A Framework for Modeling Agent-Oriented Software*

Haiping Xu and Sol M. Shatz
Department of Electrical Engineering and Computer Science
The University of Illinois at Chicago
Chicago, IL 60607
Email: {hxu1, shatz}@eecs.uic.edu

Abstract

With the increasing importance of complex software systems in the software industry, the need for using agent technologies to develop large-scale commercial and industrial software systems is growing rapidly. Such systems are complex and there is a pressing need for system modeling techniques to support reliable, maintainable and extensible design. G-Nets are a type of Petri net defined to support modeling of a system as a set of independent and loosely-coupled modules. In this paper, we first introduce an extension of G-Nets, agent-based G-Net, as a generic model for agent design. Then to progress from an agent-based design model to an agent-oriented model, new mechanisms to support inheritance modeling are introduced. To illustrate our formal modeling technique for multi-agent systems, an example of an agent family in electronic commerce is provided.

1. Introduction

With the increasing importance of complex software systems in the software industry, the need for using agent technologies to develop large-scale commercial and industrial software systems is growing rapidly. Technologies for multi-agent systems (MAS) stem from distributed artificial intelligence (DAI) research [1], and MAS are usually defined as concurrent systems based on the notion of autonomous, reactive, internally-motivated agents in a decentralized environment [2]. The increasing interest in MAS research is due to the significant advantages inherent in such systems, including their ability to solve problems that may be too large for a centralized single agent, to provide enhanced speed and reliability, and to tolerate uncertain data and knowledge [1].

Although there are many efforts on developing multi-agent systems, there is a lack of research on formal specification and design of such systems [3][4]. As multi-agent technology begins to emerge as a viable solution for large-scale industrial and commercial applications, there is an increasing need to ensure that the systems being developed are robust, reliable and fit for purpose [4]. Previous work on formal modeling multi-agent systems includes: (1) using formal languages, such as Z, to provide a framework for describing a system at different levels of abstractions; (2) using temporal modal logic to allow the dynamic aspects of agents; (3) designing formal languages, such as DESIRE, for multi-agent specification [4][12]. Unlike previous work, our approach uses the principle of "separation of concerns" in agent-oriented design. We separate the traditional object-oriented features and reasoning mechanisms in our agent-oriented software model as much as possible, and we discuss how reuse can be achieved in agent-oriented design.

In this paper, we extend a formal model, called a G-Net (a form of Petri net [7]), to support inheritance modeling of agent classes in multi-agent systems. The advantage of our formal mechanism is that it provides a clean interface between agents with asynchronous communication ability and supports formal reasoning for an agent design. Furthermore, our formal mechanism is based on Petri net formalism, which is a mature formal model in terms of both existing theory and tool support.

2. An Agent-based Model

2.1. The Standard G-Net Model

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 [8]. G-Nets are an object-based extension of Petri nets. We assume that the reader has a basic understanding of Petri nets [7], so we begin with some introduction to the G-Net model. 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.

* This material is based upon work supported by the U.S. Army Research Office under grant number DAAD19-99-1-0350, and the NSF under grant number CCR-9988168.

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 – the *Buyer* and the *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 the detailed design of four methods: *buyGoods()*, *askPrice()*, *returnPrice()* and *sellGoods()*. 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* is a method defined in *G'.IS*. 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.

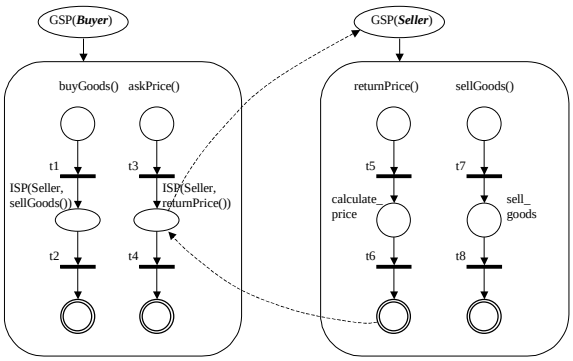


Figure 1. G-Net model of buyer and seller objects

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 [9], we have extended 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.

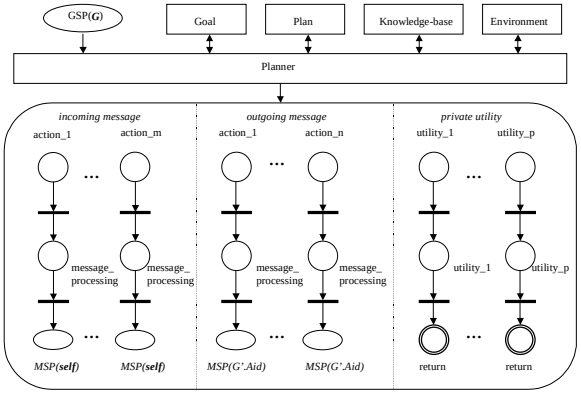
The token movement in a G-Net object is similar to that of original G-Nets [8]. A token *tkn* is a triple *(seq, sc, mtd)*, where *seq* is the propagation sequence of the token, *sc* is the status color of the token and *mtd* is a triple *(mtd_name, para_list, result)*. The usage of these elements can be found in [8][9]. 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. 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 *Seller*, the token is then dispatched into an *entry* place of the appropriate called method, i.e., *returnPrice()*, 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. At this time, the output transition (i.e., *t4* in Figure 1) can become enabled and fire.

We call a G-Net model that supports class modeling a *standard* G-Net model. 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. Although the standard G-Net model works well in object-based design, it is not sufficient in agent-based design for the following reasons. First, agents in multi-agent systems are usually developed by different vendors independently, and those agents will 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 essential for them to have a common communication language and to follow common protocols. However the standard 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 standard 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 and their knowledge. They may autonomously and spontaneously initiate internal or external behavior at any time. Standard G-Net models can only directly support a predefined flow of control.

2.2. Extending G-Nets to Support Agent Modeling

To support agent-based design, we first need to extend a G-Net to support modeling an agent class. The basic idea is similar to extending a G-Net to support class modeling for object-based design [9]. When we instantiate an agent-based G-Net (an agent class model) *G*, an agent identifier *G.Aid* is generated and the mental state of the resulting agent object (an active object [4]) is initialized. In addition, at the class level, five special modules are introduced to make an agent autonomous and internally-motivated, namely the *Goal* module, the *Plan* module, the

Knowledge-base module, the *Environment* module and the *Planner* module. The outline of an agent-based G-Net model is shown in Figure 2. We describe each of the additional modules as follows. A *Goal* module is an abstraction of a goal model [5], which describes the goals that an agent may possibly adopt. A *Plan* module is an abstraction of a plan model [5] that consists of a set of plans. A plan may be intended or committed, and only committed plans will be achieved. A *Knowledge-base* module is an abstraction of a belief model [5], which describes the information about the environment and internal state that an agent of that class may hold. An *Environment* module is an abstract model of the environment, i.e., the model of the outside world that of interest to the agent and that can be sensed by the agent. The *Planner* module can be viewed as the heart of an agent, where committed plans are achieved. It may decide to ignore an incoming message, to start a new conversation, or to continue with the current conversation The *Goal*, *Plan* and *Knowledge-base* modules of an agent are updated after each communicative act or if the environment changes.



Notes: G'.Aid = mTkn.body.msg.receiver as defined later in this section

Figure 2. A generic agent-based G-Net model

The *internal structure (IS)* of an agent model consists of three sections: *incoming message*, *outgoing message*, and *private utility*. The *incoming/outgoing message* section defines a set of *message processing units (MPU)*, which correspond to a subset of communicative acts [10][11]. Each *MPU*, labeled as *action_i* in Figure 2, is used to process incoming/outgoing messages, and may use *ISP*-type modeling for calls to methods defined in its *private utility* section. Only the agent itself can call those private utility functions defined in its *private utility* section.

Although both objects (passive objects) and agents (agent objects) 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 [4]. In particular, these messages must satisfy standardized communicative (speech) acts that define the type and the content of the message (e.g., the FIPA agent communication language, or KQML) [10][11]. Note that in Figure 2, each named *MPU action_i* refers to a communicative act, thus our agent-based 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* (we represent it as an ellipsis in Figure 2), the token is removed and deposited into the *GSP* of the called agent. But, unlike with the standard G-Net *ISP* mechanism, the calling agent does not wait for the token to return before it can continue to execute its next step. Since we usually do not think of agents as invoking methods of one-another, but rather as requesting actions to be performed [12], in our agent-based model, we restrict the usage of *ISP* mechanisms, so they are only used to refer to an agent itself. Thus, in our models, all communications between agents must be carried out through asynchronous message passing as provided by the *MSP* mechanism.

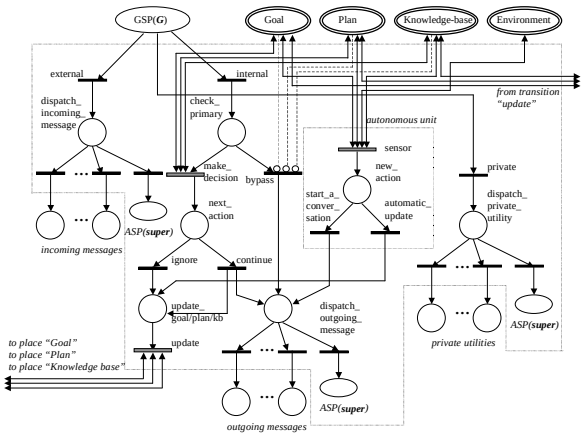


Figure 3. A template of planner module

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, based on an agent's current mental state (goal, plan and knowledge-base), the autonomous unit will then decide 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*.

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 with a tag of **internal**/**external**/**private**. 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 **private** 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**/**private** 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 *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**)*. Here the keyword **self** refers 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") 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 or continued. If the current conversation is ignored, the transition *ignore* 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 is changed from **internal** to **external**. 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.

As a result of this extension, the structure of tokens in the agent-based G-Net model should be redefined. Essentially there are five types of tokens, namely the message token *mTkn*, the goal token *gTkn*, the plan token *pTkn*, the knowledge token *kTkn* and the environment token *eTkn*. One way to construct the *gTkn*, *pTkn*, *kTkn* and *eTkn* is as linked lists. In other words, a *gTkn* represents a list of goals, *pTkn* represents a list of plans, a *kTkn* represents a list of facts, and an *eTkn* represents a list of events that are of the agent's interests. Since these four types of tokens confine themselves to those special places of their corresponding modules, we do not describe them further in this paper.

A *mTkn* is a 2-tuple *(tag, body)*, where *tag* ∈ {**internal**, **external**, **private**} and *body* is a variant, which is determined by the tag. According to the tag, the token deposited in a *GSP* will finally be dispatched into a *MPU* or a *method* defined in the internal structure of the agent-based G-Net. Then the *body* of the token *mTkn* will be interpreted differently. More specifically, we define the *mTkn* body as follows:

```
struct Message{
        int sender;          // the identifier of the message sender
        int receiver;        // the identifier of the message receiver
        string protocol_type;      // contract net protocol type
        string name;         // incoming/outgoing messages name
        string content;      // the content of this message
};

enum Tag {internal, external};

struct MtdInvocation {
        Triple (seq, sc, mtd);      // as defined in Section 2.1
}

if (mTkn.tag ∈ {internal, external})
then mTkn.body  =  struct {
        Message msg;          // message body
}
else mTkn.body =  struct {
       Message msg;          // message body
       Tag old_tag;      // to record the old tag: internal or external
       MtdInvocation miv;    // to trace method invocations
}
```

When *mTkn.tag* ∈ {**internal**, **external**}, and an *ISP* method call occurs, the following steps will take place:

1. The two variables *old_tag* and *miv* are attached to the token *mTkn* to define *mTkn.body.old_tag* and *mTkn.body.miv*, respectively. Then, *mTkn.tag* (the current tag, either **internal** or **external**) is recorded into *mTkn.body.old_tag*, and *mTkn.tag* is temporarily set to **private**.
2. Further method calls are traced by the variable *mTkn.body.miv*, which is a triple of *(seq, sc, mtd)*. The tracing algorithm is as defined in [8].
3. After all the *ISP* method calls are finished and the *mTkn* token returns to the original *ISP*, the *mTkn.tag* is set back as *mTkn.body.old_tag*, and both the variables *old_tag* and *miv* are detached.

We now provide a few key definitions giving the formal structure of our agent-based G-Net models.

Definition 2.1 An *agent-based G-Net* is a 7-tuple *AG = (GSP, GL, PL, KB, EN, PN, IS)*, where *GSP* is a *Generic Switch Place* providing an abstract for *AG*, *GL* is a *Goal* module, *PL* is a *Plan* module, *KB* is a *Knowledge-base* module, *EN* is an *Environment* module, *PN* is a *Planner* module, and *IS* is an *internal structure* of *AG*.

Definition 2.2 A *Planner module* of an agent-based G-Net *AG* is a colored sub-net defined as a 7-tuple *(IGS, IGO, IPL, IKB, IEN, IIS, DMU)*, where *IGS*, *IGO*, *IPL*, *IKB*, *IEN* and *IIS* are interfaces with *GSP*, *Goal* module, *Plan* module, *Knowledge-base* module, *Environment* module and *internal structure* of *AG*, respectively. *DMU* is a set of decision-making unit, and it contains three abstract transitions: *make_decision*, *sensor* and *update*.

Definition 2.3 An *internal structure (IS)* of an agent-based G-Net *AG* is a triple *(IM, OM, PU)*, where *IM/OM* is the *incoming/outgoing message* section, which defines a set of *message processing units (MPU)*; and *PU* is the *private utility* section, which defines a set of *methods*.

Definition 2.4 A *message processing unit (MPU)* is a triple *(P, T, A)*, where *P* is a set of places consisting of three special places: *entry* place, *ISP* and *MSP*. Each *MPU* has only one *entry* place and one *MSP*, but it may contain multiple *ISP*s. *T* is a set of transitions, and each transition can be associated with a set of guards. *A* is a set of arcs defined as: ((P-{*MSP*}) x T) ∪ ((T x (P-{*entry*}).

Definition 2.5 A *method* is a triple *(P, T, A)*, where *P* is a set of places with three special places: *entry* place, *ISP* and *return* place. Each method has only one *entry* place and one *return* place, but it may contain multiple *ISP*s. *T* is a set of transitions, and each transition can be associated with a set of guards. *A* is a set of arcs defined as: ((P-{*return*}) x T) ∪ ((T x (P-{*entry*}).

2.3. Inheritance Modeling in Agent-based G-Nets

Although there are different views with respect to the concept of agent-oriented design [12], we consider an agent as an extension of an object, and we believe that agent-oriented design should keep most of the key features in object-oriented design. Thus, to progress from an agent-based model to an agent-oriented model, we 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 private utility functions. There is existing work on agent inheritance with respect to knowledge, goals and plans [2][6]. However, we believe that 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 a individual agent object (as an instance of an agent superclass), can not be inherited by an agent object when creating an instance of an agent subclass. A superclass's reasoning mechanism can be inherited, however it is beyond the scope of this paper. For simplicity, we assume that an agent subclass always uses its own reasoning mechanisms, and thus the reasoning mechanisms in the agent superclass should be disabled in some way. This is necessary because different reasoning mechanisms may deduce different results for an agent, and to resolve this type of conflict may be time-consuming and make an agent's reasoning mechanism inefficient. Therefore, in this paper 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 2.6 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-based G-Net, an initialized agent class is modeled by an agent-based G-Net with initialized state variables. In particular, the four tokens in the special places of an agent-based 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 these tokens are combined with the tokens *gTkn*, *pTkn*, *kTkn* and *eTkn* in the primary subagent of *AO*.[1] 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 into a *MPU* defined in the primary agent or any subagents.

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-based G-Net models, and we assume that all incoming messages have been correctly defined in the primary subagent or some other subagents.

3. Examples of Agent-Oriented Design

Consider an agent family in an electronic marketplace domain. Figure 4 shows the agents in a UML class hierarchy notation. A shopping agent is defined as an abstract agent that has the ability to register in a marketplace through a facilitator, which serves as a well-known agent in the marketplace. A shopping agent cannot be instantiated as an agent object, in other words, a shopping agent cannot register itself as a shopping agent. Rather, the functionality of a shopping agent class can be inherited by an agent subclass, such as a buying agent or a selling agent. Both the buying agent object and selling agent object may reuse the functionality of a shopping agent by registering themselves as a buying agent or a selling agent through a facilitator. Furthermore, a retailer agent is an agent that can sell goods to a customer, but it also needs to buy goods from some selling agents. Thus a retailer agent class is designed as a subclass of both the buying agent class and the selling agent class. In addition, a customer agent class may be defined as a subclass of a buying agent class, and an auctioneer agent class may be defined as a subclass of a selling agent class. In this paper, we only consider four types of agent class, i.e., the shopping agent class, the buying agent class, the selling agent class and the retailer agent class. The modeling of the customer agent class and auctioneer agent class can be done in a similar way.

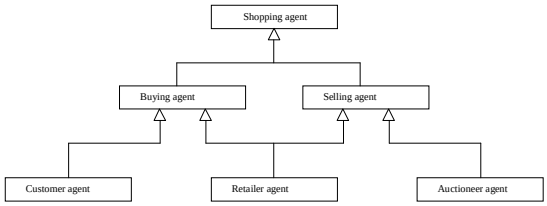


Figure 4. The class hierarchy diagram of agents in an electronic marketplace

To illustrate how to design agents by using our agent model, we use the following examples. Figure 5 (a) depicts

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

a template of a contract net protocol expressed as an agent UML (AUML) sequence diagram [11] for a registration-negotiation protocol between a shopping agent and a facilitator agent. Figure 5 (b) is a modified example of a contract net protocol adapted from [11], which depicts a template of a protocol expressed as an AUML sequence diagram for a price-negotiation protocol between a buying agent and a selling agent. Some of the notations of AUML are adapted from [11] 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 "∗". Figure 5 (c) shows an example price-negotiation protocol that is instantiated from the protocol template shown in Figure 5 (b).

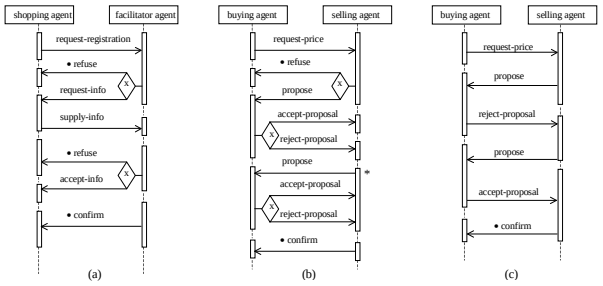


Figure 5. Contract net protocols (a) A template for the registration protocol (b) A template for the price-negotiation protocol (c) An example of the price-negotiation protocol

Consider Figure 5 (a). When a conversation based on a contract net protocol begins, the shopping agent sends a request for registration to a facilitator agent. The facilitator agent can then choose to respond to the shopping agent by refusing its registration or requesting agent information. Here the "x" in the decision diamond indicates an exclusive-or decision. If the facilitator refuses the registration based on the marketplace's size, the protocol ends; otherwise, the facilitator agent waits for agent information to be supplied. If the agent information is correctly provided, the facilitator agent then still has a choice of either accepting or rejecting the registration based on the shopping agent's reputation and the marketplace's functionality. Again, if the facilitator agent refuses the registration, the protocol ends; otherwise, a confirmation message will be provided afterwards. Similarly, the price-negotiation protocol between a buying agent and a selling agent can be illustrated in Figure 5 (b). Based on the communicative acts (e.g., request-registration, refuse, etc.) needed for the contract net protocol in Figure 5 (a), we may adopt the design template of the shopping agent shown 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 facilitator agent is similar, however it may support more protocols.

With inheritance, a buying agent class, as a subclass of a shopping agent class, may reuse *MPUs/methods* defined in a shopping agent class's internal structure. Similarly, based on the communicative acts (e.g., request-price, refuse, etc.) needed for the contract net protocol in Figure 5 (b), we may design the buying agent class as in Figure 7. Note that we do not define the *MPUs* of *refuse* and *confirm* in the internal structure of the buying agent class, for they can be inherited from the shopping agent class. A retailer agent can be designed in the same way. In addition to its own *MPU/methods*, a retailer agent class inherits all *MPU/methods* of both the buying agent class and the selling agent class.

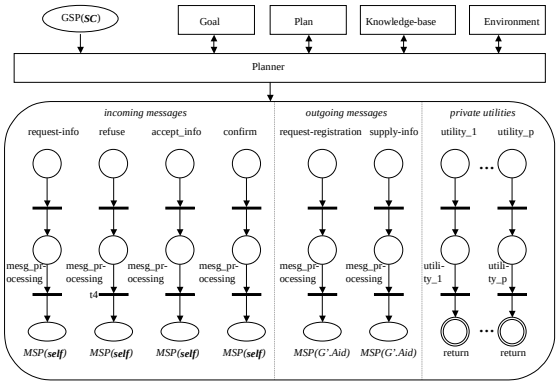


Figure 6. An agent-based G-Net model for shopping agent class

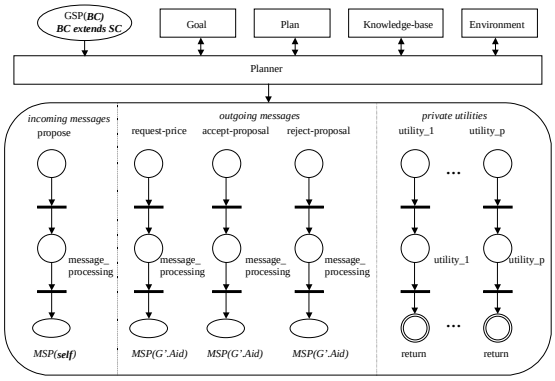


Figure 7. An agent-based G-Net model for buying agent class

Now we discuss an example to show how the reuse of *MPU/methods* works. Consider a buying agent object *BO*, which receives a message of *request-info* from a facilitator agent object *FO*. A *mTkn* token will be deposited in the *GSP* of the primary subagent of *BO*, i.e., the *GSP* of the corresponding buying agent class (*BC*). The transition *external* in *BC*'s *Planner* module may fire, and the *mTkn* will be moved to the place *dispatch_incoming_message*. Since there is no *MPU* for *request-info* defined in the internal structure of *BC*, the *mTkn* will be moved to the *ASP(**super**)* place. Since ***super*** here refers to a unique superclass – the shopping agent class (*SC*) – the *mTkn* will

be transferred to the *GSP* of *SC*. Now the *mTkn* can be correctly dispatched to the *MPU* for *request-info*. 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 *BC*. 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 *BC* may fire, and the *mTkn* token will be moved to the place *check_primary*. Since *BC* corresponds to the primary subagent of *BO*, 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 *supply-info*, following the protocol defined in Figure 5 (a). Again, there is no *MPU* for *supply-info* defined in *BC*, so the new *mTkn* token will be dispatched into the *GSP* of *SC*. Upon the arrival of the *mTkn* in the *GSP* of *SC*, the transition *internal* in the *Planner* module of *SC* may fire. However at this time, *SC* does not correspond to the primary subagent of *BO*, so all the tokens in the special places of *Goal*, *Plan*, *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 *supply-info* defined in *SC*. 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 facilitator agent.

For the reuse of private utility functions defined in a superclass, the situation is the same as in the case of object-oriented design. Examples concerning reuse of private utility functions and different forms of inheritance, such as augment inheritance and restrictive inheritance, can be found in [9].

4. Conclusions and Future Work

One of the most rapidly growing areas of interest for distributed computing is that of distributed agent systems. In this paper, we introduced a framework of agent models with an example of agent family in electronic commerce. Using this framework, shopping agent, selling agent, buying agent and retailer agent can be modeled as intelligent agents with the characteristics of being autonomous, reactive and internally-motivated. Examples of a registration-negotiation protocol between shopping agents and facilitator agents, and a price-negotiation protocol between a shopping agents and buying agents were used to illustrate our basic idea.

For our future work, we will consider the refinement of the *Goal*, *Plan*, *Knowledge-base* and *Environment* modules. The abstract transitions defined in the *Planner* module, i.e., *make_decision*, *sensor* and *update*, will be refined into correct sub-nets too. We will also look into issue like deadlock avoidance and state exploration problems in the agent-oriented design and verification processes.

5. References

[1] S. Green, L. Hurst, B. Nangle, P. Cunningham, F. Somers, R. Evans, "Software Agents: A Review," *Technical report TCD-CS-1997-06*, Trinity College Dublin, May 1997.

[2] David Kinny, Michael 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] T. J. Rogers, R. Ross, V. S. Subrahmanian, "IMPACT: A System for Building Agent Applications," *Journal of Intelligent Information Systems*, 14(2-3): 95-113, 2000.

[4] Carlos Argel Iglesias, Mercedes 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.

[5] D. Kinny, M. Georgeff, and A. Rao, "A Methodology and Modeling Technique for Systems of BDI Agents," *Tech. Rep. 58*, Australian Artificial Intelligence Institute, Melbourne, Australia, Jan. 1996.

[6] Lobel Crnogorac, Anand S. Rao, Kotagiri Ramamohanarao, "Analysis of Inheritance Mechanisms in Agent-Oriented Programming," *IJCAI* (1) 1997: 647-654.

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

[8] 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.

[9] Haiping Xu and Sol Shatz, "Extending G-Nets to Support Inheritance Modeling in Concurrent Object-Oriented Design," *IEEE Int'l Conf. on Systems, Man, and Cybernetics*, Nashville, Tenn., Oct. 2000, pp. 3128-3133.

[10] Tim Finin, Yannis Labrou, and James Mayfield, "KQML as an agent communication language," in Jeff Bradshaw (Ed.), *Software Agents*, MIT Press, Cambridge, 1997.

[11] James Odell, H. Van Dyke Parunak, Bernhard Bauer, "Representing Agent Interaction Protocols in UML," *ICSE 2000 Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, June 10, 2000, Limerick, Ireland.

[12] M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," *International Journal of Autonomous Agents and Multi-Agent Systems*, 3(3): 285-312, 2000.