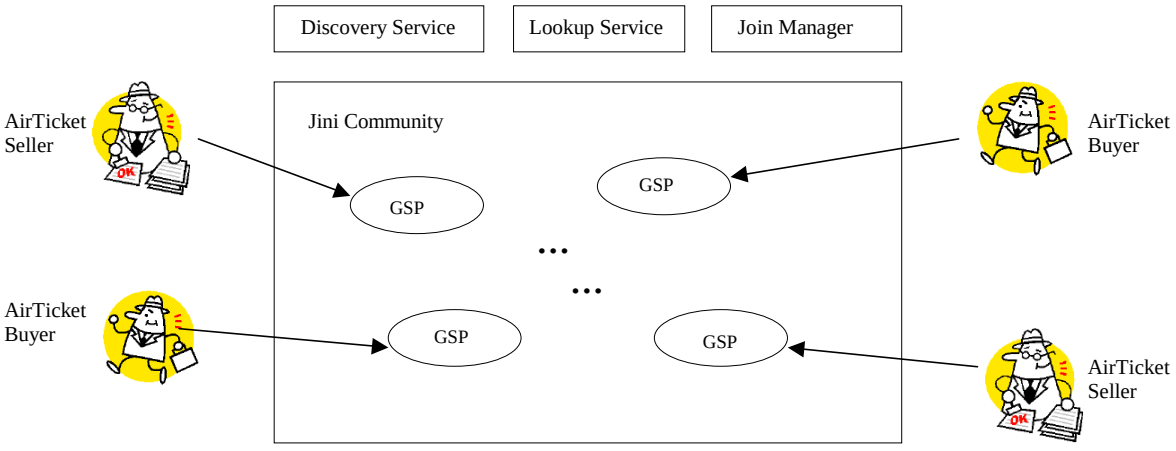


Figure 4. The Jini community with agents of *AirTicketSeller* and *AirTicketBuyer*

As an example, consider the design of an electronic marketplace in which seller agents and buyer agents may find each other and communicate with each other asynchronously through the Jini community. The design is illustrated in Figure 4, where both air ticket seller agents and air ticket buyer agents register their *GSP* interfaces with the Jini community, and they may find each other by the agent attribute, for instance, an agent name called "Seller".

3.2 A Pattern for Intelligent Communicating Agents

An intelligent communicating agent has both the properties of intelligent agents and communicating agents. An intelligent agent is defined as an agent that at least has the following characteristics: autonomy, reactivity, and proactiveness, while the definition of a communicating agent emphasizes on its sociability. Agent autonomy is akin to human free will and enables an agent to choose its own actions, while agent proactiveness requires an agent to behave in a goal-directed fashion. Agent proactiveness is usually considered in relation to planning, and is strengthened with agent autonomy. We call an autonomous and proactive agent a *goal-driven* agent. A reactive agent is defined as an agent that has the ability to perceive and to response to a changing environment. In the Jini community, whenever a new event occurs, an agent should be automatically notified by the system. For instance, when a seller agent joins or leaves the Jini community, the buyer agents need to be notified; thus, the buyer agents can always keep an up-to-date list of the seller agents that are currently in the community. We call a reactive agent an *event-driven* agent, and an event could be any environment change that may influence an agent's execution. The sociability of an agent refers to the ability of an agent to converse with other agents. The conversations, normally conducted by sending and receiving messages, provide opportunities for agents to coordinate their activities and cooperate with each other, if needed. An agent is different from an object in that agents usually do not use method invocations to communicate with each other. On the contrary, agents distinguish different types of messages and use complex protocols to negotiate. In addition, agents analyze these messages and can decide whether to execute the requested action [13]. To meet this requirement, the design of agents needs to support asynchronous message passing. We call an agent that supports asynchronous message passing a *message-triggered* agent.

Figure 5 shows the architectural design for intelligent communicating agents. Compared with the agent-oriented G-net model in Figure 2, an obvious variation in Figure 5 is that the *GSP* place of an agent now becomes a part of the environment module, which is the Jini community. This variation shows a simple design of the environment module in ADK, in which case, the only external events of concern are those related to agent entering and/or leaving the Jini community. In future design versions, it is possible to extend the environment module to include other events, such as network topology changes and user interventions.

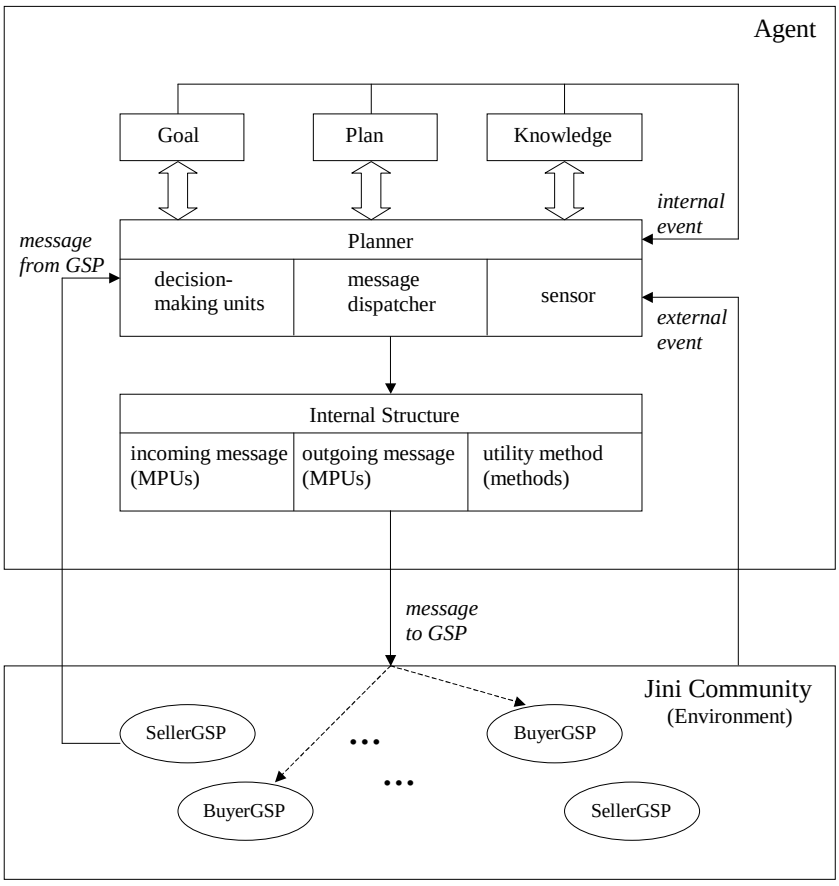


Figure 5. The architectural design of intelligent communicating agents

Similarly, data changes in *Goal*, *Plan* and *Knowledge-base* modules may act as internal events and trigger the sensor in the *Planner* module. To simplify matters, in our current version of ADK, the sensor in the *Planner* module is implemented to only capture external events.

Figure 5 also shows that, when an agent *A* wants to converse with another agent *B*, it sends a message to the *GSP* of agent *B* in the Jini community. Then the message will be sent to the *Planner* module of agent *B*. After the message is dispatched into a *MPU* in the incoming message section, the message will be processed, e.g., decoded, and sent back to the *Planner* module. Now the message goes to the decision-making units, where decisions may be made to ignore the message, or to continue with the conversation. If the conversation is to be continued, a new outgoing message is generated, and dispatched into a *MPU* defined in the outgoing message section. The outgoing message will be processed and certain actions may be executed before the message is sent to the *GSP* of agent *A*.

In addition, an intelligent communicating agent will not work as a server. Therefore, we have not included the public service section in the *Planner* module. The *MPU*s and the methods, which are defined in the incoming/outgoing message section and utility method section respectively, can be inherited by agent subclasses, and can only be accessed or called by the agent itself.

The goal of the above architectural design is to derive an architectural rendering of a system, which serves as a framework from which more detailed design activities are conducted. Based on the architectural design illustrated in Figure 5, we now proceed to describe the detailed design of intelligent communicating agents. This design is expressed in the form of a pattern or class template.

Since the agent-oriented G-net model supports inheritance, we will follow this design schema. In an object-oriented system, design patterns can be used with either inheritance or composition. Using inheritance, an existing design pattern becomes a template for a new subclass, and the attributes and operations that exist in the pattern become part of the subclass [18]. Similarly, in an agent-oriented system, a pattern of an agent superclass can serve as a template for an agent subclass, and a specific agent subclass, such as an air ticket seller agent class, can be derived from an agent superclass by augmenting the template to meet system requirements.

The *Agent* class defined in ADK provides such a pattern for agent implementation. The pattern in a form of Java pseudocode is shown in Figure 6.

```
public class Agent extends MiddlewareSupport {
    private static final String PRODUCT = "Agent";
    private static final String MANUFACTURER = "CSSL@UIC";
    private static final String VERSION = "ADK 1.0";
    …
    /*************************
     * Agent Interface -- GSP *
     *************************/
    public void asynMessagePassing(Message message) {
       Thread messageProcessThread = new Thread(new Runnable() {
           public void run() {
               dispatchMessage(message);           // -- message-triggered
           }
       });
       messageProcessThread.start();
    }

    /*********************************************
     * Class Variables for Knowledge, Goal and Plan *
    /*********************************************/
    Goal[]: myGoals; // a list of committed goals
    Plan[]: myPlans; // a set of plans
    Knowledge: myKnowledge; // a knowledge-base
    …
    /***********
     * Planner *
     ***********/
    private class Sensor extends Listener {
        …
        public void notify(RemoteEvent ev) {
            if (!(ev instanceof ServiceEvent)) return;
            updateServices();
            invokePlan(ev);                        // -- event-driven
        }
    }
    protected void dispatchMessage(Message message) {…}
    protected Message makeDecision(Message message) {…}
    protected void updateMentalState() {…)
    …
    /**********************
     * Internal Structure *
     **********************/
    // incoming message section – a set of message processing units
    protected void MPU_In_1(Message message) {…}
    …
    // outgoing message section – a set of message processing units
    protected void MPU_Out_1(Message outgoingMessage) {…}
    …
    // utility method section – a set of private utility methods
    protected void Method_1() {…}
    …

    public static void main(String[] args) {
        initAgent(args);
        autonomousRun();                           // -- goal-driven
    }
}
```

Figure 6. A pattern for intelligent communicating agents

As shown in Figure 6, the *Agent* class is defined as a subclass of *MiddlewareSupport* (as defined in Section 3.1) to reuse the functionality of discovering a lookup service, registering with the Jini community, and searching for other agents. More importantly, an agent object may communicate with other agent objects asynchronously through the *GSP* interface. This functionality makes an agent sociable. To simulate the asynchronous message passing, we have used the thread technique to generate a new thread called *messageProcessThread*. Upon receiving an incoming message, the *messageProcessThread* of the message receiver (the callee) dispatches the message to a *MPU* and returns immediately. This ends up the *messageProcessThread* quickly, and therefore, the message sender (the caller) does not need to wait for the message to be processed and may proceed to execute other tasks.

Corresponding to the three modules (*Goal*, *Plan* and *Knowledge*) in the architectural design of intelligent communicating agents (Figure 5), the *Agent* class defines a list of committed goals *myGoals*, a set of plans *myPlans*, each of which is associated with a goal or a subgoal, and a knowledge-base *myKnowledge*. The *Goal*, *Plan* and *Knowledge* class define the basic properties and behaviors for an intelligent agent, and may be refined if an application-specific agent requires further functionality. Refer to Figure 7 for the definitions of the *Goal*, *Plan* and *Knowledge* class. For brevity, other class variables, such as *theGoalSet* – a set of goals from which the goal list *myGoals* is generated – are omitted in Figure 6.

The reactivity of an agent can be designed through the Jini's notification facility. In Figure 6, we can see that the *Sensor* class is defined as a private inner class in the *Agent* class, and is derived as a subclass from the *Listener* class, which is defined by Jini. Thus, an application class, such as a seller agent class or a buyer agent class, can be defined as a subclass of the *Agent* class, and can be notified by the Jini community whenever an event occurs, as long as the corresponding agent object has instantiated a *Sensor* object and has registered it with the Jini community.

Based on the architectural design of intelligent communicating agents in Figure 5, the *Planner* module in the *Agent* pattern defines a method called *dispatchMessage()*, which is used to dispatch messages to the appropriate *MPU* defined in the incoming/outgoing message section. Examples of methods defined as

decision-making units in the *Planner* module are the methods *makeDecision()* and *updateMentalState()*. In method *makeDecision()*, decisions are made to ignore an incoming message, to start a new conversation, or to continue with the current conversation. In method *updateMentalState()*, the mental state of the agent, i.e., the goal, plan, and knowledge-base are updated whenever a decision is made or a new event occurs. The *Internal Structure* module includes three sections, i.e., the incoming message section, outgoing message section, and utility method section. Each section defines a set of *MPU*s or methods, which are depicted as *MPU_In_i()*, *MPU_Out_j()* or *Method_k()* in Figure 6. The autonomy and proactiveness of an agent are related with the *Goal*, *Plan*, *Knowledge-base*, *Planner* and *Internal Structure* modules of an agent. To connect them together, we define the control as the method *autonomousRun()*, which includes a list of committed goals to be achieved based on the agent's mental state. Each goal is defined as a goal tree that is traversed in depth-first order, and selected plans associated with each goal or subgoal are invoked accordingly. The method *autonomousRun()* is invoked in the method *main()*, as shown in Figure 6, and is executed after the agent is initialized with the method *initAgent()*.

One advantage of our model-based approach is its support for the principle of "separation of concerns," in particular the separation of agent intelligence and agent communication mechanisms. Therefore, it is possible for us to choose some existing implementation schema of intelligent agents to design and implement intelligent communicating agents for multi-agent systems. For instance, we can choose the *Task Representation Language (TRL)* to support knowledge representation and agent reasoning [19], or we can use Petri nets to model the mental state of agents for multi-agent simulation [22]. Alternatively, we can, and do, use a more commonly used intelligent agent model – the *Belief-Desire-Intention (BDI)* model [8]. A *BDI* architecture includes and uses an explicit representation for an agent's beliefs, desires and intentions. The BDI implementations, such as The Procedural Reasoning System (PRS), the University of Michigan PRS, and JAM, all define a new programming language and implement an interpreter for it [20]. The advantage of this approach is that the interpreter can stop the program at any time, save state, and execute some other plan, or intention, if it needs to. In this paper, we use a simplified implementation of the BDI agent model based on previous work, and show how to integrate it into ADK in developing intelligent communicating agents

The relationships between the key classes defined for communicating agents and intelligent agents are illustrated in Figure 7. As shown in this figure, two key classes for a communicating agent are the *Agent* class and the *Message* class, and an *Agent* object may send or receive *Message* objects through its *GSP* interface. Meanwhile, the three key classes for an intelligent BDI agent are the *Goal*, *Plan* and *Knowledge* class. A *Goal* object is defined as a goal tree, and a goal or a subgoal associates with a set of plans. When a goal or a subgoal is to be achieved, the most appropriate plan, for instance, the plan with the highest priority, is selected and executed. As a result of the execution of a plan, a *Knowledge* object may be updated. Both a *Goal* object or a *Plan* object may use the *Knowledge* object for its own purpose, e.g., to select the right plan to achieve a goal or a subgoal.

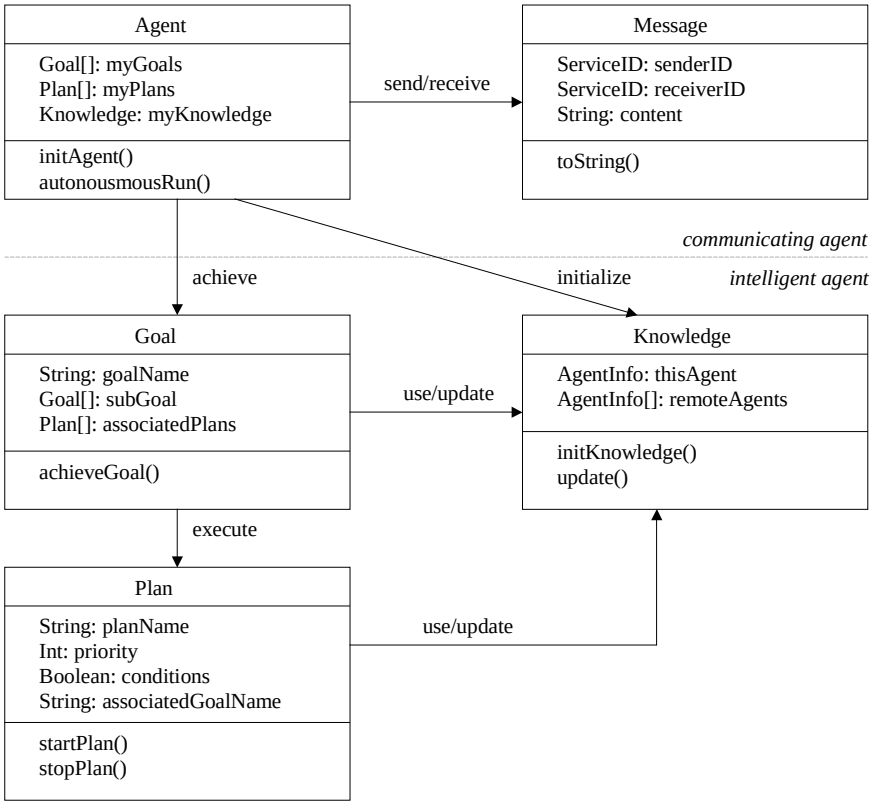


Figure 7. Relationship between classes defined for communicating agents and intelligent agents

The *Agent* class defines a list of committed goals *myGoals*, a set of plans *myPlans* that associate with a goal or a subgoal, and a knowledge-base *myKnowledge*. The list of committed goals and the set of plans

may be updated at run time. For instance, when a goal is achieved, it may be deleted from the goal list, and new goals may be added into the goal list if needed. In addition, the *myKnowledge* object is initialized by the *Agent* object, and may be updated at run time by a *Goal* or *Plan* object. The intelligent communicating agent is so-called *goal-driven*, because in the method *automousRun(),* goals defined in the goal list are achieved one by one through a loop. When all the goals are achieved, the *Agent* object waits for new committed goals to be added into the goal list.

4. Implementation of Multi-Agent Systems

4.1 An Agent Development Process

The purpose of the proposed agent design architecture is to ease the programmer's effort to develop applications of intelligent communicating agents for multi-agent systems. As we mentioned before, a specific agent, such as an air ticket seller agent, could be defined as an agent subclass of the *Agent* class. To illustrate this idea, we present a class hierarchy for an electronic marketplace in Figure 8. In this figure, all the classes above the dashed line are provided as an agent framework or a class library – these classes define the ADK environment, which supports developing intelligent communicating agents for multi-agent systems. The classes below the dashed line are derived classes that represent specific intelligent communicating agents in a multi-agent system. Since the *Agent* class shown in Figure 6 provides the basic functionality of intelligent communicating agents as well as the agent implementation framework, what we need to do for developing a specific intelligent communicating agent is to inherit the functional units and the behaviors of the *Agent* superclass and fill out certain sections in the pattern for intelligent communicating agents, such as the incoming/outgoing message section (Figure 6). In addition, we need to define subclasses of the *Goal*, *Plan*, and *Knowledge* classes defined in ADK to meet certain behavioral requirements of agent intelligence.

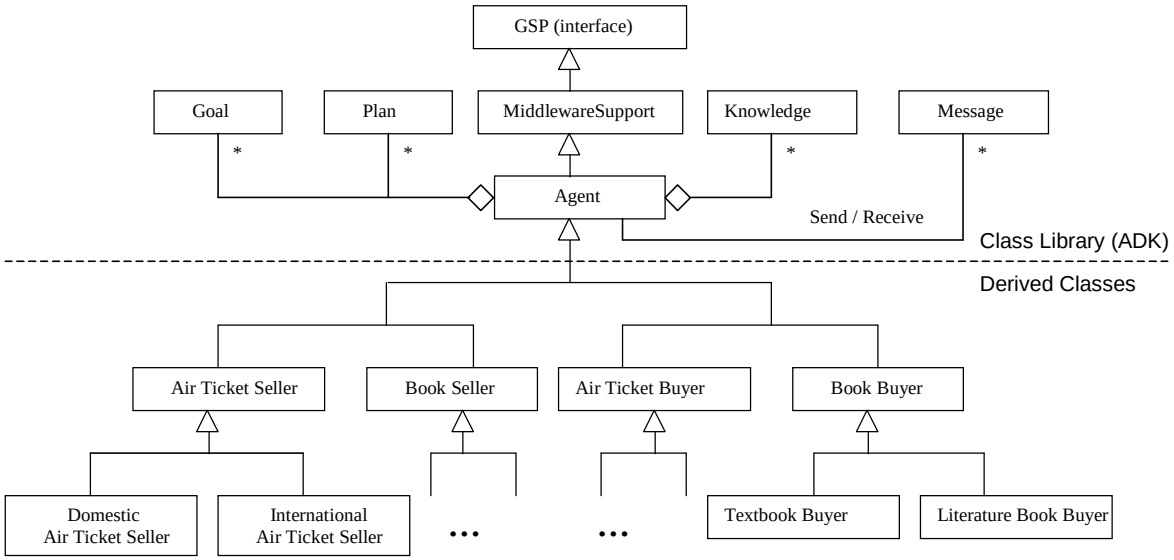


Figure 8. The class hierarchy diagram of agents in an electronic marketplace

As a summary, we now briefly describe the generic procedure to develop a specific intelligent communicating agent for multi-agent systems. In Section 4.2, we cast the procedure into more specific terms by way of an example. The 6-step procedure is defined as follows:

1. Define a set of goals Φ as the class variable *theGoalSet*, where each goal is defined as a goal tree Γ. A goal tree could consist of just a root, which means a goal may or may not have a number of subgoals.
2. Define a goal list Ω as the class variable *myGoals* (Figure 6) and initialize the goal list Ω with any committed goal $g_c \in \Phi$. The goal list Ω is dynamic, which means achieved goals may be deleted from Ω and newly committed goals could be added into Ω at run time.
3. Define a set of plans P as the class variable *myPlans* (Figure 6). Each plan $p \in P$ has a priority and a set of conditions, and is associated with a particular goal or subgoal. The plan $p_{hp} \in P$, which has the highest priority and whose conditions are evaluated to *true*, will be executed to achieve the associated goal or a subgoal.
4. Each plan p corresponds to a contract net protocol ρ [21], which serves as a template for agent conversation. From the contract net protocol, we define a set of *MPU*s Ψ, where each MPU

corresponds to a method *MPU_In_i()* or *MPU_Out_j()* as shown in Figure 6. Refer to [4][5] for a detailed description for transforming from ρ to ψ.

5. Refine the *Knowledge* class if the application-specific agent requires additional types of knowledge beyond the basic properties and behaviors predefined in Figure 7, and initialize the knowledge-base *myKnowledge* (Figure 6) for that agent.
6. Refine the decision-making units defined in the *Agent* class, if needed. Examples of decision-making units include functions like *makeDecision()*, *updateMentalState()* and *invokePlan()*.

The decision-making units serve as the reasoning engine for the agent. The major functionality of the decision-making units includes the following tasks:

- For each goal or subgoal, choose the most appropriate plan to execute.
- Create outgoing messages and send them out through *MPU*s.
- Upon receiving incoming messages, decide to ignore or continue with the conversations.
- Decide when to update the agent's mental state.
- Upon capturing new events, update the goal list and invoke certain plans.

It should be mentioned that the above procedures may be automated, or partially automated by providing a development environment, to ease the programmers' work. This is also one of the major motivations of our ADK project. An *Agent Development Environment (ADE)*, which encompasses the ADK, is envisioned as a future, and more ambitious research direction.

4.2 A Multi-Agent System Example: Air-Ticket Trading

As an example for intelligent communicating agents, suppose we wish to design and implement a multi-agent system for air ticket trading. The multi-agent-agent system will include two types of agents, air ticket seller agents and air ticket buyer agents. According to the procedures described above, a set of goals will be identified for both the air ticket sellers and the air ticket buyers. For instance, the goal list for a simplified air ticket buyer may include the goal "*buy air ticket*," and the goal "*buy air ticket*" may have

subgoals of "*find seller*," "*check price*," "*buy ticket*," and "*wait for receipt*," as shown on the right hand side of Figure 9. The air ticket seller has a similar goal list for the purpose of selling air tickets. For each goal or subgoal, we define a set of plans. For instance, for the subgoal "*find seller*", we have two plans, which are *plan_FindSeller* and *plan_BeFoundBySeller*. The plan *plan_FindSeller* can be executed to search for air ticket sellers in the Jini community, while the plan *plan_BeFoundBySeller* is executed to wait to be found by air ticket sellers. Which plan will be executed to achieve the subgoal "*find seller*" is determined by actual situations. For instance, the buyer may want to wait and be contacted by air ticket sellers initially. However, if the subgoal cannot be achieved in a period of time, the buyer can change its mind to search for air ticket sellers by itself.

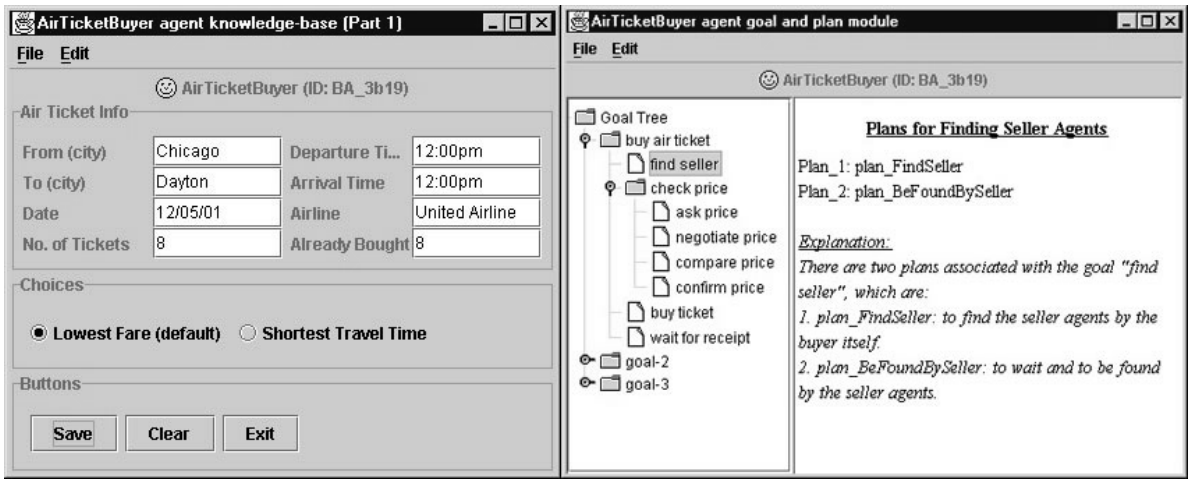


Figure 9. User Interface of the *Knowledge-base*, *Goal* and *Plan* module

The contract net protocols correspond to the above two plans are fairly simple. For the plan *plan_FindSeller*, the buyer asks the sellers in the Jini community if they sell air tickets, then the sellers may reply with "Yes" or "No", or simply ignore the conversation. If a seller replies with "Yes," the buyer may ask further questions to check if the air ticket seller has enough certain types of air tickets. For instance, the buyer may ask if the seller has tickets from "Dayton" to "Chicago." If the seller has the type of air tickets that the buyer wants, the subgoal may be achieved or partially achieved (if the seller has the

type of tickets but not enough). Then, in the next step, the seller continues to achieve the subgoal "*check price*."

This gives some examples of how to "fill out" certain sections of the implementation pattern provided by the *Agent* class. Now we list a few *MPU*s that correspond to the above two plans:

```
// incoming message section
// plan_FindSeller
protected void MPU_In_SellerYesOrNo(Message message) {}
…
// plan_BeFoundBySeller
protected void MPU_In_BeFoundBySeller(Message message) {}

// outgoing message section
// plan_FindSeller
protected void MPU_Out_FindSeller(Message outgoingMessage) {}
…
// plan_BeFoundBySeller
protected void MPU_Out_BuyerYesOrNo(Message outgoingMessage) {}
…
```

The *Knowledge-base* of a seller or buyer agent includes two parts, which provides information about the agent itself and information about other agents. For instance, the *Knowledge-base* of the buyer agent should include ticket information for the type of tickets that the buyer agent wants to buy (as shown on the left hand side of Figure 9), and ticket information for the type of tickets that other seller agents may hold. Other information, such as the state of the agent itself and other agents, may also be stored in the *Knowledge-base* of that agent. We do not show these types of knowledge in our illustrated figures. Finally, for the decision-making units for this air ticket trading application, we simply reuse those that are predefined in ADK.

The user interface of a seller agent is designed as a console window as shown in Figure 10. In the agent console window, the content for the agent communication is displayed. Meanwhile, a list of agents, including the agent itself and those agents with which that agent communicates, is displayed on the left hand side of the window. The user interface will also provide a set of tools, such as to lookup existing services, to test message sending/receiving, and to edit agent properties. Figure 10 shows an example of air ticket trading process. In Figure 10, a buyer agent, with an agent ID of *BA_3b19*, first asks if the seller agent *SA_16fb* sells air tickets. After the seller agent *SA_16fb* confirms with "Yes", the buyer agent *BA_3b19* continues to ask if the seller agent *SA_16fb* has the type of air tickets it wants. After the seller agent *SA_16fb* confirms with "Yes" again (although it does not have enough tickets), the buyer agent *BA_3b19* begins to bargain price with the seller. Finally, the conversation between agent *SA_16fb* and agent *BA_3b19* ends up with a confirmation message that the buyer agent *BA_3b19* buys all the 5 tickets from the seller agent *SA_16fb* with the price of $180.0 for each ticket.

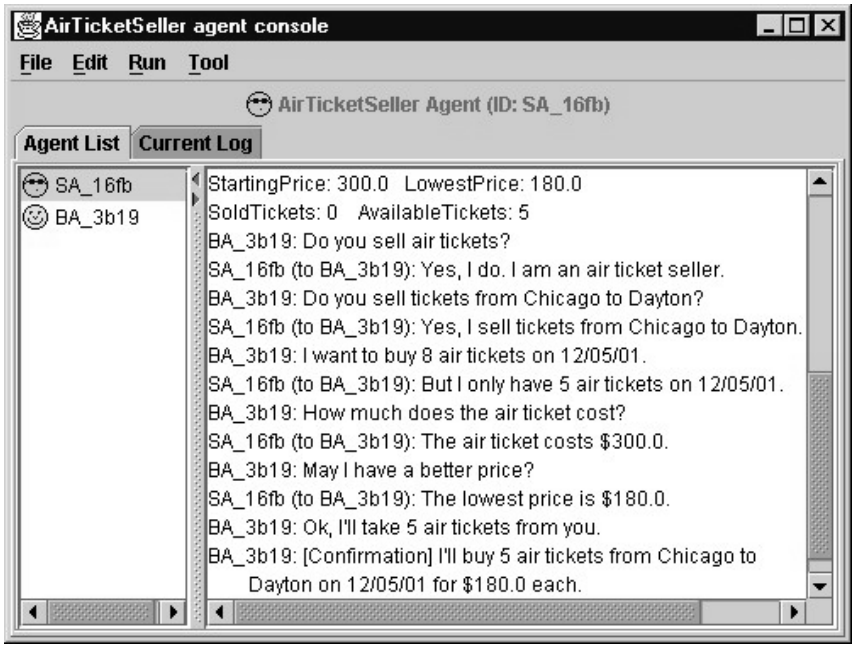


Figure 10. User interface of the seller agent *SA_16fb*

In this example, the agent ID for the seller agent or the buyer agent is defined by a prefix of *SA* (seller agent) or *BA* (buyer agent) with the last four digits of the service ID of that agent, where the service ID is a 32 digits hexadecimal number provided by the Jini community when the agent is registered [16][17].

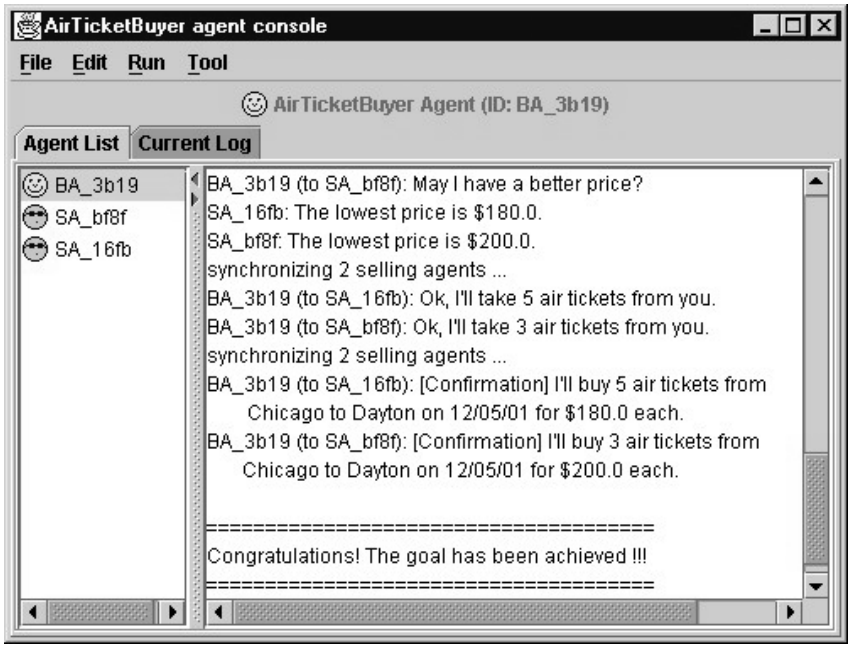


Figure 11. User interface of the buyer agent *BA_3b19*

In Figure 11, we show the user interface for the air ticket buyer agent. In this figure, we can see that the buyer agent *BA_3b19* concurrently communicates with two seller agents: *SA_bf8f* and *SA_16fb*, and buys 5 tickets from the seller *SA_16fb* and 3 tickets from the seller *SA_bf8f* with the *lowest fare* criteria.

5. Conclusions and Future Work

Although a number of agent-oriented systems have been built in the past few years, there is very little work on bridging the gap between theory, systems, and application. The contribution of this paper is to use the agent-oriented G-net model, which is a formal agent model, as a specification and a high-level design for agent development. Based on the architectural design and the detailed design of a generic

intelligent communicating agent, we developed the ADK as a class library that supports designing and implementing applications of intelligent communicating agents for multi-agent systems. An air ticket trading example was presented to illustrate the derivation of a multi-agent application using the ADK approach. The generality of the example supports the notion that our model-based approach is feasible and effective. For future work, we will formalize the design procedure for developing specific intelligent communicating agents, and based on the ADK class library, we will partially automate the implementation process to reduce the programming-level tasks. In future versions of this project, we plan to develop an *Agent Development Environment (ADE)* to support the development process.

References:

[1] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 44, April 1989, pp. 541-580.

[2] A. Perkusich and J. de Figueiredo, "G-Nets: A Petri Net Based Approach for Logical and Timing Analysis of Complex Software Systems," *Journal of Systems and Software*, vol. 39, no. 1, 1997, pp. 39-59.

[3] Y. Deng, S. K. Chang, A. Perkusich and J. de Figueredo, "Integrating Software Engineering Methods and Petri Nets for the Specification and Analysis of Complex Information Systems," *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, Chicago, June 21-25, 1993, pp. 206-223.

[4] H. Xu and S. M. Shatz, "An Agent-Based Petri Net Model with Application to Seller/Buyer Design in Electronic Commerce," *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems* (ISADS 2001), March 26-28, 2001, Dallas, Texas, USA, pp. 11-18.

[5] H. Xu and S. M. Shatz, "A Framework for Modeling Agent-Oriented Software," *Proceedings of the 21st International Conference on Distributed Computing Systems* (ICDCS-21), April 16-19, 2001, Phoenix, Arizona, USA, pp. 57-64.

[6] M. Fisher, "Representing and Executing Agent-Based Systems," *Intelligent Agents -- Proceedings of the International Workshop on Agent Theories, Architectures, and Languages*, M. Wooldridge,

and N. Jennings, eds., Lecture Notes in Computer Science, vol. 890, Springer-Verlag, 1995, pp. 307-323.

[7] M. Huber, "JAM: a BDI-Theoretic Mobile Agent Architecture," *Proceedings of International Conference on Autonomous Agents*, 1999, pp. 236-243.

[8] D. Kinny, M. Georgeff, and A. Rao, "A Methodology and Modeling Technique for Systems of BDI Agents," *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, W. Van de Velde and J. W. Perram, eds., LNAI vol. 1038, Springer-Verlag: Berlin, Germany, 1996, pp. 56-71.

[9] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas, "JACK Intelligent Agents – Summary of an Agent Infrastructure," *Proceedings of the 5th International Conference on Autonomous Agents*, 2001.

[10] R. Ashri and M. Luck, "Paradigma: Agent Implementation through Jini," *Proceedings of the Eleventh International Workshop on Database and Expert Systems Applications*, A. M. Tjoa and R. R. Wagner and A. Al-Zobaidie, eds., IEEE Computer Society, 2000, pp. 453-457.

[11] M. Luck and M. d'Inverno, "A Formal Framework for Agency and Autonomy," *Proceedings of the First International Conference on Multi-Agent Systems*, AAAI Press / MIT Press, 1995, pp. 254-260.

[12] J. Siegel, and the OMG Staff Strategy Group, "Developing in OMG's Model Driven Architecture (MDA)," *OMG White Paper*, Object Management Group, November 2001.

[13] M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 3, 2000, pp. 285-312.

[14] C. A. Iglesias, M. Garrijo, J. Centeno-González, "A Survey of Agent-Oriented Methodologies," *Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Language* (ATAL-98), 1998, pp. 317-330.

[15] J. Odell, H. Van Dyke Parunak, and B. Bauer, "Representing Agent Interaction Protocols in UML," *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge, eds., Springer-Verlag, Berlin, 2001, pp. 121–140.

[16] W. K. Edwards, *Core Jini*, The Sun Microsystems Press, Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[17] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath, *The Jini Specification*, Addison-Wesley, 1999.

[18] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th Edition, McGraw-Hill, 2001.

[19] T. R. Ioerger, R. A. Volz, and J. Yen, "Modeling Cooperative, Reactive Behaviors on the Battlefield Using Intelligent Agents," *Proceedings of the Ninth Conference on Computer Generated Forces* (9th CGF), 2000, pp. 13-23.

[20] J. M. Vidal, P. A. Buhler, and M. N. Huhns, "Inside an Agent," *IEEE Internet Computing*, vol. 5, no. 1, January-February 2001.

[21] R.A. Flores and R.C. Kremer, "Formal Conversations for the Contract Net Protocol," *Multi-Agent Systems and Applications II*, V. Marik, M. Luck & O. Stepankova, eds., Lecture Notes in Computer Science, Springer-Verlag, 2001.

[22] J. Yen, J. Yin, T.R. Ioerger, M. Miller, D. Xu, and R.A. Volz, "CAST: Collaborative Agents for Simulating Teamwork," *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (IJCAI-01), Seattle, WA, August 2001, pp. 1135-1142.

[23] W. Vasconcelos, J. Sabater, C. Sierra and J. Querol, "Skeleton-based Agent Development for Electronic Institutions," *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Italy, July 2002.

[24] J. A. Rodriguez-Aguilar, F. J. Martin, P. Garcia, P. Noriega and C. Sierra, "Towards a Formal Specification of Complex Social Structures in Multi-agent Systems," *Collaboration between Human and Artificial Societies*, J. Padget, ed., LNAI, vol. 1624, Springer-Verlag, 1999, pp. 284-300.

[25] H. Xu and S. M. Shatz, "Extending G-Nets to Support Inheritance Modeling in Concurrent Object-Oriented Design," *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics* (SMC 2000), October 2000, Nashville, Tennessee, USA, pp. 3128-3133.

[26] Y. Deng and S. K. Chang, "A G-Net Model for Knowledge Representation and Reasoning," *IEEE Transactions on Knowledge and Data Engineering*, vol.2, no. 3, September 1990, pp. 295-310.

[27] M. Wooldridge, *An Introduction to Multiagent Systems*, John Wiley and Sons, Ltd., 2002.

[28] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge, "Formalisms for Multi-Agent Systems," *The Knowledge Engineering Review*, vol. 12, no. 3, 1997.

[29] T. Finin, Y. Labrou, and J. Mayfield, "KQML as an agent communication language," *Software Agents*, Jeff Bradshaw, ed., MIT Press, Cambridge, 1997.

[30] M. J. Huber, S. Kumar, P. R. Cohen, and D. R. McGee, "A Formal Semantics for Proxy Communicative Acts," *Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages* (ATAL-2001), Seattle, Washington, USA, August 1-3, 2001.

[31] H. Xu and S. M. Shatz, "A Framework for Model-Based Design of Agent-Oriented Software," To appear in *IEEE Transactions on Software Engineering*, 2002.