ADK: An Agent Development Kit Based on a Formal Design Model for Multi-Agent Systems

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

The advent of multi-agent systems has brought us opportunities for the development of complex software that will serve as the infrastructure for advanced distributed applications. During the past decade, there have been many agent architectures proposed for implementing agent-based systems, and also a few efforts to formally specify agent behaviors. However, research on narrowing the gap between agent formal models and agent implementation is rare. In this paper, we propose a model-based approach to designing and implementing intelligent agents for multi-agent systems (MAS). Instead of using formal methods for the purpose of specifying agent behavior, we bring formal methods into the design phase of the agent development life cycle. Specifically, we use the formalism called agent-oriented G-net model, which is based on the G-net formalism (a type of high-level Petri net), to serve as the high-level design for intelligent agents. Based on the high-level design, we further derived the agent architecture and the detailed design for agent implementation. To demonstrate the feasibility of our approach, we developed the toolkit called ADK (Agent Development Kit) that supports rapid development of intelligent agents for multi-agent systems and we discuss the role of inheritance in agent-oriented development. As a potential solution for automated software development, we summarize the procedure to generate a model-based design of application-specific agents. Finally, to illustrate an application built on ADK, we present an air-ticket trading example.

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

1. Introduction

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. A multi-agent system (MAS) is a distributed and concurrent system that consists of a number of intelligent agents (Wooldridge, 2002). An intelligent agent is defined as an agent that at least has the following characteristics: autonomy, reactivity, proactiveness, and 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 respond to a changing environment. The idea of reactive agents is often based on work in ethology, the study of animal behavior. 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. Though a standalone agent may not need to interact with other agents, agent communication is a key characteristic for agents in a multi-agent system. The conversations among agents, normally conducted by sending and receiving messages, provide opportunities for agents to coordinate their activities and cooperate/compete 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. In contrast, agents distinguish different types of messages and use complex protocols, such as Contract Net protocols (Smith, 1980; Flores and Kremer, 2001), to collaborate or to negotiate. In addition, agents analyze these messages and can decide whether to execute the requested action (Wooldridge et al., 2000). 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.

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 agents in multi-agent systems. The approach is based on the formal agent model, called agent-oriented G-net model, introduced in earlier work (Xu and Shatz, 2001a; Xu and Shatz, 2001b) and subsequently described in more detail, including examples of model checking for design properties (Xu and Shatz, 2003). To bridge the gap between formal agent modeling and agent implementation, we extend this earlier work so that formal methods are integrated into the design phase of the agent development life cycle. Unlike most current research on formal modeling of agent systems or agent behavior (Wooldridge and Ciancarini, 2001), our agent model specifically serves as a high-level design for agent implementation, rather than just as a specification for agent behaviors. In other words, the formal model guides a software engineer by prescribing “how,” rather than “what,” to develop in terms of intelligent agents. Our formal agent model supports design modularization and inheritance. To show the feasibility of our approach, we highlight a system that provides a full class-library for the domain of intelligent agents for multi-agent systems. We call the development system ADK (Agent Development Kit). The high-level design, i.e., the agent-oriented G-net model, is based on the G-net formalism (Perkusich and de Figueiredo, 1997; Deng et al., 1993), which is a type of high-level Petri net (Murata, 1989). The significance of this model is that it explicitly supports asynchronous message passing among agents (Xu and Shatz, 2001a), and it supports inheritance for functional units defined in its internal structure (Xu and Shatz, 2001b). The functional units in this model not only include methods, as in the case of the 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 (Xu and Shatz, 2003).

The rest of this paper is organized as follows. In Section 2, we describe related work and highlight the relationships to our research. In Section 3, we first briefly review the agent-oriented G-net model, which has been previously proposed [Xu and Shatz, 2001b] and subsequently described in more details for design analysis [Xu and Shatz, 2003]. We then discuss the role of ADK in serving as a bridge between the formal agent model and the agent implementation platform. In Section 4, we describe the architectural design and detailed design of intelligent agents, and discuss the role of inheritance in agent development. We also summarize the procedures to design and implement application-specific agents for multi-agent systems. In Section 5, we use 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 6, we provide conclusions and our future work.

2. Related Work

There are three main strands of work to which our research is related, i.e., work on formal modeling of agent systems, work on building practical agent-based systems or developing tool kits for rapid development of agent systems, and work on narrowing the gap between agent formal models and implementation of agent-based systems.

Previous work on formal modeling of agent systems has been based on formalisms, such as Z, temporal logic, and Petri nets, to specify agent systems or agent behaviors. Luck and d'Inverno tried to use the formal language Z to provide a framework for describing the agent architecture at different levels of abstraction. They proposed a four-tiered hierarchy comprising entities, objects, agents and autonomous agents (Luck and d'Inverno, 1995). The basic idea for this is that all components of the world are entities with attributes. Of these entities, objects are entities with capabilities of actions, agents are objects with goals, and autonomous agents are agents with motivations. Fisher' work on Concurrent METATEM used temporal logic to represent dynamic agent behavior (Fisher 1995). Such a temporal logic is more powerful than the corresponding classic logic and is useful for the description of dynamic behavior in reactive systems. Fisher took the view that a multi-agent system is simply a system consisting of concurrently executing objects. Xu and his colleagues used Predicate/Transition (PrT) nets, which is a high-level formalism of Petri net, to model and verify multi-agent behaviors (Xu et al., 2002). Based on the PrT model, certain properties, such as parallel execution of multi-plans and plans' guarantee for the achievement of the goal, can be verified by analyzing the dependency relations among the transitions. More recently, PrT nets were used to model logical agent mobility (Xu et al., 2003). The proposed model for logical agent mobility specifies a mobile agent system that consists of a set of components and a set of (external) connectors. Pr/T nets were used because in a Pr/T net a token may carry structured data – the mobility modeling was based on the idea of "agent nets" being able to be routed, as tokens, within "system nets." Other efforts on formal modeling of agents focus on the design of modeling languages for conceptual design and specification of multi-agent systems. For instance, the modeling language DESIRE (framework for DEsign and Specification of Interacting REasoning component) is based on the philosophy of viewing a complex software system as a series of interacting components, and it is suited to the specification of multi-agent systems (Brazier et al., 1997). Similarly, SLABS (formal Specification Language for Agent-Based Systems) provides a way of specifying agent behaviors to enable software engineers to analyze agent-based systems before they are implemented (Zhu, 2001).

In summary, formal methods are typically used for specification of agent systems and agent behaviors. In other words, the primary purpose of the formal agent models is to define *what* properties are to be realized by the agent system – behavioral properties. In contrast, the formal agent model that we have proposed in (Xu and Shatz, 2003) provides a high-level design of multi-agent systems – it not only provides a conceptual framework for agent development, but it also aids a software engineer in understanding *how* to structure and implement an agent system. This is accomplished by explicitly identifying the major components and mechanisms in the design and showing how to derive a detailed design and corresponding implementation. A direct benefit of this approach is that it brings formal methods into the design phase, providing opportunities for formal verification of correctness of an agent design. We have shown ways of using analysis techniques, including model checking, to verify the correctness and certain properties of the formal agent model in our previous work (Xu and Shatz, 2003). Ideally, formal methods can be applied in each phase of the software development life cycle; however, to bring formal methods into the later phases (e.g., design and implementation) of the software development life cycle is not an easy task. For instance, to verify that an implemented agent system satisfies its original specification, Rao and Georgeff presented an algorithm for model checking BDI systems (Rao and Georgeff, 1993). However, how to derive the logical model for BDI logic from the concrete computational models used to implement the agents is not clarified (Wooldridge and Ciancarini, 2001). Recent work along this line includes Penczek and Lomuscio's attempt to use the semantic model of interpreted systems, and integrate it with the verification technique of bounded model checking for verification of multi-agent systems (Penczek and Lomuscio, 2003). Thanks to Petri nets' graphical modeling approach and its similarity with the UML modeling technique (Saldhana et al., 2001), we argue that Petri nets provide a reasonable and promising way to bring formal methods into the design phase. With refinement of our original agent-oriented G-net models (Pan, 2002), our approach supports formal design of agent-oriented software.

A second strand of related work is the building of practical agent-based systems or tools for rapid development of agent systems. During recent years, many agent architectures have been proposed. For instance, JAM (Java Agent Model) 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 (Huber 1999). Based on the BDI theories (Kinny et al., 1996), 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 (Jave Agent Kernel) intelligent agent framework proposed by the *Agent Oriented Software Group* brings the concept of intelligent agents into the mainstream of commercial software engineering and Java (Howden et al., 2001). JACK is designed as a set of lightweight components with high performance and strong data typing. Paradigma has been implemented to support the development of agent-based systems (Ashri and Luck, 2000). It relies on a formal agent framework, i.e., Luck and d'Inverno's formal agent framework (Luck and d'Inverno, 1995), and is implemented by using recent advances in Java technology. Although the above agent architectures use formal agent models as conceptual guidelines, the formal methods serve as agent specifications rather than formal designs.

Some other efforts had tried to provide a rapid prototyping development environment for the systematic construction and deployment of agent-oriented applications. A typical example is the Zeus MAS framework developed by British Telecom Labs (Nwana et al., 1999). The MAS development environment based on Zeus MAS framework consists of an API, a code generator, agent and society monitoring tools, and programming documentation. A complete Zeus agent has a coordination engineer enabling functional behavior organized around conversation protocols, a planner that schedules sub-goal resolution, an engine for rule-based behavior, and databases to manage resources, abilities, relationships between agents, tasks, and protocols. More recently, many agent frameworks have been proposed for developing agent applications in compliance with the FIPA specifications (FIPA 2002) for interoperable intelligent agents in multi-agent systems. Examples of such efforts are JADE (Java Agent DEvelopment framework) (Bellifemine et al., 1999), FIPA-OS agent platform (Poslad et al., 2000), and the current Zeus platform (Nwana et al., 1999). The major difference between the above work and our approach is that most of the existing agent architectures attempt to provide a comprehensive set of agent-wide services that can be utilized by application programmers; however, these services are usually made available through an ad-hoc architecture that is highly coupled. Application programmers must face a steep learning curve for such systems due to a lack of explicit control flow and modularization. In contrast, our approach provides programmers a set of loosely coupled modules, an explicit control flow, and a clean interface among agents. We believe that our approach can significantly flatten a programmer's learning curve, and ease the work load for developing application-specific agents. Another difference between the above work and our approach is that most of the agent architectures that originated from industry aim to provide practical platforms or toolkits for agent development; therefore, unlike our approach there is not the direct motivation for an agent design that supports formal analysis and verification. We have discussed the use of some analysis techniques including model checking in our previous work (Xu and Shatz, 2003). Meanwhile, most of the existing systems use object-oriented languages, such as Java, but without considering how to use object-oriented mechanisms effectively in developing agent-oriented software. In contrast, our approach carefully considers the role of inheritance in agent-oriented development, and discusses which components of an agent could be reused in a subclass agent. This treatment of inheritance in agent-oriented software engineering is based on previous work (Crnogorac et al., 1997), but our approach emphasizes on reuse of functional components rather than mental states. Finally, although our current version of ADK does not strictly follow the FIPA specifications, we have designed our agent model with standardization in mind. Further work (Pan, 2002) on this prototype has shown that it is fairly straightforward to extend our agent design and development kit to a level of detail that is compliant with the FIPA specifications (FIPA, 2002).

Previous efforts on narrowing the sizable gap between agent formal models and agent-based practical systems can be summarized as follows. 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 (Fisher, 1995). Vasconcelos and his colleagues have tried to provide a design pattern for skeleton-based agent development (Vasconcelos et al., 2002), 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 (Rodriguez-Aguilar et al., 1999). 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 (D'Inverno et al., 1997), 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. Other efforts have attempted to start with specific deployed systems and provide formal analyses of them. For instance, d'Inverno and Luck tried to move backwards to link the system specification based on a simplified version of dMARS (distributed Multi-Agent Reasoning System) to the conceptual formal agent framework in Z, and also to provide a means of comparing and evaluating implemented and deployed agent systems (D'Inverno and Luck, 2001).

In contrast to the above approaches, we have tried to bring formal methods directly into the design phase, and to let the formal agent model serve as the high-level design for agent implementation. In particular, we use the agent-oriented G-net model to define the agent structure, agent behavior, and agent functionality for intelligent 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 agents closely follows the agent-oriented G-net model. By supporting design reuse, our approach follows the basic philosophy of *Model Driven Architecture (MDA)* (Siegel et al., 2001) that is gaining popularity in many communities, for example UML.

3. A Framework for Agent-Oriented Software

3.1 G-Net Model Background

Many researchers have suggested object-based formal methods using high-level Petri nets. Formalisms such as LOOPN++ (Lakos and Keen, 1994), CO-OPN/2 (Buchs and Guelfi, 2000), and G-nets (Perkusich and Figueiredo, 1997; Deng et al., 1993) were proposed to extend the Petri net formalism into so-called object

Petri nets. Among these models, G-nets have been proposed with the motivation of integrating Petri net theory with the software engineering approach for system design (Deng et al., 1993). A notable benefit of using G-nets is its modular and object-based approach for the specification and prototyping of complex software system. We assume that the reader is familiar with the basic concepts of Petri nets (Murata, 1989). 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 G-net models. G-nets serve as the foundation for our agent-oriented model, which is highlighted in the next subsection.

A G-net system is composed of a number of G-nets, each of them representing a self-contained module or object (Perkusich and Figueiredo, 1997; Deng et al., 1993). 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. Formally, a G-net is a tuple *G = (GSP, IS)*, where

1. $GSP \in G$ is defined by *(NID, MS, AS)*, where *G.NID* is a unique identification of G-net *G*, *G.MS* is a set of methods specifying the functions, operations or services defined by the net, and *G.AS* is a set of attributes specifying the state variables of the model.
2. $IS \in G$ is the internal structure of G-net *G*. In *G.IS*, 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'*.

G-nets were initially proposed to represent a module or an object rather than an abstraction of a set of similar objects, i.e., class. In a recent paper (Xu and Shatz, 2000), 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.

The token movement in a G-net object is similar to that of original G-nets (Perkusich and Figueiredo, 1997; Deng et al., 1993). A token *tkn* is a triple *(seq, sc, mtd)*, where *seq* is the propagation sequence of the token, $sc \in$ {**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. 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 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 can become enabled and fire. Examples and further details about G-net models can be found in references (Perkusich and Figueiredo, 1997; Deng et al., 1993; Xu and Shatz, 2000).

3.2 Review of Agent-Oriented G-Net Model

Although the G-net model works well in object-based design, it is not sufficient for 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.

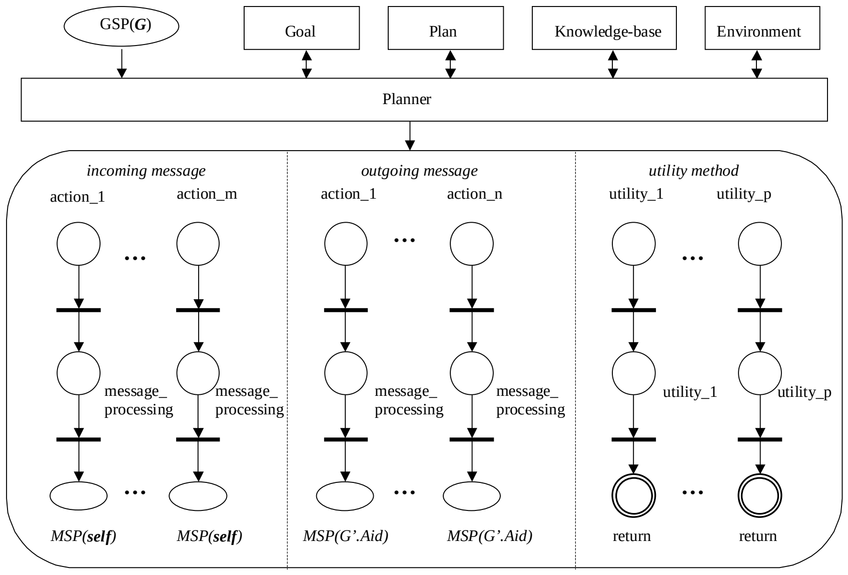


Figure 1. Agent-oriented G-net model – from (Xu and Shatz, 2003)

To support agent-oriented design, we extended G-nets to support modeling an agent class[1] (Xu and Shatz, 2003). 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 1, the five special modules are the *Goal* 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 (Kinny et al., 1996). The *Goal* module consists of a goal set that specifies the goal domain and goal states. The *Plan* module consists of a set of plans that are associated with a goal or a subgoal. Each goal or subgoal may associate with more than one plan, and the most suitable one will be selected to achieve that goal or subgoal. A *Knowledge-base* module describes the information about the agent's internal state, its environment, and interaction protocols. The *Environment* module is an abstract model of the environment, i.e., the model of the outside world of an agent. Finally, 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 (Finin et al., 1997; Huber et al., 2001), 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.

[1] 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.

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.

The *Internal Structure (IS)* of an agent-oriented G-net consists of three sections: *incoming message*, *outgoing message*, and *utility method*. The *incoming/outgoing message* section defines a set of *Message Processing Units (MPU)*, which corresponds to a set of communicative acts. Each *MPU*, labeled as *action_i* in Figure 1, is used to process incoming/outgoing messages and execute any necessary actions before or after the message being processed. Finally, 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 (Wooldridge et al., 2000; Iglesias et al., 1998). 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 (FIPA, 2002) or KQML (Odell et al., 2001; Finin et al., 1997; Huber et al., 2001). Note that in Figure 1, 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 (Xu and Shatz, 2001a). When a token reaches an *MSP* (represented as an ellipsis in Figure 1), 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 utility methods defined in the internal structure of an agent-oriented G-net model. Further details on the design and operation of the *Planner* module are outside the scope of focus for this paper. A detailed description for this module can be found in earlier work (Xu and

Shatz, 2001a; Xu and Shatz, 2001b; Xu and Shatz, 2003). 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 as demonstrated in other research that focuses on the issues of knowledge representation and reasoning (Deng and Chang, 1990).

3.3 From Formal Agent Model to Agent Implementation

We now turn to the fundamental contribution of this paper – the principles and practice of a proposed Agent Development Kit (ADK). ADK 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 high-level design for intelligent agents. Specifically, the key components or mechanisms defined in the agent-oriented model serve as building blocks of our agent development kit.

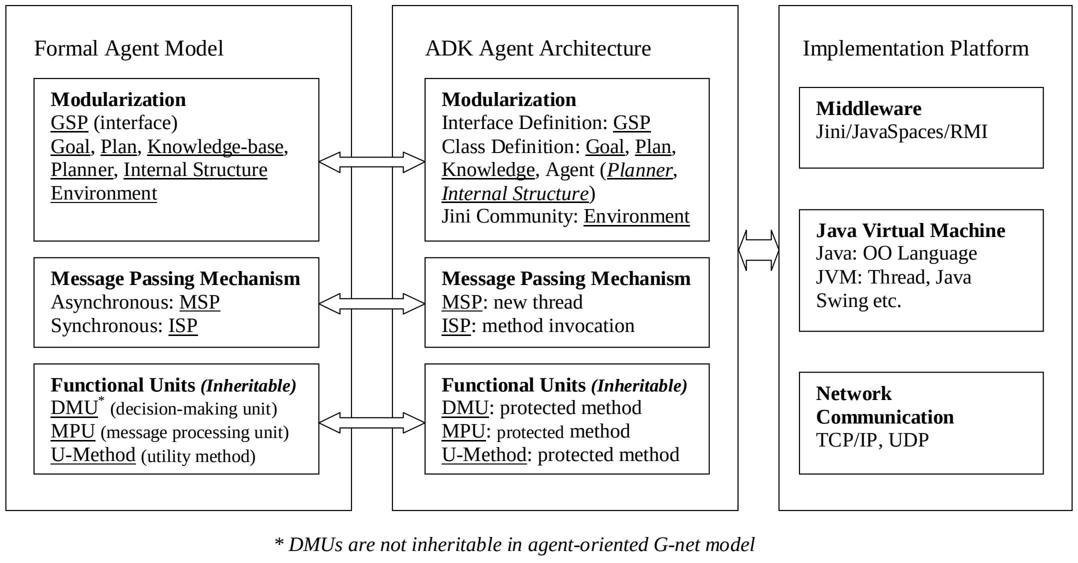

Formal Agent Model
Modularization
GSP (interface)
Goal, Plan, Knowledge-base, Planner, Internal Structure Environment
Message Passing Mechanism
Asynchronous: MSP
Synchronous: ISP
Functional Units (Inheritable)
DMU* (decision-making unit)
MPU (message processing unit)
U-Method (utility method)
ADK Agent Architecture
Modularization
Interface Definition: GSP
Class Definition: Goal, Plan, Knowledge, Agent (Planner, Internal Structure)
Jini Community: Environment
Message Passing Mechanism
MSP: new thread
ISP: method invocation
Functional Units (Inheritable)
DMU: protected method
MPU: protected method
U-Method: protected method
Implementation Platform
Middleware
Jini/JavaSpaces/RMI
Java Virtual Machine
Java: OO Language
JVM: Thread, Java Swing etc.
Network Communication
TCP/IP, UDP
* DMUs are not inheritable in agent-oriented G-net model

Figure 2. The role of ADK between formal agent model and implementation platform

As Figure 2 shows, the role of ADK is to serve as a bridge between the formal agent model and the agent implementation platform. Between the formal agent model and the ADK agent design architecture, there is a clear mapping of agent components and mechanisms. Thus we claim that the formal agent model can be interpreted as an agent design model; thus the model reveals to the software engineers not only the agent properties and behaviors, but more importantly, the agent architecture.

The key components and mechanisms defined in the formal agent model and their mappings to the components and implementation strategies in the ADK agent architecture 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. The *GSP* place in the formal model is defined as the only interface for agent communication, and this is carried forward in the ADK agent design architecture. The *Goal*, *Plan*, and *Knowledge-base* modules are based on the BDI agent model (Kinny et al., 1996) that is a conceptual model for intelligent agents. These modules are mapped to the class definitions of *Goal*, *Plan* and *Knowledge* in the ADK agent architecture. 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. These two modules are defined as two sections in the definition of the *Agent* class. Finally, the *Environment* module in the formal agent model will be implemented as the Jini community in ADK.

Second, the message passing mechanisms are defined in two cases: 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 (Xu and Shatz, 2001a). 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. Straightforwardly, the *ISP* mechanism maps to method invocation in ADK, and *MSP* maps to a new thread on platforms such as JVM.

Third, the formal agent model defines the functional units as inheritable components. As methods are defined as inherited units in object-oriented design, all *U-Methods* (Utility Methods) and *MPU*s (Message Processing Units) could be inherited from an agent superclass to an agent subclass. In ADK, the functional units, including *DMUs* (Decision-Making Units), which are defined in a superclass, are implemented as protected methods that can be reused by their subclasses.

As shown on the right hand side of Figure 2, the implementation platform provides the standard computer technologies, such as the Jini middleware (Edwards, 1999; Arnold et al., 1999) 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 therefore, we can concentrate on high-level communication protocols, such as price-negotiation protocol. In summary, ADK represents the design and implementation of intelligent agents for multi-agent systems, and it refines the formal agent model and derives the detailed design as will be discussed in Section 4.

4. Design of Intelligent Agents

4.1 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 (Edwards, 1999; Arnold et al., 1999). Such a collection of services is called a Jini community as shown in Figure 3, 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.

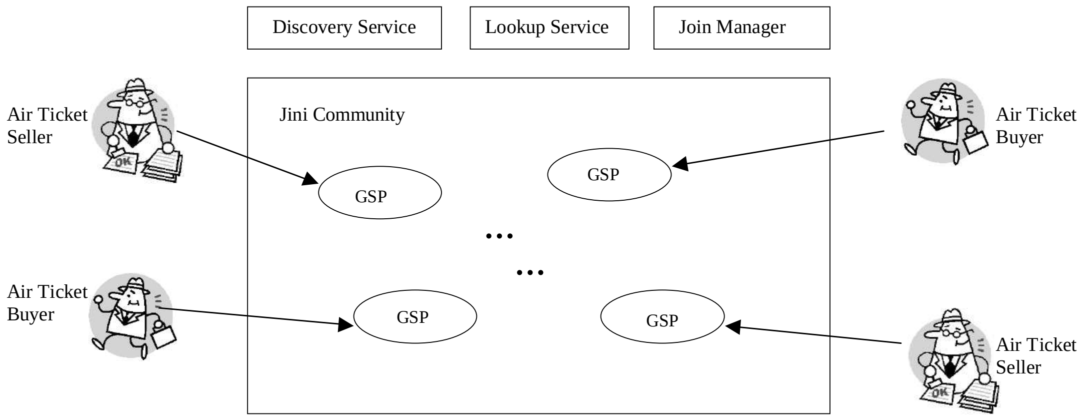


Figure 3. The Jini community with agents of *AirTicketSeller* and *AirTicketBuyer*

In designing the ADK, we use Jini as a middleware for agents to find each other and to communicate with each other. Each agent is designed as both a service provider and a service consumer. Since agents only interact with each other through asynchronous message passing, the service provided by an agent through

Jini is designed as an interface to let other agents send asynchronous messages to that agent, and the agent who sends out the messages becomes the service consumer. This approach is consistent with the agent-oriented G-net model, in which the *GSP (Generic Switch Place)* is defined as the only interface among agents (Xu and Shatz, 2003). Thus, we design the schema for an agent interface as in Table 1.

Table 1

SCHEMA FOR AN AGENT INTERFACE

```
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 4.2, 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*.

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 3, 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”.

4.2 A Pattern for Intelligent Agents

Figure 4 shows the ADK architectural design for intelligent agents. By comparing this figure to the agent-oriented G-net model in Figure 1, one can observe how the agent model drives the agent design. One obvious variation is that the *GSP* place of an agent model becomes a part of the environment module, which is the Jini community, in the ADK agent design architecture. In the design architecture, each agent is composed of its *GSP* component, which serves as the agent's interface element, and its action component (consisting of the following major elements: *Goal*, *Plan*, *Knowledge-base*, *Planner*, and *Internal Structure*). Note that Figure 4 explicitly shows only the action component for one agent, agent *B*. The environment module contains the interface element for agent *B*, as well as some other interface elements (e.g., for agent *A*). The directed arcs shown in Figure 4 represent only a sample of the logical connections between the various elements – for example we explicitly see the connection from agent *B*'s interface to its planner module, and from agent *B*'s outgoing message processing unit to agent *A*'s interface (under the assumption that agent *B* does send messages to agent *A*).

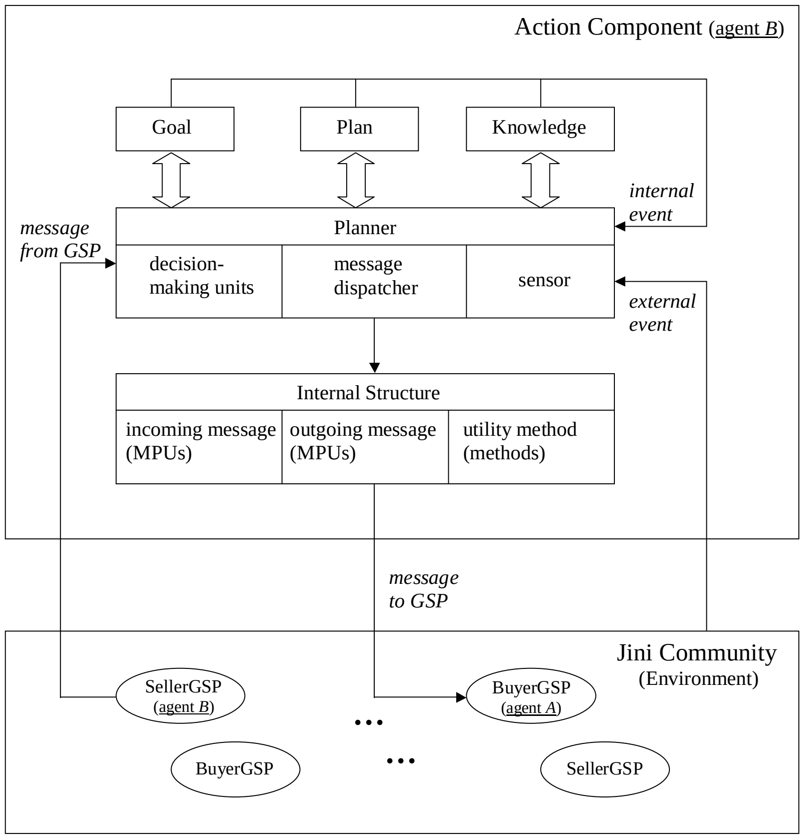


Figure 4. The ADK architectural design of intelligent agents

Currently, we use a simplified version of the environment module in ADK, in which case the only external events of concern are those related to agents entering and/or leaving the Jini community. In future design versions, we can extend the environment module to include other events, such as network topology changes and user interventions. 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 ADK the sensor in the *Planner* module is implemented to only capture external events.

Referring again to Figure 4, we can observe that when agent *A* wants to converse with agent *B*, it sends a message to the *GSP* of agent *B* in the Jini community (but, this connection is not explicitly shown in Figure 4 since agent *A*'s action component is not shown). 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, the *MPU*s and the *U-Methods* (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. Unlike the agent-oriented G-net model, methods defined in the planner module can also be inherited optionally if a subclass agent chooses to reuse or refine the reasoning mechanisms defined in its superclass. This treatment is practical if we need to derive a subclass agent with similar behavior to its superclass – for instance, to derive a domestic air ticket seller agent class from a general air ticket seller agent class.

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 4, we now proceed to describe the detailed design of intelligent agents for multi-agent systems. 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 and present first the pattern for the *Agent* class, which is a superclass for application-specific agents. The design schema for application-specific agents will be introduced in Sections 4.3 and 4.4. 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 (Pressman, 2001). 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 is shown in Table 2 in a form of Java pseudocode. As shown in Table 2, the *Agent* class is defined as a subclass of *MiddlewareSupport* (defined in Section 4.1) 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 method *asynMessagePassing()* of the message receiver (the callee) generates the *messageProcessThread* thread, which will dispatch the message to a *MPU* that is defined in the callee's incoming message section. Once the new thread is created, the *asynMessagePassing()* method can immediately return control to its caller, so that the message sender (the caller) does not need to wait for the message to be processed. The caller can proceed to execute other tasks.

Table 2

A PATTERN FOR INTELLIGENT AGENTS

```
public class Agent extends MiddlewareSupport {
    private static final String PRODUCT = "Agent";
    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_Hello(Message message) {…}
     …
     // outgoing message section – a set of message processing units
     protected void MPU_Out_Hello(Message outgoingMessage) {…}
     …
     // utility method section – a set of private utility methods
     initAgent(String[] args) {…}
     protected void autonomousRun() {…}
     protected void other_Method_1() {…}
     …

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

Corresponding to the three modules (*Goal*, *Plan* and *Knowledge*) in the architectural design of intelligent agents (Figure 4), 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 or redefined 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 Table 2.

The reactivity of an agent can be designed through the Jini's notification facility. 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 (by keeping a list of interested agents locally, it can also decrease the network traffic). In Table 2, 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 the Jini. Thus, an application class, such as a seller agent class or a buyer agent class, which will be defined as a subclass of the *Agent* class, 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 agents in Figure 4, 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_x()*, *MPU_Out_y()* or *Method_k()* in Table 2. We only implemented the *MPU_In_Hello()* and *MPU_Out_Hello()* in the *Agent* class, which allows instances of the *Agent* class or any of its subclasses to greet with each other. Further protocol related *MPUs* shall be defined in agent subclasses. 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 Table 2, and is executed after the agent is initialized with the method *initAgent()*.

4.3 Inheritance in Agent-Oriented Development

Inheritance in agent-oriented programming has been studied in terms of reusing mental states such as goal, plan and knowledge (Crnogorac et al., 1997). We argue that since an agent maintains a dynamic list of goals and plans, and acquires most of its knowledge during its lifetime, to inherit mental states is not appropriate. Furthermore, agents are autonomous with different goal-directed behavior. For instance, in the class hierarchy of Figure 5, an air ticket seller agent and a book seller agent shall have different goals and plans, and more practically, they may have different negotiation strategies and reasoning mechanisms. Thus, the class hierarchy in Figure 5 shall only imply the reuse of superclass' functional mechanisms, for instance, the communication mechanism. Since inheritance happens at the class level, new knowledge acquired, new plans made, and new goals generated in an agent object (e.g., an air ticket seller), cannot be inherited by a subclass agent object (e.g., a domestic air ticket seller). In contrast, most of the functional mechanisms, for instance the function of comparing prices or selecting the ticket with shortest travel time, can be reused. Optionally, the domestic air ticket seller may also reuse the negotiation strategies adopted by an air ticket seller, but practically it may have its own specific strategies. As a result, we need to allow a subclass agent to inherit any reasoning mechanisms defined in its superclass agent, but also allow such a subclass agent to redefine or refine these mechanisms.

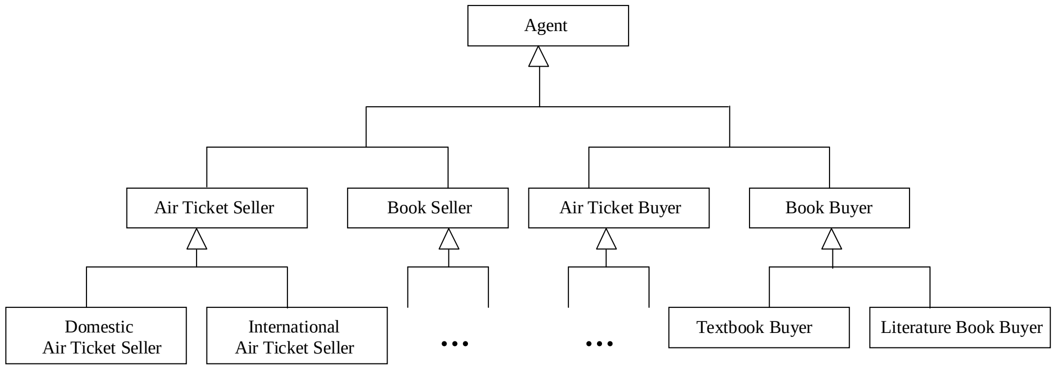


Figure 5. The class hierarchy diagram of agents in an electronic marketplace

Figure 6 shows the inheritance relationship between the classes defined in ADK and classes derived from the *Agent* class. 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 agents for multi-agent systems. The classes below the dashed line are derived classes that represent specific intelligent agents in a multi-agent system. This figure shows that both the air ticket seller agent and the air ticket buyer agent may reuse the functional mechanisms and reasoning mechanisms defined in the superclass *Agent*. Especially, air ticket seller agents and air ticket buyer agents may communicate with each other through Jini. We do not need to deal with this issue again in the design of these two classes, since all needed functionality for communication through Jini has been implemented in the *Agent* class and can be reused by its subclasses. The event-driven feature is also inherited by the air ticket seller agents and the air ticket buyer agents. In other words, a designer of subclasses of the *Agent* class does not need to be concerned with this feature, since subclasses automatically have this feature inherited from their superclass, i.e., the *Agent* class. In addition, the air ticket seller agent and air ticket buyer agent may reuse the default reasoning mechanisms defined in the *Agent* class. The default reasoning mechanism is defined as a search through a goal tree that achieves each subgoal, with associated plans, in a depth-first search order.

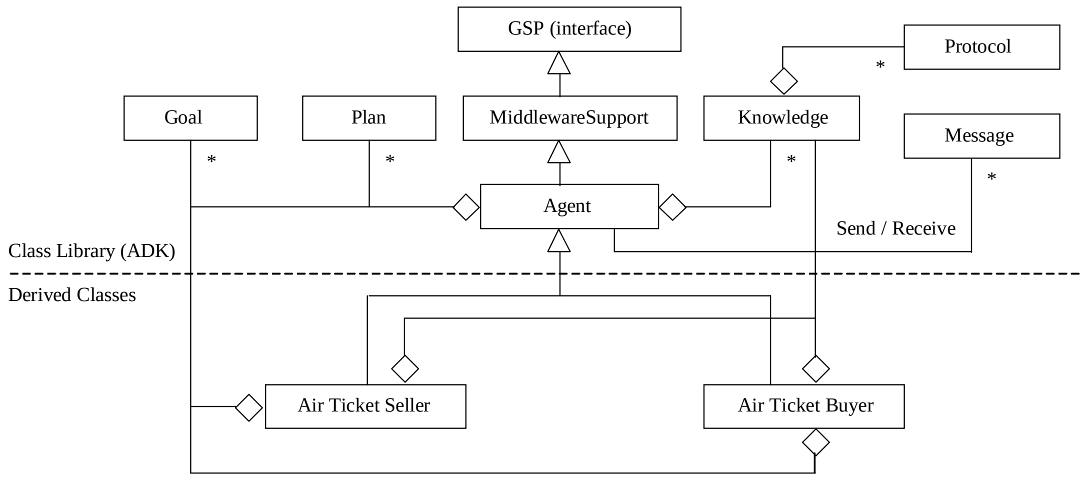


Figure 6. Classes defined in ADK and derived classes of the *Agent* class

Though most of the features defined in the *Agent* class can be reused, each subclass of the *Agent* class shall associate with the *Goal*, *Plan* and *Knowledge* class directly. This implies that any goal, plan or knowledge defined in a superclass cannot be inherited by its subclasses. This design is consistent with the high-level design of agent-oriented G-net models, in which the *Goal*, *Plan* and *Knowledge-base* modules of the superclass are disabled when the inheritance mechanism is invoked. The *Goal*, *Plan*, and *Knowledge* classes define the basic (default) structure for their corresponding modules. Subclasses of the *Agent* class may either reuse these structures or define their own. Obviously, if these classes are redefined, the reasoning mechanisms shall also be redefined in the agent subclasses.

This approach derives the template (pattern) for application-specific agent design, which is defined as a subclass of the *Agent* class in ADK. The template is shown in Table 3. In this template, we use the definitions of the *Goal*, *Plan*, and *Knowledge* classes that are defined in ADK, but it is worth noting that designers can also define their own classes for these modules. Alternatively, they may refine these classes (defined in ADK) by subclassing them and inheriting their default structures.

Table 3

DESIGN OF APPLICATION-SPECFIC AGENTS

```
public class ApplicationSpecificAgent extends Agent {

    /***********************************************
     * Class Variables for Knowledge, Goal and Plan *
    /***********************************************/
    Goal myGoals; // committed goals, redefinition of Goal class is optional
    Plan myPlans; // plans, redefinition of Plan class is optional
    Knowledge myKnowledge; // knowledge-base, redefinition of Knowledge
                           // class is optional
    …

    /***********
     * Planner *
     ***********/
    protected void dispatchMessage(Message message) {…}  // refinement
                                                         // or redefinition
    protected Message makeDecision(Message message) {…}  // refinement
                                                         // or redefinition
    protected void updateMentalState() {…)               // refinement
                                                         // or redefinition
    …

    /**********************
     * Internal Structure *
     **********************/
    // incoming message section – a set of message processing units
    protected void MPU_In_1(Message message) {…}         // new definition
    …

    // outgoing message section – a set of message processing units
    protected void MPU_Out_1(Message outgoingMessage) {…}// new definition
    …

    // utility method section – a set of private utility methods
    protected void initAgent(String[] args) {…}          // refinement
                                                         // or redefinition
    protected void autonomousRun() {…}                   // refinement
                                                         // or redefinition
    protected void other_Inherited_Method_1() {…}        // refinement
                                                         // or redefinition
    …
    protected void other_New_Method_1() {…}              // new definition
    …

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

In the *Planner* section of the *ApplicationSpecificAgent* class, all the decision-making units (e.g., *makeDecision* and *updateMentalState*) inherited from those defined in the *Agent* class can be refined or redefined. In an extreme case, this section can be left blank, if the default reasoning mechanisms defined in the *Agent* class are reused. In the *Internal Structure* section, sets of *MPU*s are defined corresponding to a set of protocols. For instance, from a price-negotiation protocol, we can derive a set of *MPU*s, such as *request-price*, *propose*, *accept-proposal*. A description of how to derive *MPU*s from interaction protocols has been developed (Xu and Shatz, 2003), but this level of detail is outside the scope of this paper.

In the *utility method* section, methods (i.e., *U-Methods*) are defined as "protected" so that they can be further inherited by their subclasses. In addition to refining or redefining the two outstanding methods, *initAgents()* and *autonomousRun()*, we can refine or redefine any inherited methods defined in the utility method section of the *Agent* class. Furthermore, new application-specific functions shall be added here.

One advantage of our model-based approach is its support for the principle of "separation of concerns," in particular the separation of agent mental states and agent communication capabilities. Therefore, it is possible for us to choose some existing implementation schema of intelligent agents (agent with or without communication capabilities) to design and implement intelligent agents for multi-agent systems. For instance, we can choose the *Task Representation Language (TRL)* to support knowledge representation and agent reasoning (Ioerger et al., 2000), or we can use Petri nets to model the mental state of agents for multi-agent simulation (Yen et al., 2001). Alternatively, we can, and do, use a more commonly used intelligent agent model – the *Belief-Desire-Intention (BDI)* model (Kinny et al., 1996). 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 (Vidal et al., 2001). 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 the relationships between the key classes defined for communication capabilities and agent mental states are illustrated in Figure 7.

As Figure 7 shows, two key classes for communication capabilities 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 appropriate plan to achieve a goal or a subgoal. Figure 8 provides a visual example of these objects. Protocol instances are defined inside the *Knowledge* class. Therefore, the *Knowledge* class may use/update protocols.

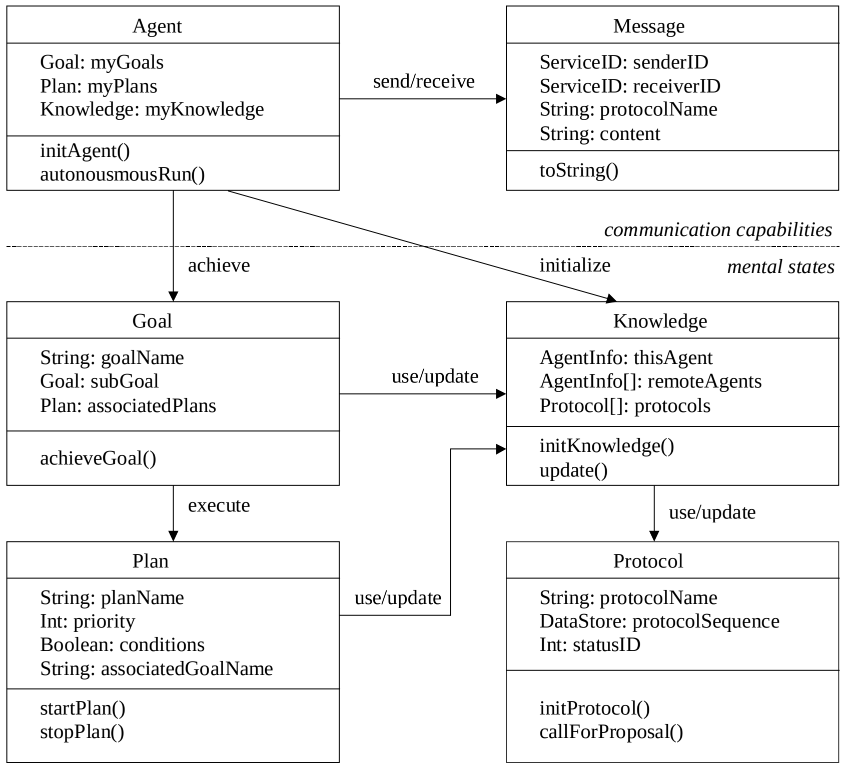


Figure 7. Relationship between classes defined for communication capabilities and mental states

The *Agent* class or any subclasses of the *Agent* class, e.g., an air ticket buyer 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*. For instance, an air ticket buyer agent may have a committed goal to buy a number of air tickets from “Chicago” to “Dayton” on a certain date from some air ticket seller agents. The air ticket buyer agent may choose a plan with the highest priority, i.e., the most suitable one, to achieve this goal. Based on the knowledge about the air ticket information, the existence of air ticket seller agents in the Jini community, and the interaction protocols needed for query and negotiation with those air ticket seller agents, the air ticket buyer agent can proceed to purchase its needed air tickets. The list of committed goals and the set of plans may be updated at run time. 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 or an *ApplicationSpecificAgent* object, and may be updated at run time by a

Goal or *Plan* object. The intelligent 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.

4.4 An Agent Development Process

The purpose of the proposed agent design architecture and agent design patterns is to ease the programmer's effort to develop applications of intelligent agents for multi-agent systems. As we mentioned before, an application-specific agent, such as an air ticket seller agent, could be defined as an agent subclass of the *Agent* class. Since the *Agent* class shown in Table 2 provides the basic functionality of intelligent agents as well as the agent implementation framework, what we need to do for developing an application-specific agent is to inherit the functional units and the behaviors of the *Agent* superclass and fill out certain sections in the pattern for application-specific agent, as shown in Table 3. In addition, we need to redefine or define subclasses of the *Goal*, *Plan*, and *Knowledge* classes that are defined in ADK to meet certain behavioral requirements of agent intelligence.

As a summary, we now briefly describe the generic procedure to develop an application-specific agent for multi-agent systems. In Section 5, 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* (Table 3) 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* (Table 3). 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. Refine the *Knowledge* class, including the *Protocol* 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* (Table 3) for that agent.

5. An interaction protocol ρ serves as a template for agent conversation. Based on ρ, 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 Table 3. Refer to (Xu and Shatz, 2001a; Xu and Shatz, 2001b) for a detailed description for transforming from ρ to ψ.
6. Refine the decision-making units defined in the *ApplicationSpecificAgent* class, if needed. Examples of decision-making units include functions like *makeDecision()* and *updateMentalState()*.

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.

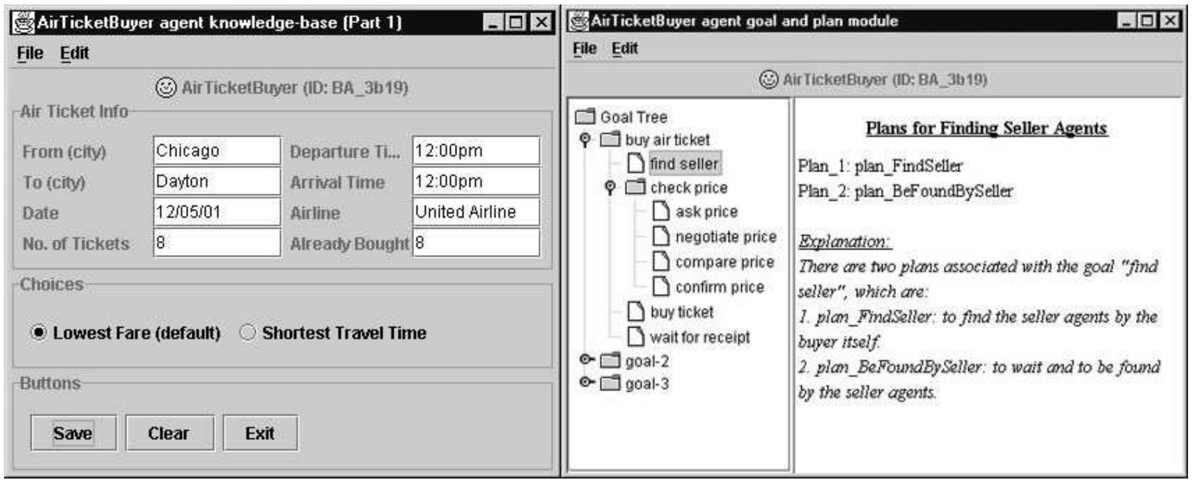
It should be mentioned that the above procedures could be automated, or partially automated by providing a development environment, to ease the programmers' work. This is also one of the major motivations of our ADK project. An *Agent Development Environment (ADE)*, which encompasses the ADK, is envisioned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

5. A Case Study: Air-Ticket Trading

We can now discuss an example that shows how to develop intelligent agents upon the ADK platform. Suppose we wish to design and implement a multi-agent system for air ticket trading. The multi-agent system will include two types of agents, air ticket seller agents and air ticket buyer agents. According to the procedures described previously, 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 8. 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.

The protocols used for 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 of certain types of air tickets. For instance, the buyer may ask if the seller has tickets from "Chicago" to "Dayton." 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*."



From (city)	Chicago	Departure Ti...	12:00pm
To (city)	Dayton	Arrival Time	12:00pm
Date	12/05/01	Airline	United Airline
No. of Tickets	8	Already Bought	8

● Lowest Fare (default) ○ Shortest Travel Time

Plans for Finding Seller Agents

Plan_1: plan_FindSeller
Plan_2: plan_BeFoundBySeller

Explanation:
There are two plans associated with the goal "find seller", which are:
1. plan_FindSeller: to find the seller agents by the buyer itself.
2. plan_BeFoundBySeller: to wait and to be found by the seller agents.

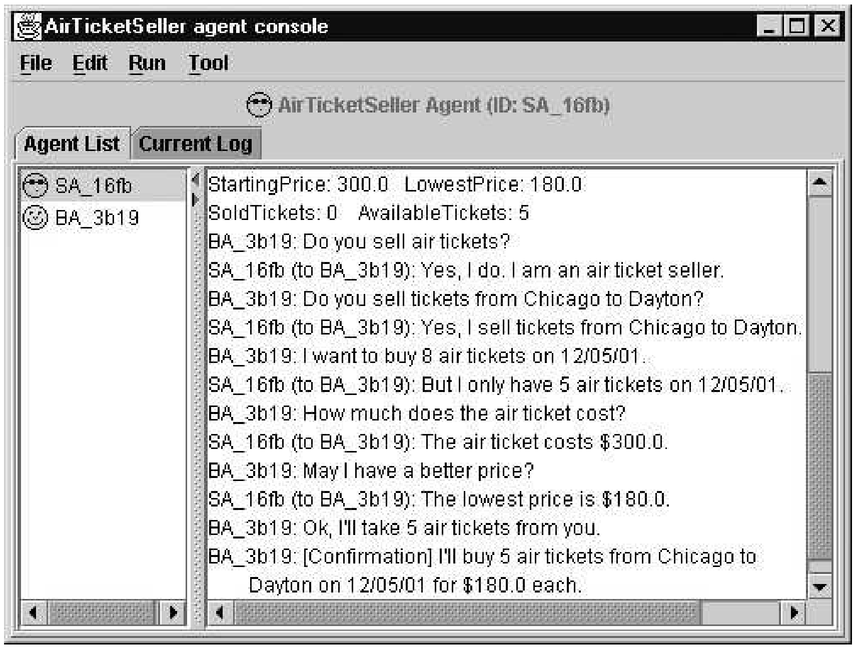
Figure 8. User Interface of the *Knowledge-base*, *Goal* and *Plan* module

The following pseudo-code gives some examples of how to "fill out" certain sections of the implementation pattern provided by the *ApplicationSpecificAgent* 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) {…}
…
```

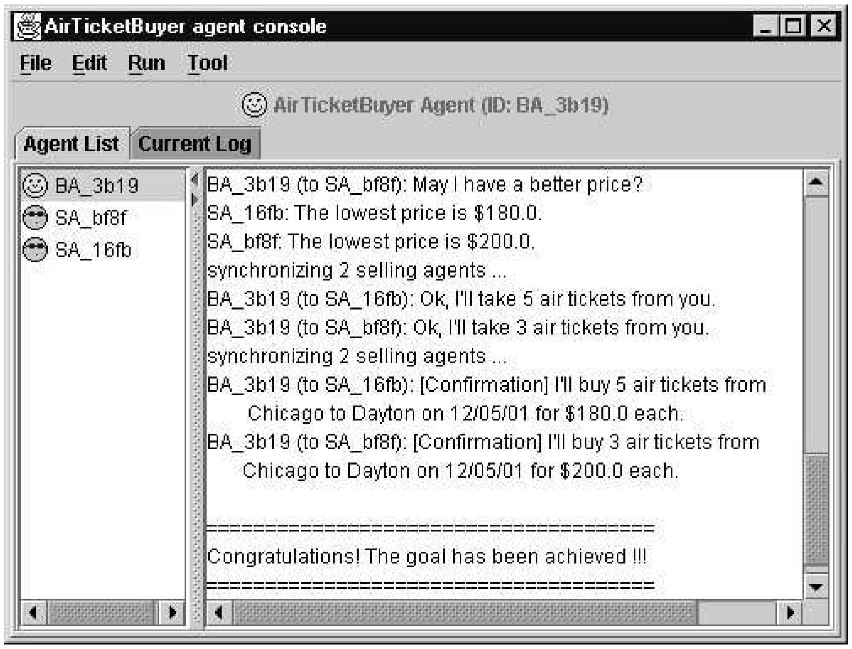
The *Knowledge-base* of a seller or buyer agent includes two parts, which provides information about the agent itself and information about other agents. For instance, the *Knowledge-base* of the buyer agent should include ticket information for the type of tickets that the buyer agent wants to buy (as shown on the left hand side of Figure 8), and ticket information for the type of tickets that other seller agents may hold. Other information, such as the interaction protocols and agent states, 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.


AirTicketSeller agent console
File Edit Run Tool
AirTicketSeller Agent (ID: SA_16fb)
Agent List
Current Log
SA_16fb
BA_3b19

```
StartingPrice: 300.0   LowestPrice: 180.0
SoldTickets: 0    AvailableTickets: 5
BA_3b19: Do you sell air tickets?
SA_16fb (to BA_3b19): Yes, I do. I am an air ticket seller.
BA_3b19: Do you sell tickets from Chicago to Dayton?
SA_16fb (to BA_3b19): Yes, I sell tickets from Chicago to Dayton.
BA_3b19: I want to buy 8 air tickets on 12/05/01.
SA_16fb (to BA_3b19): But I only have 5 air tickets on 12/05/01.
BA_3b19: How much does the air ticket cost?
SA_16fb (to BA_3b19): The air ticket costs $300.0.
BA_3b19: May I have a better price?
SA_16fb (to BA_3b19): The lowest price is $180.0.
BA_3b19: Ok, I'll take 5 air tickets from you.
BA_3b19: [Confirmation] I'll buy 5 air tickets from Chicago to
      Dayton on 12/05/01 for $180.0 each.
```

Figure 9. User interface of the seller agent *SA_16fb*

The user interface of a seller agent is designed as a console window as shown in Figure 9. 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 also provides a set of tools, such as those to lookup existing services, to test message sending/receiving, and to edit agent properties. Figure 9 shows an example of air ticket trading process. In Figure 9, 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. It is worth noting that although we have used natural language in this example, agents do not talk with each other in natural language. The sentences in natural language have been generated based on the values carried with messages and the semantic of their corresponding interaction protocols.


AirTicketBuyer agent console
File Edit Run Tool
AirTicketBuyer Agent (ID: BA_3b19)
Agent List
Current Log
BA_3b19
SA_bf8f
SA_16fb

```
BA_3b19 (to SA_bf8f): May I have a better price?
SA_16fb: The lowest price is $180.0.
SA_bf8f: The lowest price is $200.0.
synchronizing 2 selling agents ...
BA_3b19 (to SA_16fb): Ok, I'll take 5 air tickets from you.
BA_3b19 (to SA_bf8f): Ok, I'll take 3 air tickets from you.
synchronizing 2 selling agents ...
BA_3b19 (to SA_16fb): [Confirmation] I'll buy 5 air tickets from
      Chicago to Dayton on 12/05/01 for $180.0 each.
BA_3b19 (to SA_bf8f): [Confirmation] I'll buy 3 air tickets from
      Chicago to Dayton on 12/05/01 for $200.0 each.

=====================================
Congratulations! The goal has been achieved !!!
=====================================
```

Figure 10. User interface of the buyer agent *BA_3b19*

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 (Edwards, 1999; Arnold et al., 1999).

In Figure 10, 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.

6. 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 high-level design for agent development, thus we bring formal methods directly into the design phase of the agent development life cycle. Also the role of inheritance in agent development has been carefully discussed. Based on the architectural design and the detailed design of a generic intelligent agent, we developed the ADK as a class library that supports designing and implementing applications of intelligent 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 application-specific intelligent 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.

Acknowledgments: 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. We thank Zhaoxiao Hu and Yan Pan for proof-reading our paper and providing very valuable comments. We also thank all anonymous referees for the careful review of this paper and the many suggestions for improvements they provided.

References

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

Ashri, R. and Luck, M. 2000. Paradigma: Agent implementation through Jini. In *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, pp. 453-457.

Bellifemine, F., Poggi, A., and Rimassa, G. 1999. JADE - A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Application of Intelligent Agent and Multi Agent Technology* (PAAM99), London, U.K., pp. 97-108.

Brazier, F.M.T., Dunin Keplicz, N., Jennings, N., and Treur, J. 1997. DESIRE: Modelling multi-agent systems in a compositional formal framework. In: *International Journal of Cooperative Information Systems* (M. Huhns, M. Singh Ed.), Volume 6, pp. 67-94, *Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems.*

Buchs, D. and Guelfi, N. 2000. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering* (TSE), Vol. 26, No. 7, pp. 635-652.

Crnogorac, L., Rao, A. S., and Ramamohanarao, K. 1997. Analysis of inheritance mechanisms in agent-oriented programming. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (IJCAI), pp. 647-654.

Deng, Y. and Chang, S. K. 1990. A G-net model for knowledge representation and reasoning. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3, pp. 295-310.

Deng, Y., Chang, S. K., Perkusich, A., and de Figueredo, J. 1993. Integrating software engineering methods and Petri nets for the specification and analysis of complex information systems. In *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, Chicago, pp. 206-223.

D'Inverno, M., Fisher, M., Lomuscio, A., Luck, M., De Rijke, M., Ryan, M., and Wooldridge, M. 1997. Formalisms for multi-agent systems. *The Knowledge Engineering Review*, Vol. 12, No. 3.

D'Inverno, M. and Luck, M. 2001. Formal agent development: framework to system. In *Formal Approaches to Agent-Based Systems: First International Workshop*, FAABS 2000, Rash, J.L., Rouff, C.A., Truszkowski, W., Gordon, D., Hinchey, M.G., (eds.), Lecture Notes in Artificial Intelligence, 1871, Berlin, Springer-Verlag, 2001, pp. 133-147.

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

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

FIPA 2002. The foundation for intelligent physical agents. *FIPA 2000 Specifications*. Http://www.fipa.org.

Fisher, M. 1995. 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, pp. 307-323.

Flores, R. A. and Kremer, R. C. 2001. 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.

Howden, N., Rönnquist, R., Hodgson, A., and Lucas, A. 2001. JACK intelligent agents – summary of an agent infrastructure. In *Proceedings of the 5th International Conference on Autonomous Agents*.

Huber, M. 1999. JAM: a BDI-theoretic mobile agent architecture. In *Proceedings of International Conference on Autonomous Agents*, pp. 236-243.

Huber, M. J., Kumar, S., Cohen, P. R., and McGee, D. R. 2001. A formal semantics for proxy communicative acts. In *Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages* (ATAL-2001), Seattle, Washington, USA.

Iglesias, C. A., Garrijo, M., and Centeno-González, J. 1998. A survey of agent-oriented methodologies. In *Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Language* (ATAL-98), pp. 317-330.

Ioerger, T. R., Volz, R. A., and Yen, J. 2000. Modeling cooperative, reactive behaviors on the battlefield using intelligent agents. In *Proceedings of the Ninth Conference on Computer Generated Forces* (9th CGF), pp. 13-23.

Kinny, D., Georgeff, M., and Rao, A. 1996. 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, pp. 56-71.

Lakos, C. A. and Keen, C. 1994. LOOPN++: A new language for object-oriented Petri nets. In *Proceedings of Modelling and Simulation* (European Simulation Multi-Conference), Barcelona, Society for Computer Simulation, pp. 369-374.

Luck, M. and d'Inverno, M. 1995. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, AAAI Press / MIT Press, pp. 254-260.

Murata, T. 1989. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, Vol. 77, No. 44, pp. 541-580.

Nwana, H., Ndumu, D., Lee, L., and Collins, J. 1999. ZEUS: A toolkit for building distributed multi-agent systems. *Applied Artificial Intelligence Journal*, Vol. 13, No. 1, pp. 129-186.

Odell, J., Van Dyke Parunak, H., and Bauer, B. 2001. Representing agent interaction protocols in UML. *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge, eds., Springer-Verlag, Berlin, pp. 121–140.

Pan, Y. 2002. *Refinement of an Agent-Based Model to support Decision Making and Standard Agent Communication Languages*, Masters Thesis, University of Illinois at Chicago.

Penczek, W. and Lomuscio, A. 2003. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems* (AAMAS'03), Melbourne.

Perkusich, A. and de Figueiredo, J. 1997. 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, pp. 39-59.

Poslad, S., Buckle, P., and Hadingham, R. 2000. The FIPA-OS agent platform: Open source for open standards. In *Proceedings of the 5th International Conference on the Practical Application of Intelligent Agent and Multi Agent Technology* (PAAM2000), Manchester, UK.

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

Rao, A. S. and Georgeff, M. P. 1993. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, Chambery, France, pp. 318-324.

Rodriguez-Aguilar, J. A., Martin, F. K., Garcia, P., Noriega, P., and Sierra, C. 1999. 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, pp. 284-300.

Saldhana, J., Shatz, S. M., and Hu, Z. 2001. Formalization of object behavior and interactions from UML models. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, Vol. 11, No. 6, pp. 643-673.

Siegel, J. and the OMG Staff Strategy Group 2001. Developing in OMG's model driven architecture (MDA). *OMG White Paper*, Object Management Group.

Smith, R. G. 1980. The Contract Net protocol: high-level communication and control in a distributed problem solver. *IEEE Transactions on Computer*, Vol. C-29, pp. 1104-1113.

Vasconcelos, W., Sabater, J., Sierra, C., and Querol, J. 2002. Skeleton-based agent development for electronic institutions. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Italy.

Vidal, J. M., Buhler, P. A., and Huhns, M. N. 2001. Inside an agent. *IEEE Internet Computing*, Vol. 5, No. 1, pp. 82-86.

Wooldridge, M., Jennings, N. R., and Kinny, D. 2000. The Gaia methodology for agent-oriented Analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, Vol. 3, No. 3, pp. 285-312.

Wooldridge, M. and Ciancarini, P. 2001. Agent-oriented software engineering: the state of the art. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, Springer-Verlag Lecture Notes in AI, Volume 1957.

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

Xu, D., Volz, R. A., Ioerger, T. R., and Yen, J. 2002. Modeling and verifying multi-agent behaviors using Predicate/Transition nets. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (SEKE'02), Italy, pp. 193-200.

Xu, D., Yin, J., Deng, Y., and Ding, J. 2003. A formal architectural model for logical agent mobility. *IEEE Transactions on Software Engineering*, Vol. 29, No.1, pp. 31-45.

Xu, H. and Shatz, S. M. 2000. Extending G-nets to support inheritance modeling in concurrent object-oriented design. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics* (SMC 2000), Nashville, Tennessee, USA, pp. 3128-3133.

Xu, H. and Shatz, S. M. (2001a). An agent-based Petri net model with application to seller/buyer design in electronic commerce. In *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems* (ISADS 2001), Dallas, Texas, USA, pp. 11-18.

Xu, H. and Shatz, S. M. (2001b). A framework for modeling agent-oriented software. In *Proceedings of the 21st International Conference on Distributed Computing Systems* (ICDCS-21), Phoenix, Arizona, USA, pp. 57-64.

Xu, H. and Shatz, S. M. 2003. A framework for model-based design of agent-oriented software. *IEEE Transactions on Software Engineering*, Vol. 29, No. 1, pp. 15-30.

Yen, J., Yin, J., Ioerger, T. R., Miller, M., Xu, D., and Volz, R. A. 2001. CAST: Collaborative agents for simulating teamwork. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (IJCAI-01), Seattle, WA, pp. 1135-1142.

Zhu, H. 2001. SLABS: A formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 5, pp. 529-558.