A Design Model for Intelligent Mobile Agent Software Systems[1]

Haiping Xu and Sol M. Shatz
Department of Computer Science
The University of Illinois at Chicago
Email: {hxu1, shatz}@cs.uic.edu

Abstract

One of the grand challenges to achieving wide spread use and rapid development of multi-agent systems is to adapt principles of software engineering. Agent-oriented software provides a new software engineering paradigm and the opportunities for development of new domain-specific software models. With the continuing improvement of agent technology, and the rapid growth of software system complexity, especially for Internet applications, there is a pressing need for general models of mobile agents – models that explicitly support the features of mobility, cooperative behavior, and intelligence. We present a design model for intelligent mobile agent software by introducing mobility into a framework for agent-oriented software. The model facilitates design reuse by providing an inheritance mechanism and explicitly supports asynchronous message passing. The approach benefits from a formal foundation that is based on the agent-oriented G-net formalism, a formalism derived from an object-based Petri net model. Thus the approach supports model analysis and property verification.

Keywords: Intelligent mobile agent (IMA), Petri net, agent-oriented G-net, design model, inheritance

1. Introduction

Software agents can be classified in terms of a space defined by the three dimensions of intelligence, agency and mobility [1]. The first dimension, *intelligence*, is rooted in artificial intelligence research and dates back to the fifties, where intelligent agents can be classified according to their capabilities to express preferences, beliefs and emotions, and according to their ability to fulfill a task by reasoning, planning and learning techniques. The second dimension, *agency*, is the degree of autonomy and authority vested in the agent, and can be measured by the nature of the interaction between an agent and other entities of the system. Particularly, an agent must run asynchronously. The third dimension of software agent research, *mobility*, has emerged in the nineties and is motivated by the rise and rapid growth of a networked

[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.

computing environment, and the need for techniques to locally exploit distributed resources. Within this dimension of software agent research, the goal is remote action and mobility of data and computation.

Current agent systems generally do not exploit all the capabilities classified by these three dimensions. For example, multi-agent systems (MAS) of distributed artificial intelligence try to execute a given task using a large number of possibly distributed but static agents that collaborate and cooperate in an intelligent manner [2][3]. On the other hand, research on mobile agents usually emphasizes agent mobility and agent coordination, and mobile agents are typically assumed to only have very limited or even no intelligence [4][5][6]. The development schema in the later case is sometimes called weak agent approach, which contrasts with the strong agent approach that involves artifical intelligence techniques [25].

Previous work on multi-agent systems has fostered the concept of agent-oriented software [7][8][16], where agents are viewed as intelligent software that has the properties of autonomy, reactivity, pro-activeness and social ability. Corresponding agent-oriented design methodologies are also proposed to provide guidelines for agent specification and design. Examples of such work are the AAII methodologies [9] and the Gaia methodologies [8], which are extensions of object-oriented methodologies. In our own previous work [16], an inheritance mechanism, in terms of agent functionalities, is introduced into agent-oriented software design.

For mobile agents, the concern is with intelligent software agents that can migrate over computer networks. The concept of location has been one of the key features to characterize mobility in most theoretical models of mobile agents, such as the distributed join-calculus [10], which is an extension of the π-calculus that introduces the explicit notions of named localities and distribution failure. Additional typical formalisms for agent mobility modeling are summarized as follows. Mobile UNITY [4] provides a programming notation that captures the notion of mobility and transient interactions among mobile nodes. Inspired by Mobile UNITY, the concept of connectors [11] is explicitly identified to describe different kinds of transient interactions, and facilitate the separation of coordination from computation in mobile computing. The connectors are written in COMMUNITY, a UNITY-like program design language whose semantics is given in a categorical framework. MobiS [5], as an extended version of PoliS, is a specification language based on multiple tuple spaces. It can be used to specify agent coordination and architectures containing mobile components. More recently, LIME [12], also based on tuple spaces, has been proposed as a middleware that supports the development of applications that exhibit both physical and logical mobility.

Although the above results formally model mobile agents in terms of their mobility, they are not built upon a framework that explicitly supports the intelligence feature of agents. Furthermore, they are weak in agent communication modeling, and typically such models are reactive rather than pro-active. In other words, these models may simply act in response to their environment, but they are not able to exhibit goal-directed

behaviors. Additional efforts, such as the MARS project [6], attempt to introduce context-dependent coordination into agent models, however, without explicitly suggesting the communication mechanism among mobile agents. There are also some research efforts concerned with mobile agent communication mechanisms, however they are not formally defined [13][14].

From the above review, we can see that current work on mobile agents mostly emphasizes some particular features of the mobile agents, e.g., agent mobility. With the continuing improvement of agent technology, and the rapid growth of software system complexity, especially for Internet applications, there is a pressing need for a more general model of mobile agents, in which agents are not only mobile and cooperative, but also intelligent. There are a few previous efforts that discuss intelligent mobile agents [18][14], however they lack a formal framework for intelligent mobile agent design. In this paper, we propose an intelligent mobile agent (IMA) model by introducing mobility into a framework for agent-oriented software. This framework has been designed to model intelligent software agents [15][16] for multi-agent systems, and it supports design reuse by providing an inheritance mechanism. Meanwhile, the resulting mobile agent models explicitly support asynchronous message passing. Another advantage of our approach is that our fundamental agent model is based on the agent-oriented G-net formalism [16], a formalism derived from an object-based Petri net model. As a formal model, this agent-oriented formalism can be translated into more "standard" forms of a Petri net for design analysis, including model checking. Examples of such analysis can be found in earlier work [17].

The rest of this paper is organized as follows. Section 2 describes the agent-oriented G-net model, which was first proposed in [16]. Section 3 proposes the architecture for a mobile agent system, and illustrates how to design the principle agent system components: the intelligent mobile agents (IMA) and the intelligent facilitator agents (IFA). Section 4 uses an electronic marketplace example to show how to incrementally design application-specific intelligent mobile agents using the discussed architecture. Finally, in Section 5, we summarize our contributions and discuss the 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 [20][21]. 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

[19]. 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 detailed 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 [23], 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.

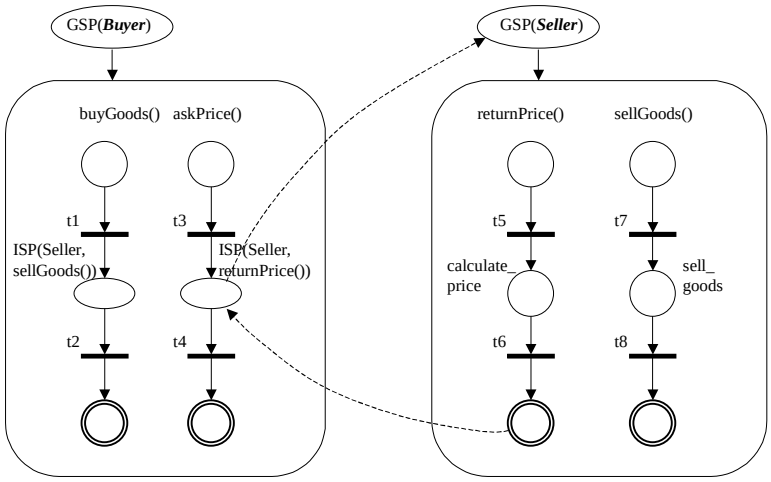


Figure 1. G-net models of buyer and seller objects

The token movement in a G-net object is similar to that of original G-nets [20][21]. 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.

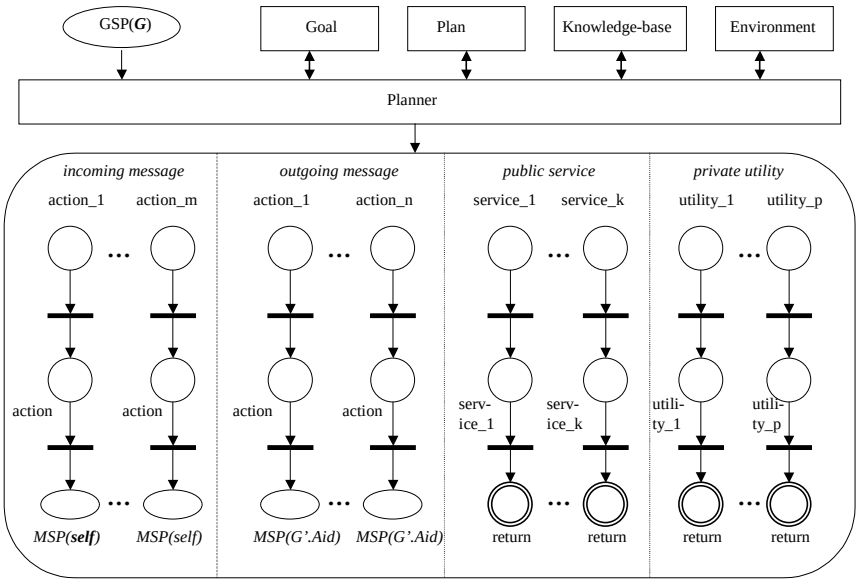
Notice that the example we provide in Figure 1 follows the *Client-Server* paradigm, in which a *Seller* object works as a server and a *Buyer* object is a client. Further details about G-net models can be found in references [20][21].

2.2 An Architecture for Agent-Oriented Design

Although the G-net model works well in object-based design, it is not sufficient in agent-based design for the following reasons. First, agents that form a multi-agent system may be developed independently by different vendors, 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]. 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* 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 [9], 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 plans are achieved, and the *Goal*, *Plan* and *Knowledge-base* modules of an agent are updated after the execution of each communicative act [15][16] 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 agents. 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.

[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.



Notes: G'.Aid = mTkn.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 *private utility*. The *incoming/outgoing message* section defines a set of *message processing units (MPU)*, which correspond to a subset of communicative acts [15][16]. 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 processing. The *public service* section defines a set of methods that provide services to other agents, and it makes an agent work as a server. Similarly, the *private utility* 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 [8]. In particular, these messages must satisfy standardized communicative (speech) acts, which define the type and the content of the message (e.g., the FIPA agent communication language, or KQML) [22]. Note that in Figure 2 each named *MPU action_i* refers to a communicative act, thus our agent-oriented model supports an agent communication interface. In addition, agents analyze these messages and can decide whether to execute the requested action. As we 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. 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 does not wait for the token to return before it can continue to execute its next step.

A template of the *Planner* module is shown in Figure 3 The modules *Goal*, *Plan*, *Knowledge-base* and *Environment* are represented as four special places (denoted by double ellipses in Figure 3), each of which contains a token that represents a set of goals, a set of plans, a set of beliefs and a model of the environment, respectively. These four modules connect with the *Planner* module through abstract transitions, denoted by shaded rectangles in Figure 3 (e.g., the abstract transition *make_decision*). Abstract transitions represent abstract units of decision-making or mental-state-updating. At a more detailed level of design, abstract transitions would be refined into sub-nets; however how to make decisions and how to update an agent's mental state is beyond the scope of this paper, and will be considered in our future work. In the *Planner* module, there is a unit called *autonomous unit* that makes an agent autonomous and internally motivated. An *autonomous unit* contains a sensor (represented as an abstract transition), which may fire whenever the pre-conditions of some committed plan are satisfied or when new events are captured from the environment. If the abstract transition *sensor* fires, the autonomous unit will then decide based on an agent's current mental state (goal, plan and knowledge-base) whether to start a conversation or to simply update its mental state. This is done by firing either the transition *start_a_conversation* or the transition *automatic_update* after executing any necessary actions associated with place *new_action*.

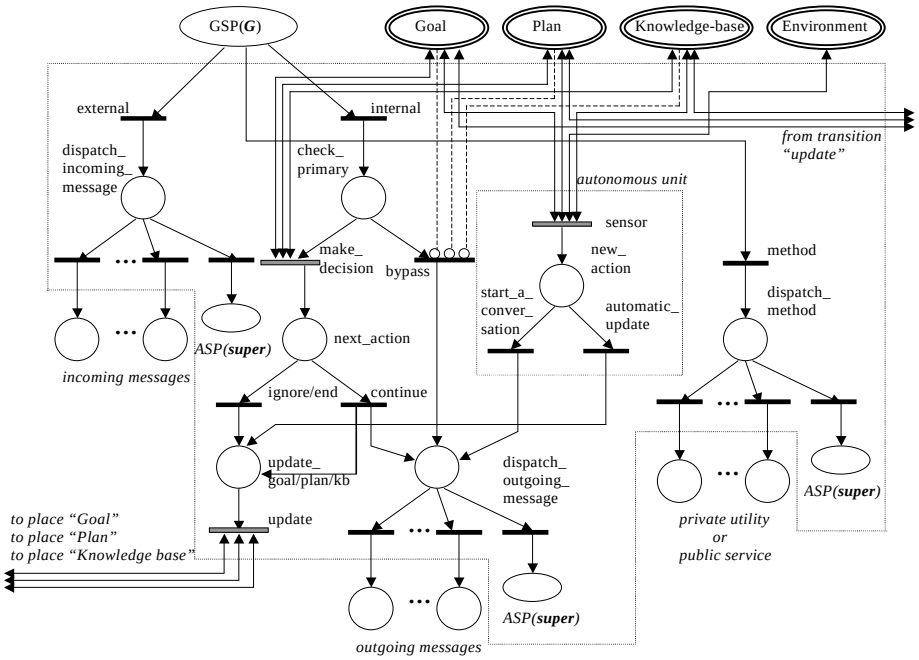


Figure 3. A template for the *Planner* module

Note that the *Planner* module is both goal-driven and event-driven because the transition *sensor* may fire when any committed plan is ready to be achieved or any new event happens. In addition, the *Planner* module is also message-triggered because certain actions may initiate whenever a message arrives (either from some other agent or from the agent itself). A message is represented as a message token, denoted as *mTkn*, with a tag of **internal**/**external**/**method**. A message token with a tag of **external** represents an incoming message which comes from some other agent, or a newly generated outgoing message before sending to some other agent; while a message token with a tag of **internal** is a message forwarded by an agent to itself with the *MSP* mechanism. In either case, the message token with the tag of **internal**/**external** should not be involved in an invocation of a method call. On the contrary, a message token with a tag of **method** indicates that the token is currently involved in an invocation of some method call. When an incoming message/method arrives, with a tag of **external**/**method** in its corresponding token, it will be dispatched to the appropriate *MPU/method* defined in the internal structure of the agent. If it is a method invocation, the method defined in the *public service* or *private utility* section of the internal structure will be executed, and after the execution, the token will return to the calling unit, i.e., an *ISP* of the calling agent. However, if it is an incoming message, the message will be first processed by a *MPU* defined in the *incoming message* section in the internal structure of the agent. Then the tag of the token will be changed from **external** to **internal** before it is transferred back to the *GSP* of the receiver agent by using *MSP(**self**)*. Note that we have extended G-nets to allow the use of the keyword **self** to refer to the agent object itself. Upon the arrival of a token tagged as **internal** in a *GSP*, the transition *internal* may fire, followed by the firing of the abstract transition *make_decision*. Note that at this point of time, there would exist tokens in those special places *Goal*, *Plan* and *Knowledge-base*, so the transition *bypass* is disabled (due to the "inhibitor arc"[3]) and may not fire (the purpose of the transition *bypass* is for inheritance modeling, which will be addressed in Section 2.3). Any necessary actions may be executed in place *next_action* before the conversation is either ignored/ended or continued. If the current conversation is ignored, the transition *ignore/end* fires; otherwise, the transition *continue* fires. If the transition *continue* fires, a newly constructed outgoing message, in the form of a token with a tag of **internal**, will be dispatched into the appropriate *MPU* in the *outgoing message* section of the internal structure of the agent. After the message is processed by the *MPU*, the message will be sent to a receiver agent by using the *MSP(G'.Aid)* mechanism, and the tag of the message token will be changed from **internal** to **external**, accordingly. In either case, a token will be deposited into place *update_goal/plan/kb*, allowing the abstract transition *update* to fire. As a consequence, the *Goal*, *Plan* and *Knowledge-base* modules are updated if needed, and the agent's mental state may change.

[3] An inhibitor arc connects a place to a transition and defines the property that the transition associated with the inhibitor arc is enabled only when there are no tokens in the input place.

2.3 Inheritance Modeling

To support agent-oriented design, we also need to incorporate some inheritance modeling capabilities. But inheritance in agent-oriented design is more complicated than in object-oriented design. Unlike an object (passive object), an agent object has mental states and reasoning mechanisms. Therefore, inheritance in agent-oriented design invokes two issues: an agent subclass may inherit an agent superclass's knowledge, goals, plans, the model of its environment and its reasoning mechanisms; on the other hand, as in the case of object-oriented design, an agent subclass may inherit all the services that an agent superclass may provide, such as public services and private utility functions [16]. Since inheritance happens at the class level, an agent subclass may be initialized with an agent superclass's initial mental state, but new knowledge acquired, new plans made, and new goals generated in an individual agent object (as an instance of an agent superclass), cannot be inherited by an agent object when creating an instance of an agent subclass. For simplicity, we assume that an instance of an agent subclass (i.e., an subclass agent) always uses its own reasoning mechanisms, and thus the reasoning mechanisms in the agent superclass should be disabled in some way. Therefore, as proposed in earlier work [16] we only consider how to initialize a subclass agent's mental state while an agent subclass is instantiated; meanwhile, we concentrate on the inheritance of services that are provided by an agent superclass, i.e., the *MPUs* and *methods* defined in the internal structure of an agent class. Before presenting our inheritance scheme, we need the following definition:

Definition 2.1 *Subagent and Primary Subagent*
When an agent subclass *A* is instantiated as an agent object *AO*, a unique agent identifier is generated, and all superclasses and ancestor classes of the agent subclass *A*, in addition to the agent subclass *A* itself, are initialized. Each of those initialized classes then becomes a part of the resulting agent object *AO*. We call an initialized superclass or ancestor class of agent subclass *A*, a *subagent*, and the initialized agent subclass *A* the *primary subagent*.

The result of initializing an agent class is to take the agent class as a template and create a concrete structure of the agent class and initialize its state variables. Since we represent an agent class as an agent-oriented G-net, an initialized agent class is modeled by an agent-oriented G-net with initialized state variables. In particular, the four tokens in the special places of an agent-oriented G-net, i.e., *gTkn*, *pTkn*, *kTkn* and *eTkn*, are set to their initial states. Since different subagents of *AO* may have goals, plans, knowledge and environment models that conflict with those of the primary subagent of *AO*, it is desirable to resolve them in an early stage. In our case, we deal with those conflicts in the instantiation stage in the following way. All the tokens *gTkn*, *pTkn*, *kTkn* and *eTkn* in each subagent of *AO* are removed from their associated special places, and the tokens are combined with the *gTkn*, *pTkn*, *kTkn* and *eTkn* in the primary

subagent of *AO*.[4] The resulting tokens *gTkn*, *pTkn*, *kTkn* and *eTkn* (newly generated by unifying those tokens for each type), are put back into the special places of the primary subagent of *AO*. Consequently, all subagents of *AO* lose their abilities for reasoning, and only the primary subagent of *AO* can make necessary decisions for the whole agent object. More specifically, in the *Planner* module (as shown in Figure 3 that belongs to a subagent, the abstract transitions *make_decision*, *sensor* and *update* can never be enabled because there are no tokens in the following special places: *Goal*, *Plan* and *Knowledge-base*. If a message tagged as **internal** arrives, the transition *bypass* may fire and a message token can directly go to a *MPU* defined in the internal structure of the subagent if it is defined there. This is made possible by connecting the transition *bypass* with inhibitor arcs (denoted by dashed lines terminated with a small circle in Figure 3) from the special places *Goal*, *Plan* and *Knowledge-base*. So the transition *bypass* can only be enabled when there are no tokens in these places. In contrast to this behavior, in the *Planner* module of a primary subagent, tokens do exist in the special places *Goal*, *Plan* and *Knowledge-base*. Thus, the transition *bypass* will never be enabled. Instead, the transition *make_decision* must fire before an outgoing message is dispatched.

To reuse the services (i.e., *MPUs* and *methods*) defined in a subagent, we need to introduce a new mechanism called *Asynchronous Superclass switch Place (ASP)*. An *ASP* (denoted by an ellipsis in Figure 3) is similar to a *MSP*, but with the difference that an *ASP* is used to forward a message or a method call to a subagent rather than to send a message to an agent object. For the *MSP* mechanism, the receiver could be some other agent object or the agent object itself. In the case of *MSP(**self**)*, a message token is always sent to the *GSP* of the primary subagent. However, for *ASP(**super**)*, a message token is forwarded to the *GSP* of a subagent that is referred to by ***super***. In the case of single inheritance, ***super*** refers to a unique superclass G-net, however with multiple inheritance, the reference of ***super*** must be resolved by searching the class hierarchy diagram.

When a message/method is not defined in an agent subclass model, the dispatching mechanism will deposit the message token into a corresponding *ASP(**super**)*. Consequently, the message token will be forwarded to the *GSP* of a subagent, and it will be again dispatched. This process can be repeated until the root subagent is reached. In this case, if the message is still not defined at the root, an exception occurs. In this paper, we do not provide exception handling for our agent-oriented G-net models, and we assume that all incoming messages have been correctly defined in the primary subagent or some other subagents.

[4] The process of generating the new token values would involve actions such as conflict resolution among goals, plans or knowledge-bases, which is a topic outside the scope of our model and this paper.

3. Intelligent Mobile Agent Design

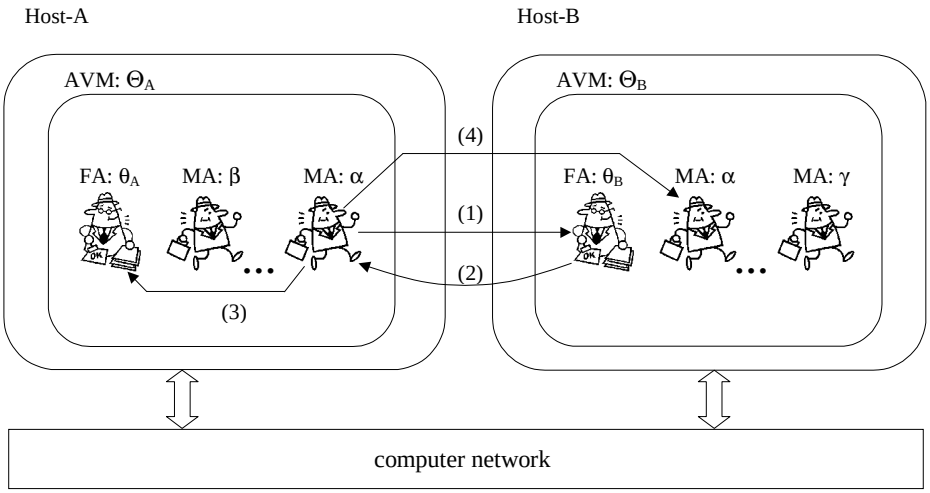
Today's users demand ubiquitous network access independent of their physical location. This style of computation, often referred to as *mobile computing*, is enabled by rapid advances in wireless communication technology [12]. The networking scenarios enabled by mobile computing range roughly between two extremes. At one end, the availability of a fixed network is assumed, and its facilities are exploited by the mobile infrastructure. We call this form of mobility *logical mobility*. At the other end, the fixed network is absent and all network facilities (e.g., routing) must be implemented by relying only on the available mobile hosts, namely *ad hoc* networks. This form of mobility is called *physical mobility*. Mobile agent technology is a new networking technology that deals with both forms of mobility. It offers a new computing paradigm in which a program, in the form of an intelligent software agent, can suspend its execution on a host computer, transfer itself to another agent-enabled host on the network, and resume execution on the new host. Here, as we will see in the next section, we define a host as either a static host or a mobile host, which is situated in an *ad hoc* network.

3.1 Agent World Architecture

First, we introduce the concepts of agent virtual machine (AVM) and agent world (AW), which serve to define a framework for a mobile agent system. Figure 4 shows a generic mobile agent system, and an example of agent migration. In the figure, Host-A and Host-B are two machines connected by a network. To make mobile agents platform independent, a mobile agent runs on an agent virtual machines (AVM), which provides a protected agent execution environment. Each host may have a number of AVMs, however, to make it simple, we only illustrate one AVM for each host in Figure 4. Each AVM is responsible for hosting and executing any agents created on that AVM or that arrive over the network, and for providing API for agent programmers.

We now provide a few key definitions for the mobile agent system.

Definition 3.1 *Agent World (AW)*
An *agent world (AW)* is a 3-tuple *(WKHOST, SHOST, HCOM)*, where *WKHOST* is a well-known static host, which is responsible for recording the most recent IP address of all other hosts. *SHOST* is a set of hosts that can provide agent virtual machines, where members of this set could be either *static* or *mobile*. Note that, in a special case, *WKHOST* is a member of *SHOST*. *HCOM* is the communication protocol among hosts in *SHOST*, an example of such protocols is TCP/IP.



(1) move-request (2) grant (3) notify (4) move

Figure 4. Agent world architecture and an example of agent migration

Definition 3.2 *Static Host (SH) and Mobile Host (MH)*

A host is 4-tuple *(SAVM, ACOM, HOMEIP, CURIP)*, where *SAVM* is a set of *agent virtual machines (AVM)*. *ACOM* is the communication protocol among AVMs in *SAVM*, and examples of such protocols are IPC and TCP/IP. *HOMEIP* is the original IP address of the host, and *CURIP* is the current IP address of the host. If at any time, *CURIP = HOMEIP*, we call the host a *static host (SH)*; otherwise, we call it a *mobile host (MH)*.

Definition 3.3 *Agent Virtual Machine (AVM)*

An *agent virtual machine (AVM)* is a 5-tuple *(FA, SMA, MCOM, HOSTIP, ID)*, where *FA* is a facilitator agent for *AVM*, which is responsible for recording information of mobile agents running on that *AVM*, and also for providing services for mobile agents running on other *AVM*s. Note that *FA* is a static agent, i.e., it does not migrate. *SMA* is a set of mobile agents. *MCOM* is the communication protocols for both static and mobile agents. *HOSTIP* is the current IP address of the host where the *AVM* runs on, and *ID* is a unique identifier for that *AVM*.

Definition 3.4 *Static Agent (SA) and Mobile Agent (MA)*

An *agent A* is 3-tuple *(HOMEIP, CURIP, AO)*, where *HOMEIP* is the IP address of the host, on which agent *A* is created. *CURIP* is the IP address of the host where agent *A* currently runs on. *AO* is the agent object with the general structure as we described in Section 2. If at ant time, *CURIP = HOMEIP*, we refer to agent *A* as a *static agent (SA)*; otherwise, we refer to agent *A* as a *mobile agent (MA)*.

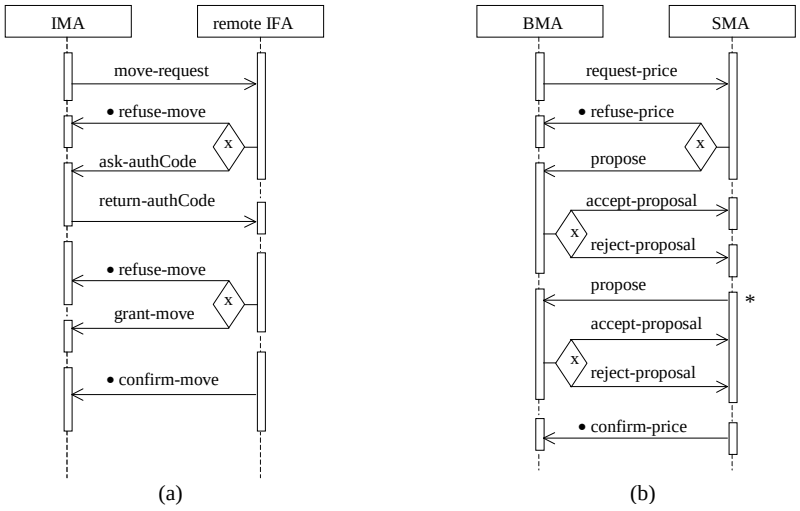
Since in this paper we view mobile agents and facilitator agents (an example of static agent) as intelligent software agents, for the rest of this paper a mobile/facilitator agent always refers to an intelligent mobile agent (IMA) or an intelligent facilitator agent (IFA). As shown in Figure 4, when a mobile agent α on AVM Θ_A wants to migrate to another AVM Θ_B, it needs to contact with the remote facilitator agent θ_B first, which resides on AVM Θ_B (step 1). In fact, the mobile agent α needs to know the address of the remote facilitator agent θ_B before the communication can begin. This could be done by querying this information from its local facilitator agent θ_A, which resides on AVM Θ_A. If the local facilitator agent θ_A knows the address of the remote facilitator agent θ_B, it will provide this information to the mobile agent α; otherwise, it will contact with the well-known static host Π (we do not show it in Figure 4) for this information and forward the results to the mobile agent α thereafter. For simplicity, this procedure is omitted in Figure 4. Based on security and resource criteria, the remote facilitator agent θ_B decides if the migration request is granted. If the migration request is granted (step 2), the mobile agent α notifies its local facilitator agent θ_A about its leaving (step 3), and it finally moves to the remote AVM Θ_B (step 4). In the following section, we will see that, in our approach, step 1 and step 2 are modeled by asynchronous message passing; while step 3 and step 4 are modeled by method invocation.

The situation above is an example of logical mobility. For physical mobility, a host may at some time change its IP address or lose its IP address temporarily (detached from the network) at some time. In this case, the well-known static host Π is critical for recording this information. To successfully send a message to an agent on which the AVM has changed its *HOSTIP* address, the knowledge of the sender agent's local facilitator agent needs to be consistent with the latest network information. Further discussion about this issue is beyond the scope of this paper, which concentrates on logical mobility.

3.2 Intelligent Mobile Agent (IMA) and Intelligent Facilitator Agent (IFA)

To illustrate the processes for design of intelligent mobile agents (IMA) and intelligent facilitator agents (IFA) by using our agent model, we use the following examples. Since we view a facilitator agent as an IFA, in addition to provide public services to a mobile agent or some other IFA, an IFA also has the capability of making decisions. This feature is vitally important for an IFA to cater for the needs of service allocation in a dynamic network environment, such as resource management and security verifications. Figure 5 (a) depicts a template of a contract net protocol [24] expressed as an agent UML (AUML) sequence diagram [22] for a migration-request protocol between a *mobile agent (MA)* and a remote *facilitator agent (FA)*. Figure 5 (b) is a modified example of a contract net protocol adapted from [22], which depicts a template of a protocol expressed as an AUML sequence diagram for a price-negotiation protocol between a *buying mobile agent (BMA)* and a *selling mobile agent (SMA)*. Some of the notations of AUML are adapted from [22] as extensions of UML sequence diagrams for agent design. In addition, to

correctly draw the sequence diagram for the protocol templates, we introduce two new notations, i.e., the end of protocol operation "•" and the iteration of communication operation "∗".



IMA: intelligent mobile agent, IFA: intelligent facilitator agent, BMA: buying mobile agent, SMA: selling mobile agent

Figure 5. Contract net protocols (a) a temple for the migration-request protocol (b) a template for the price-negotiation protocol

Consider Figure 5 (a). When a conversation based on a contract net protocol begins, the *intelligent mobile agent (IMA)* sends a request for migration to a remote *intelligent facilitator agent (IFA)* on a different AVM. The remote *IFA* can then choose to respond to the *IMA* by refusing its migration or asking the *IMA*'s authorization code, which is used to verify that the *IMA* is on a trustable AVM. Here the "x" in the decision diamond indicates an exclusive-or decision. If the remote *IFA* refuses the migration based on resource limitation or some other reasons, the protocol ends; otherwise, the remote *IFA* waits for the *IMA*'s authorization code to be supplied. If the *IMA*'s authorization code is correctly provided, the remote *IFA* may grant the *IMA* for migration if it is trustable, or refuse the migration otherwise. Again, if the remote *IFA* refuses *IMA*'s migration, the protocol ends; otherwise, a confirmation message will be provided afterwards. Similarly, the price-negotiation protocol between a *buying mobile agent (BMA)* and a *selling mobile agent (SMA)*, which are subclasses of *IMA*, can be illustrated in Figure 5 (b).

Based on the communicative acts (e.g., *move-request*, *refuse-move*, etc.) needed for the contract net protocol in Figure 5 (a), we may adopt the agent design template shown in Figure 2, and design the mobile agent class as in Figure 6. The *Goal*, *Plan*, *Knowledge-base* and *Environment* modules remain as abstract units and can be refined in a further detailed design stage. The *Planner* module may reuse the template shown in Figure 3. The design of the remote facilitator agent is similar, which is illustrated in Figure 7.

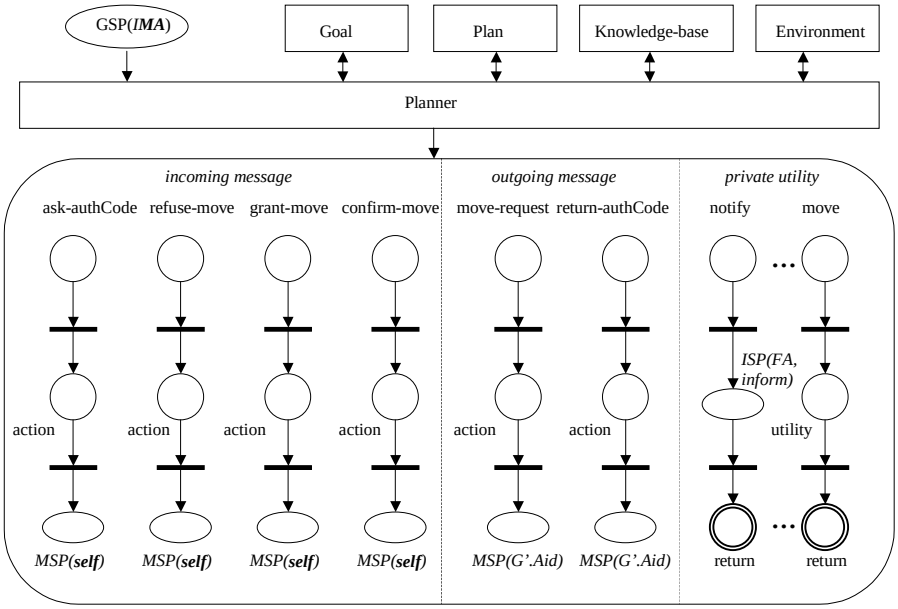


Figure 6. An agent-based G-net model for intelligent mobile agent class (IMA)

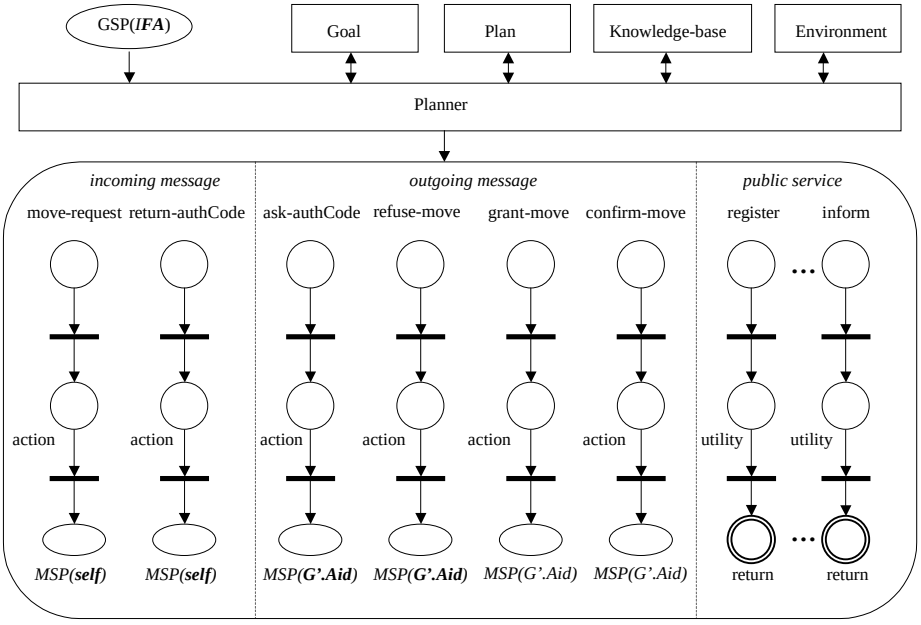


Figure 7. An agent-based G-net model for intelligent facilitator agent class (IFA)

To show how our agent models work correctly in an agent conversation, we now discuss an example. Consider a mobile agent object *MAO*, which receives a message of *ask-authCode* from a remote facilitator agent object *FAO*. A *mTkn* token with a tag of **external** will be deposited in the *GSP* of the primary subagent of *MAO*, i.e., the *GSP* of the corresponding intelligent mobile agent class (*IMA*). The transition *external* in *MA*'s *Planner* module may fire, and the *mTkn* will be moved to the place *dispatch_incoming_message*. Since there is an *MPU* for *ask-authCode* defined in the internal structure of *MA*, the *mTkn* will be dispatched to the entry place of that *MPU*. After the message is processed, *MSP(**self**)* changes the tag of the *mTkn* from **external** to **internal**, and sends the processed *mTkn* token back into the *GSP* of *IMA*. Upon the arrival of this message token, the transition *internal* in the *Planner* module of *MA* may fire, and the *mTkn* token will be moved to the place *check_primary*. Since *IMA* corresponds to the primary subagent of *MAO*, there are tokens in the special places *Goal*, *Plan*, *Knowledge-base* and *Environment*. Therefore the abstract transition *make_decision* may fire, and any necessary actions are executed in place *next_action*. Then the current conversation is either ignored or continued based on the decision made in the abstract transition *make_decision*. If the current conversation is ignored, the goals, plans and knowledge-base are updated as needed; otherwise, in addition to the updating of goals, plans and knowledge-base, a newly constructed *mTkn* with a tag of **internal** is deposited into place *dispatch_outgoing_message*. The new *mTkn* token has the message name *return-authCode*, following the protocol defined in Figure 5 (a). Again, there is an *MPU* for *return-authCode* defined in *IMA*, so the new *mTkn* token will be dispatched into the entry place of that *MPU*. After the message is processed, the *MSP(G'.Aid)* mechanism changes the tag of the *mTkn* token from **internal** to **external**, and transfers the *mTkn* token to the *GSP* of the receiver agent, in this case, the remote facilitator agent object *FAO*.

To further illustrate how to refine the MPU/method in a mobile agent's internal structure, we use the examples of the MPU *confirm-move* defined in the incoming message section and the method *move* defined in the private utility section. The refinement of another method *notify()* is straightforward; as shown in Figure 6, the *notify()* method makes a method invocation *inform()* to its local facilitator agent. This is done to notify the facilitator agent that the calling agent is leaving. The refinement of method *move()* and MPU *confirm-move* are shown in Figure 8 (a) and Figure 8 (b), respectively. In Figure 8 (a), when there is a token deposited in the entry place, the transition *start_move* fires, and deposit a token into place *migration*. The migration might be successful or failed, due to the network condition. If the migration fails, the transition *fail* fires, and deposits a token into place *retry*. The mobile agent will then count the number of retrials. If it has retried less than *MAX_TRIAL* times, the mobile agent will try to migrate again; otherwise, the transition *else* fires, and a method call *inform(FAILURE)* will be made to its local *facilitator agent (FA)* to notify the local *FA* that its migration is failed. This is modeled by the *ISP(rFA, inform(FAILURE))* mechanism. After that, the method call *move()* returns. If the migration succeeds, the transition *succeed* fires, and the mobile agent's current IP address *CURIP* will be changed to the new one. Then a method call

ISP(rFA, register) is made to the remote *facilitator agent (FA)*, which is actually the mobile agent's local *FA* now. After registering with the *FA*, the method call *move()* returns.

In Figure 8 (b), the refinement of MPU *confirm-move* is straightforward. When there is a token deposited into the entry place of the MPU *confirm-move*, the transition *begin_process* fires. After processing the message token, it makes a method call *ISP(**self**, notify)* to the agent itself, which further makes a method call to the mobile agent's local facilitator agent -- to inform the facilitator agent that the mobile agent is leaving. After that, the migration starts by invoking the method *move()*. Finally, after finishing the migration, either failed or succeeded, it transfers the message token to the agent itself, and ends the conversation.

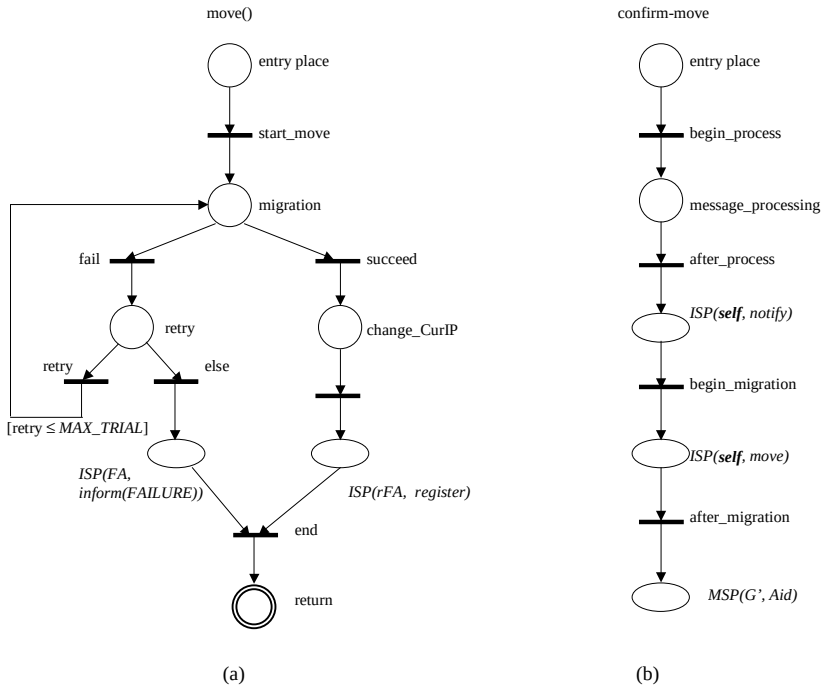


Figure 8. Examples of detailed design (a) refinement of method *move()*
(b) refinement of MPU *confirm-move*

4. Intelligent Mobile Agent Design in an Electronic Marketplace

Consider a mobile agent family in an electronic marketplace domain, which is a global stock market tracking and trading system. Figure 9 shows the agents in a UML class hierarchy notation. An *intelligent mobile agent class (IMA)* is defined as a superclass that is capable of communicating with an *intelligent facilitator agent class (IFA)*, and migrating among *AVM*s. The functionality of an *intelligent mobile agent class (IMA)* can be inherited by an agent subclass, such as a *buying mobile agent class (BMA)* or a *selling*

mobile agent class (SMA). Both the *BMA* and *SMA* may reuse the functionality of *IMA* for communication with *IFA* and migration among *AVM*s. Furthermore, a *broker mobile agent class* is designed as a subclass of both the *BMA* and *SMA*, and a *stock-buyer/stock-seller mobile agent class* may be defined as a subclass of a *BMA*/*SMA*.

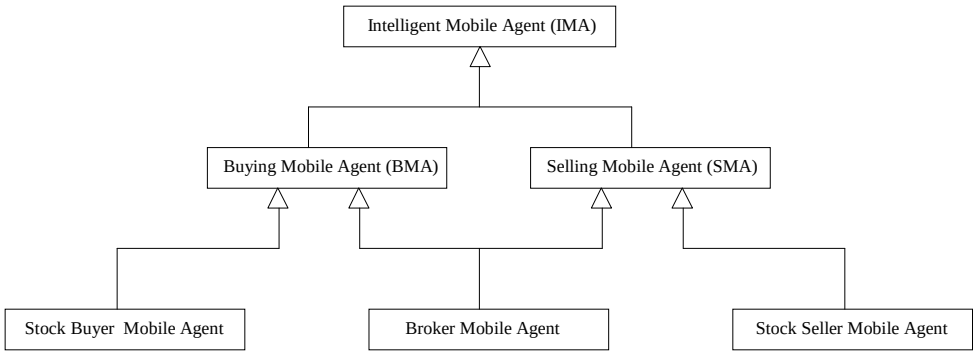


Figure 9. The class hierarchy diagram of mobile agents in an electronic marketplace

Based on the communicative acts (e.g., *request-price*, *refuse-price*, etc.) needed for the contract net protocol between the *buying mobile agent (BMA)* and the *selling mobile agent (SMA)*, we may design the *BMA* as shown in Figure 10. The *SMA* can be designed in the same way.

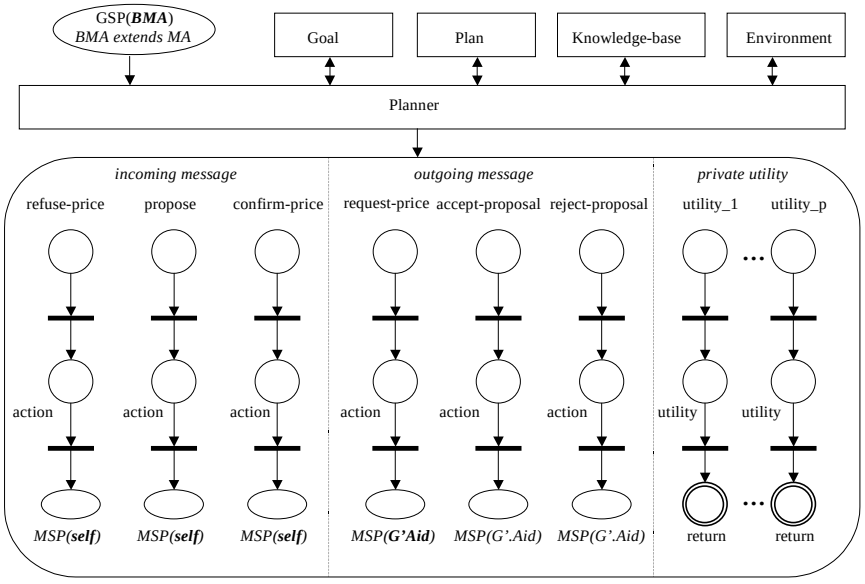


Figure 10. An agent-based G-net model for buying mobile agent class (BMA)

With inheritance, a *buying mobile agent class (BMA)*, as a subclass of a *mobile agent class (MA)*, may reuse *MPUs/methods* defined in *MA*'s internal structure. Similarly, a *selling mobile agent class (SMA)* inherits all *MPU/methods* of *MA*, and a *retailer mobile agent class* inherits all *MPU/methods* of both the *BMA* and the *SMA*.

Now we discuss an example to show how the reuse of *MPU/methods* works. Consider a buying mobile agent object *BMO*, which receives a message of *ask-authCode* from a remote facilitator agent object *FAO*. A *mTkn* token will be deposited in the *GSP* of the primary subagent of *BMO*, i.e., the *GSP* of the corresponding buying mobile agent class (*BMA*). The transition *external* in *BMA*'s *Planner* module may fire, and the *mTkn* will be moved to the place *dispatch_incoming_message*. Since there is no *MPU* for *ask-authCode* defined in the internal structure of *BMA*, the *mTkn* will be moved to the *ASP(**super**)* place. Since ***super*** here refers to a unique superclass – the mobile agent class (*MA*) – the *mTkn* will be transferred to the *GSP* of *MA*. Now the *mTkn* can be correctly dispatched to the *MPU* for *ask-authCode*. After the message is processed, *MSP(**self**)* changes the tag of the *mTkn* from **external** to **internal**, and sends the processed *mTkn* token back into the *GSP* of *BMA*. Note that *MSP(**self**)* always sends a *mTkn* back to the *GSP* of the primary subagent. Upon the arrival of this message token, the transition *internal* in the *Planner* module of *BMA* may fire, and the *mTkn* token will be moved to the place *check_primary*. Since *BMA* corresponds to the primary subagent of *BMO*, there are tokens in the special places *Goal*, *Plan*, *Knowledge-base* and *Environment*. Therefore the abstract transition *make_decision* may fire, and any necessary actions are executed in place *next_action*. Then the current conversation is either ignored or continued based on the decision made in the abstract transition *make_decision*. If the current conversation is ignored, the goals, plans and knowledge-base are updated as needed; otherwise, in addition to the updating of goals, plans and knowledge-base, a newly constructed *mTkn* with a tag of **internal** is deposited into place *dispatch_outgoing_message*. The new *mTkn* token has the message name *return-authCode*, following the protocol defined in Figure 5 (a). Again, there is no *MPU* for *return-authCode* defined in *BMA*, so the new *mTkn* token will be dispatched into the *GSP* of *MA*. Upon the arrival of the *mTkn* in the *GSP* of *MA*, the transition *internal* in the *Planner* module of *MA* may fire. However at this time, *MA* does not correspond to the primary subagent of *BMO*, so all the tokens in the special places of *Goal*, *Plan*, and *Knowledge-base* have been removed. Therefore, the transition *bypass* is enabled. When the transition *bypass* fires, the *mTkn* token will be directly deposited into the place *dispatch_outgoing_message*, and now the *mTkn* token can be correctly dispatched into the *MPU* for *return-authCode* defined in *MA*. After the message is processed, the *MSP(G'.Aid)* mechanism changes the tag of the *mTkn* token from **internal** to **external**, and transfers the *mTkn* token to the *GSP* of the receiver agent, in this case, the remote facilitator agent *FAO*.

For the reuse of public services and private utility functions defined in a superclass, the situation is the same as in the case of object-oriented design. In addition, there are three different forms of inheritance that are commonly used, namely augment inheritance, restrictive inheritance and refinement inheritance. The

usage of these three forms of inheritance in agent-oriented design is also similar to that in object-oriented design. Examples concerning reuse of public services and private utility functions and different forms of inheritance can be found in earlier work [23].

5. Conclusion and Future Work

Agent-oriented software provides a new software engineering paradigm and the opportunities for development of new domain-specific software models. With the continuing improvement of agent technology, and the rapid growth of software system complexity, especially for Internet applications, there is a pressing need for general models of mobile agents. Such models can allow a structured approach for design of agent software systems and facilitate the application of formal methods techniques for design analysis and implementation synthesis.

We presented the design models of intelligent mobile agents in a framework for agent-oriented software. Unlike previous work, which only models a particular feature of mobile agents, our mobile agent models can be served as a general agent model that has the capabilities of mobility, corporative behavior, and intelligence. With the example of electronic marketplace, we show that specific mobile agents can be design incrementally as subclasses of the mobile agent base class. Furthermore, our intelligent mobile agent models are based on the agent-oriented G-net formalism, which can be translated into a standard form of Petri net (Predicate-Transition net, Pr/T net) [19][20]. Because the Petri net formalism is theoretically mature and supported by robust tools, our approach supports formal analysis, such as model checking.

For our future research work, we plan to use our marketplace example to demonstrate the analysis power inherent in our intelligent mobile agent models. We will investigate use of model checking techniques to show that our agent models satisfy certain behavioral properties, such as effective movement and freedom of deadlock. We will also try to implement a mobile agent prototype following our formal design, by which we can show our approach supports rapid development of mobile agents. Finally, we need to explore various security issues in mobile agent design. As a side note, since we embed agent movement, and any other possible actions, in the context of agent conversations, we believe that our approach leaves adequate room for security modeling.

References

[1] K. Rothermel and M. Schwehm, “Mobile Agents,” In: A. Kent and J. G. Williams (Eds.): *Encyclopedia for Computer Science and Technology*, Volume 40 - Supplement 25, New York: M. Dekker Inc., 1999, pp. 155-176.

[2] D. Kinny, M. P. Georgeff, "Modeling and Design of Multi-Agent Systems," *Proceedings of the 4th Int'l Workshop on Agent Theories, Architectures, and Language (ATAL-97)*, 1997, pp. 1-20.

[3] N. R. Jennings, K. Sycara and M. Wooldridge, "A Roadmap of Agent Research and Development," *International Journal of Autonomous Agents and Multi-Agent Systems*, 1(1), 1998, pp. 7-38.

[4] G.-C. Roman, P. J. McCann, and J. Y. Plun, "Mobile UNITY: Reasoning and Specification in Mobile Computing," *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, July 1997, pp. 250-282.

[5] C. Mascolo, "MobiS: A Specification Language for Mobile Systems," In *Third Int. Conference on Coordination Models and Languages*, Amsterdam, The Netherlands. April 1999. P. Ciancarini and A. Wolf (editors). Lecture Notes in Computer Science, Springer-Verlag, No.1594, pp. 37-52.

[6] G. Cabri, L. Leonardi, F. Zambonelli, "Engineering Mobile-Agent Applications via Context-dependent Coordination," In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, 2001, pp.371-380.

[7] C. Argel Iglesias, M. Garrijo, José 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.

[8] 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*, 3 (3): 285-312, 2000.

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

[10] C. Fournet, G. Gonthier, J. Lévy, L. Maranget, and D. Rémy, "A Calculus of Mobile Agents," In *Proceedings of the 7th International Conference on Concurrency Theory* (CONCUR'96), Springer-Verlag, LNCS 1119, August 1996, pp. 406-421.

[11] M. Wermelinger, J. L. Fiadeiro, "Connectors for Mobile Programs," *IEEE Transactions on Software Engineering* 24(5), pp. 331-341, May 1998.

[12] A. L. Murphy, G. P. Picco, and G.-C. Roman, "LIME: A Middleware for Physical and Logical Mobility," In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, April 2001, Phoenix, Arizona, USA, pp. 524-533.

[13] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel and M. Strasser, "Communication Concepts for Mobile Agent Systems," In *Proceedings of the 1st International Workshop on Mobile Agents (MA'97)*, Springer Verlag, 1997, pp.123-135.

[14] T. Finin, Y. Labrou and Y. Peng, "Mobile Agents can Benefit from Standards Efforts in Inter-agent Communication," *IEEE Communications Magazine*, Vol. 36, No. 7, pp. 50-56, July 1998.

[15] H. Xu and S. M. Shatz, "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)*, March 2001, Dallas, Texas, USA, pp.11-18.

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

[17] H. Xu and S. M. Shatz, "A Framework for Model-Based Design of Agent-Oriented Software," *Technical Report*, Computer Science Department, The University of Illinois at Chicago, June 2001.

[18] H. Ku H., G. W. Luderer and B. Subbiah, "An Intelligent Mobile Agent Framework for Distributed Network Management," In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'97)*, Phoenix, USA, November 1997.

[19] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, 77(4), April 1989, pp. 541-580.

[20] Deng, Y., 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 Int'l Conf. on Application and Theory of Petri Nets*, Chicago, June 21-25, 1993, pp. 206-223.

[21] 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*, 39(1): 39–59, 1997.

[22] 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.

[23] 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.

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

[25] A. R. Silva, A. Romão, D. Deugo, and M. M. da Silva, "Towards a Reference Model for Surveying Mobile Agent Systems," *Autonomous Agents and Multi-Agent Systems*, Vol. 4, No. 3, pp.187-231, Sept. 2001.