A Framework for Model-Based Design of Agent-Oriented Software[1]

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

Abstract

Agents are becoming one of the most important topics in distributed and autonomous decentralized systems, and there are increasing attempts to use agent technologies to develop large-scale commercial and industrial software systems. The complexity of such systems suggests 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 system modeling in terms of a set of independent and loosely-coupled modules. In this paper, we customize the basic G-net model to define a so-called "agent-based G-net" that can serve 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. Finally, we demonstrate how we can use model checking to verify some key behavioral properties of our agent model. This is facilitated by the use of an existing Petri net tool.

Keywords: Multi-agent systems, Petri net, G-net, design model, electronic commerce, model checking

1. Introduction

Over the past decade, research and development efforts in computer science have increasingly embraced the concept of software agents and multi-agent systems. One key reason is that the idea of an agent as an autonomous system, capable of interacting with other agents in order to satisfy its design objectives, is a naturally appealing one for software designers. This has led to the growth of interest in agents as a new design-paradigm for software engineering [1].

[1] This material is based upon work supported by the U.S. Army Research Office under grant number DAAD19-99-1-0350, and the U.S. National Science Foundation under grant number CCR-9988168.

Applications that can most directly benefit from an agent-oriented design are typically structured as multi-agent systems (MAS), which are usually defined as a concurrent system based on the notion of autonomous, reactive and internally-motivated agents in a decentralized environment [2]. One example of such an application is intelligent team training environments [3]. Many of the technologies supporting multi-agent systems stem from distributed artificial intelligence (DAI) research [4]. 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 [4].

Although there are many efforts aimed at developing multi-agent systems, there is sparse research on formal specification and design of such systems [5][6]. As multi-agent technology begins to emerge as a viable solution for large-scale applications, there is an increasing need to ensure that the systems being developed are robust, reliable and fit for purpose [7]. Previous work on formal modeling agent systems includes: (1) using formal languages, such as Z, to provide a framework for describing the agent architecture at different levels of abstractions [8]; (2) using temporal logics and multi-modal logics to represent individual agent behaviors where the representations can be executed directly, e.g., Fisher's work on Concurrent METATEM [9], and (3) designing formal languages, such as DESIRE and SLABS, for specifying agent-based systems [10][11]. Although these formalisms are claimed to be agent specifications, they are not oriented for software engineering in terms of providing a modeling notation that directly supports software development. For instance, as stated in paper [12], formalisms such as temporal logics and multi-modal logics are often abstract and quite distant from agents that have actually been implemented. There are previous efforts to narrow the gap between agent formal models and agent-based practical systems, e.g., to use formal approaches for prototyping and simulation of multi-agent systems [13]; however, it is still hard to apply these formal methods directly to agent implementation. In contrast, our approach is explicitly oriented for specifying and defining the design architecture of multi-agent software systems. Also, unlike most previous work, our approach exploits the principle of "separation of concerns" in an agent-oriented design, similar to the basic idea proposed for some distributed object architectures that define a meta agent (on top of a base-level object) to handle non-functional requirements [14]. Specifically, we separate the traditional object-oriented features and reasoning mechanisms in our agent-oriented software model, and we discuss how reuse can be achieved in terms of functional units in agent-oriented design. While some have advocated that inheritance has limited value in conceptual models of agent behavior [15], we illustrate a useful role for inheritance in our agent-oriented models. Our agent-based model is derived from the general agent model given in [16], and the extensions that create an agent-oriented model are derived from the framework presented in [17]. At the heart of our approach is the use of a model that is rooted in the Petri net formalism [18]. As such, this work is complementary to other research efforts that use Petri nets to model the mental states of agents as part of an architecture for multi-agent simulation [19].

The rest of this paper is organized as follows. Section 2 begins with a brief introduction to the standard G-net model, an object-based Petri net notation. It then presents the general structure of the proposed agent-based G-net model based on BDI models [20], and discusses how inheritance modeling can be integrated into agent-based G-net models. Section 3 provides an example agent family in electronic commerce to illustrate our approach to agent design and inheritance modeling. Section 4 verifies some behavioral properties of our agent model by using Petri net theory and an existing Petri net tool. Finally, Section 5 provides a brief conclusion and mentions future work.

2. An Agent-Oriented 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 [21][22]. 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 [18]. 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 standard G-net models.

A G-net system is composed of a number of G-nets, each of them representing a self-contained module or object. A G-net is composed of two parts: a special place called *Generic Switch Place (GSP)* and an *Internal Structure (IS)*. The *GSP* provides the abstraction of the module, and serves as the only interface between the G-net and other modules. The *IS*, a modified Petri net, represents the design of the module. An example of G-nets is shown in Figure 1. Here the G-net models represent two objects – a *Buyer* and a *Seller*. The generic switch places are represented by *GSP(Buyer)* and *GSP(Seller)* enclosed by ellipses, and the internal structures of these models are represented by round-cornered rectangles that contain four methods: *buyGoods()*, *askPrice()*, *returnPrice()* and *sellGoods()*. The functionality of these methods are defined as follows: *buyGoods()* invokes the method *sellGoods()* defined in G-net *Seller* to buy some goods; *askPrice()* invokes the method *returnPrice()* defined in G-net *Seller* to get the price of some goods; *returnPrice()* is defined in G-net *Seller* to calculate the latest price for some goods and *sellGoods()* is

defined in G-net *Seller* to wait for the payment, ship the goods and generate the invoice. A *GSP* of a G-net *G* contains a set of methods *G.MS* specifying the services or interfaces provided by the module, and a set of attributes, *G.AS*, which are state variables. In *G.IS*, the internal structure of G-net *G*, Petri net places represent primitives, while transitions, together with arcs, represent connections or relations among those primitives. The primitives may define local actions or method calls. Method calls are represented by special places called *Instantiated Switch Places (ISP)*. A primitive becomes *enabled* if it receives a token, and an enabled primitive can be executed. Given a G-net *G*, an *ISP* of *G* is a 2-tuple *(G'.Nid, mtd)*, where *G'* could be the same G-net *G* or some other G-net, *Nid* is a unique identifier of G-net *G'*, and *mtd* ∈ *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.

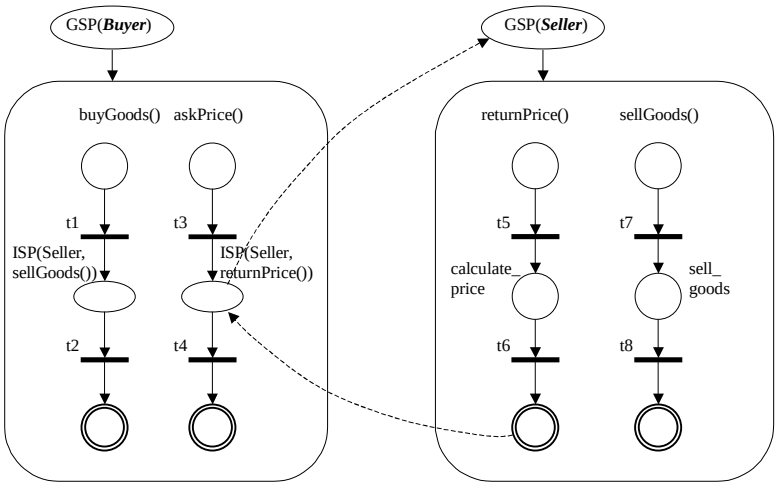


Figure 1. G-net model of buyer and seller objects

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 [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.

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

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. Further details about G-net models can be found in references [21][22][23].

2.2 An Architecture for Agent-Based Modeling

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 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 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, their knowledge and the environment. They may autonomously and spontaneously initiate internal or external behavior at any time. The standard G-net models can only directly support a predefined flow of control.

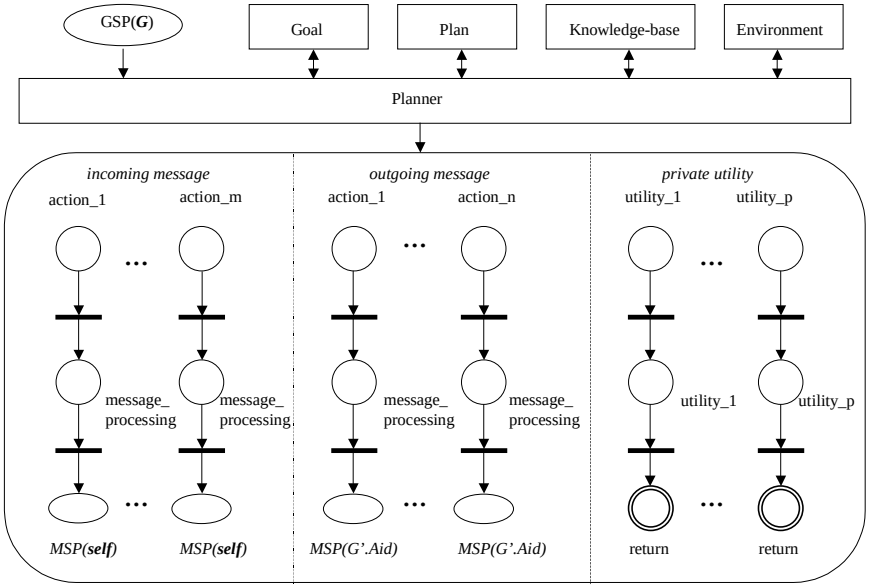
To support agent-based design, we need to extend a G-net to support modeling an agent class[2]. The basic idea is similar to extending a G-net to support class modeling for object-based design [23]. When we instantiate an agent-based G-net (an agent class model) *G*, an agent identifier *G.Aid* is generated and the mental state of the resulting agent object (an active object [7]) is initialized. In addition, at the class level, five special modules are introduced to make an agent autonomous and internally-motivated. They are the *Goal* module, the *Plan* module, the *Knowledge-base* module, the *Environment* module and the *Planner* module. Note that the *Goal*, *Plan* and *Knowledge-base* module are based on the BDI agent model proposed by Kinny and his colleagues [20].

The template for 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 [20], which describes the goals that an agent may possibly adopt, and the events to which it can respond. It consists of a goal set which specifies the goal domain and one or more goal states. A *Plan* module is an abstraction of a plan model [20] that consists of a set of plans, known as a plan set. 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 [20], which describes the information about the environment and internal state that an agent of that class may hold. The possible beliefs of an agent are described by a belief set. An *Environment* module is an abstract model of the environment, i.e., the model of the outside world of an agent. The *Environment* module only models elements in the outside world that are of interest to the agent and that can be sensed by the agent.

In the *Planner* module, committed plans are achieved, and the *Goal*, *Plan* and *Knowledge-base* modules of an agent are updated after the processing of each communicative act that defines the type and the content of a message [24][25], or if the environment changes. Thus, the *Planner* module can be viewed as 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.

The *internal structure (IS)* of an agent-based G-net 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 corresponds to a subset of communicative acts. 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. Unlike with the methods defined in a standard G-net model, the private utility functions or methods defined in the *private utility* section can only be called by the agent itself.

[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 as defined later in this section

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

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 [7]. In particular, these messages must satisfy the format of the standardized communicative (speech) acts, e.g., the format of the communicative acts defined in the FIPA agent communication language, or KQML [24][25][26]. 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 [27], 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, one agent may not directly invoke a method defined in another agent. All communications between agents must be carried out through asynchronous message passing as provided by the *MSP* mechanism.

A template of the *Planner* module is shown in Figure 3[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*.

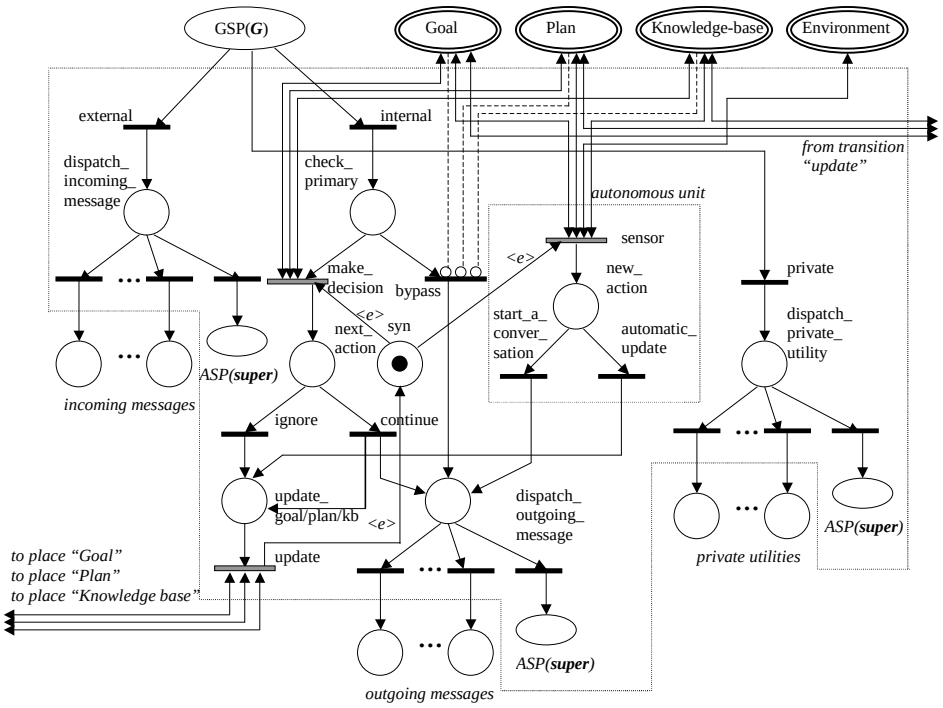


Figure 3. A template for the planner module (initial design)

[3] Actually, this module purposely contains a somewhat subtle design error that is used to demonstrate the value of automated verification in section 4.

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 **internal** represents a message forwarded by an agent to itself with the *MSP* mechanism, or a newly generated outgoing message before sending to some other agent; while a message token with a tag of **external** is an incoming message which comes from some other agent. In either case, the message token with the tag of **internal**/**external** should not be involved in an invocation of a method call. In contrast, 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**)*. 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"[4]) 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 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.

To ensure that all decisions are made upon the latest mental state of the agent, i.e., the latest values in the goal, plan, and knowledge-base modules, and similarly to ensure that the sensor always captures the latest mental state of the agent, we introduce a synchronization unit *syn*, modeled as a place marked with an

[4] 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.

ordinary token (black token). The token in place *syn* will be removed when the abstract transition *make_decision* or *sensor* fires, thus delaying further firing of these two abstract transitions until completion of actions that update the values in the goal, plan and knowledge-base modules. This mechanism is intended to guarantee the mutual exclusive execution of decision-making, capturing the latest mental state and events, and updating the mental state. Note that we have used the label <*e*> on each of the arcs connecting with the place *syn* to indicate that only ordinary tokens may be removed from or deposited into the place *syn*.

As a result of this extension to G-nets, the structure of tokens in the agent-based G-net model should be redefined. In addition to the ordinary token introduced in place *syn*, essentially there are five types of colored 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;   // the type of contract net protocol
    string name;            // the name of incoming/outgoing messages
    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 {  // mTkn.tag equals to the tag: private
    Message msg;            // message body
    Tag old_tag;            // to record the old tag: internal/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 *mTkn* to define *mTkn.body.old_tag* and *mTkn.body.miv*, respectively. Then, *mTkn.tag* (the current tag, one of **internal** or **external**) is recorded into *mTkn.body.old_tag*, and *mTkn.tag* is 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 defined as in the original G-net definitions [21].
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.

The *MSP(id)* mechanism defined in an agent *AO* is responsible for asynchronously transferring a message token *mTkn* to the agent itself or some other agent, and for changing the tag of the message token, *mTkn.tag*, before *mTkn* is "sent out." The steps for handling the message token are as follows:

1. If *id* equals to ***self*** (in this case *mTkn.tag* must be **external**), set *mTkn.tag* to **internal**, and transfer the message token *mTkn* to the *GSP* place of agent *AO*.
2. Else-If *id* equals to *G'.Aid*, where *G'.Aid* does not represent the agent *AO* (in this case *mTkn.tag* must be **internal**), set *mTkn.tag* to **external**, and transfer the message token *mTkn* to the *GSP* place of the agent represented by *G'.Aid*.

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

Definition 2.1 *Agent-based G-net*

An *agent-based G-net* is a 7-tuple *AG = (GSP, GO, PL, KB, EN, PN, IS)*, where *GSP* is a *Generic Switch Place* providing an abstract for the agent-based G-net, *GO* 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 *Planner Module*

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 *Internal Structure (IS)*
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 *Message Processing Unit (MPU)*
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: $((P-\{MSP\}) \times T) \cup ((T \times (P-\{entry\})$.

Definition 2.5 *Method*
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: $((P-\{return\}) \times T) \cup ((T \times (P-\{entry\})$.

2.3 Inheritance Modeling

Although there are different views with respect to the concept of agent-oriented design [15], 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][28]. 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 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. 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 *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-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 the tokens are combined with the *gTkn*, *pTkn*, *kTkn* and *eTkn* in the primary subagent of *AO*.[5] 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

[5] 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.

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-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

3.1 A Hierarchy of Agents in an Electronic Marketplace

Consider an agent family in an electronic marketplace domain. Figure 4 shows the agents in a UML class hierarchy notation. A shopping agent class is defined as an abstract agent class 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 class cannot be instantiated as an agent object; however the functionality of a shopping agent class can be inherited by an agent subclass, such as a buying agent class or a selling agent class. Both the buying agent and selling agent may reuse the functionality of a shopping agent class 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.

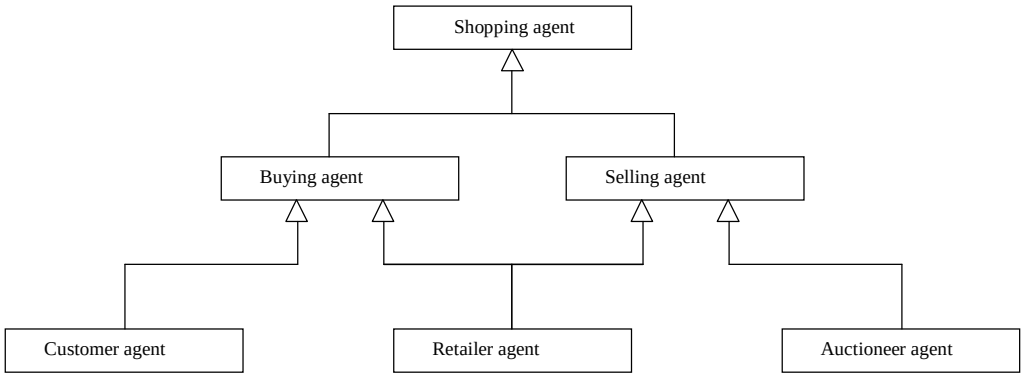


Figure 4. The class hierarchy diagram of agents in an electronic marketplace

3.2 Modeling Agents in an Electronic Marketplace

To illustrate the processes for design of agents by using our generic agent model, we use the following examples. Figure 5 (a) depicts a template of a contract net protocol [29] expressed as an agent UML (AUML) sequence diagram [26] for a registration-negotiation protocol between a shopping agent and a facilitator agent. Note that although AUML is on the way to be standarized, many researchers have attempted to exploit UML to support design of multi-agent systems [30][31][26]. Figure 5 (b) is a modified example of a contract net protocol adapted from [26], which depicts a template of a price-negotiation protocol between a buying agent and a selling agent. Figure 5(c) shows an example of price-negotiation contract net protocol that is instantiated from the protocol template in Figure 5(b). Some of the notations of AUML are adapted from [26] 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 "∗". Examples of using these two notations are as follows. In Figure 5 (a), we put a mark of "•" in front of the message name "*refuse*" to indicate that this message ends the protocol. In Figure 5 (b), a mark "∗" is put on the right corner of the narrow rectangle for the message "*propose*" to indicate that the communication actions in this section can be repeated zero or more times.

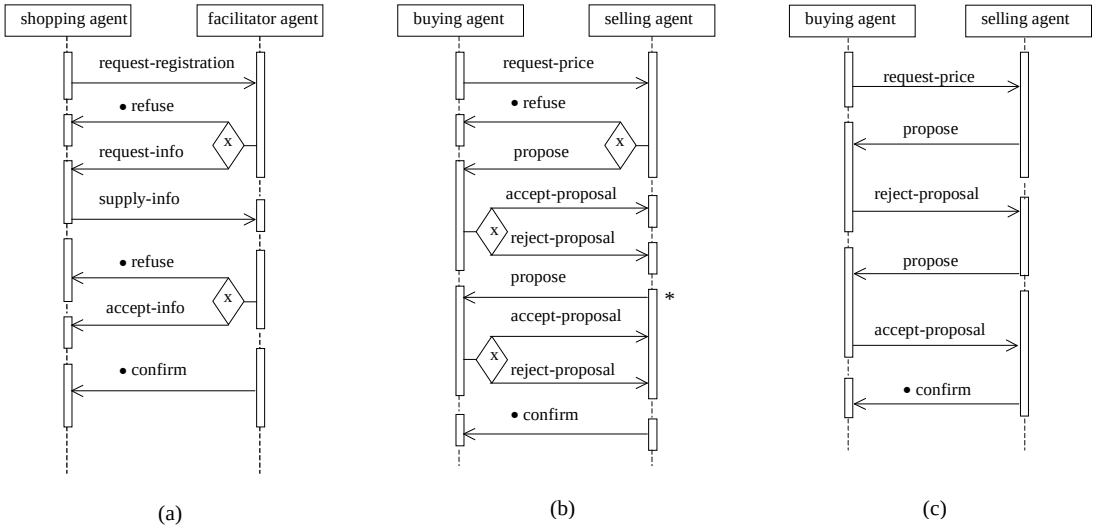


Figure 5. Contract net protocols (a) A template for the registration protocol (b) A template for the price-negotiation protocol (c) An example of the price-negotiation protocol

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 between a buying agent and a selling agent is clearly 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 design the shopping 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 facilitator agent class is similar, however it may support more protocols and should define more *MPUs* and *methods* in its internal structure.

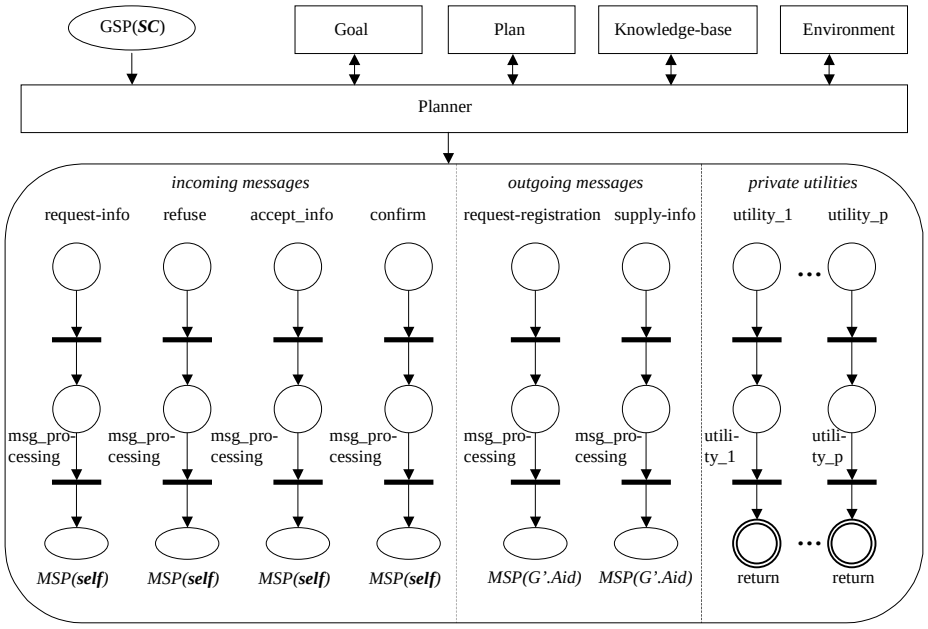


Figure 6. An agent-based G-net model for shopping agent class (SC)

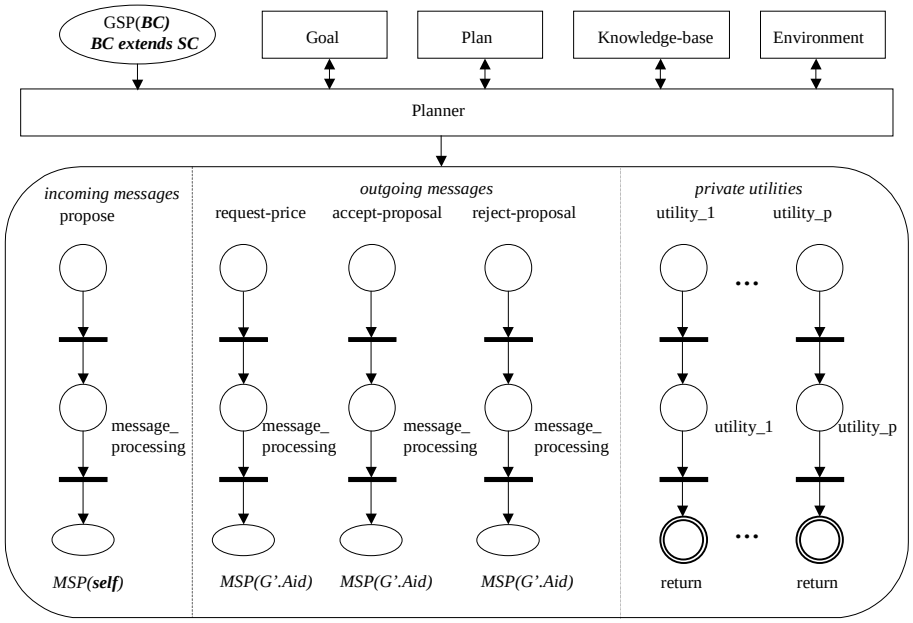


Figure 7. An agent-based G-net model for buying agent class (BC)

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 selling agent class or a retailer agent class can be designed in the same way. In addition to their own *MPU/methods*, a selling agent class inherits all *MPU/methods* of the shopping agent class, and a retailer agent class inherits all *MPU/methods* of both the buying agent class and the selling 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 receiver agent, in this case, 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. 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 private utility functions and different forms of inheritance can be found in [23].

With single inheritance, the ***super*** in *ASP(**super**)* in an agent object *AO*, as an instance of an agent class *A*, refers to the subagent of *AO*, which corresponds to the unique superclass of *A*. However, with multiple inheritance, ***super*** may refer to any one of the subagents, which corresponds to a superclass or an ancestor classes of *A*. One way to resolve the reference of ***super*** is to use a modified breadth-first-search of the inheritance hierarchy graph to find the appropriate reference of ***super***. Due to lack of space, we do not discuss further details on this issue.

4. Analysis of Agent-Oriented Models

One of the advantages of building a formal model for agents in agent-oriented design is to help ensure a correct design that meets certain specifications and system requirements. A correct agent design should meet certain key requirements, such as liveness, deadlock freeness and concurrency. Also certain properties, such as the inheritance mechanism, need to be verified to ensure its correct functionality. Petri nets offer a promising, tool-supported technique for checking the logic correctness of a design. In this section, we use a Petri net tool, called INA (Integrated Net Analyzer) [32], to analyze and verify our agent models. We use an example of a simplified Petri net model for the interaction between a single buying agent and two selling agents.

The INA tool is an interactive analysis tool that incorporates a large number of powerful methods for analysis of Petri nets [32]. These methods include analysis of (1) structural properties, such as structural boundedness, and T- and P-invariant analysis; (2) behavioral properties, such as boundedness, safeness, liveness, and deadlock-freeness; and (3) model checking, such as checking Computation Tree Logic (CTL) formulas. These analyses employ various techniques, such as linear-algebraic methods (for invariants), reachability and coverability graph traversals. Here we focus on behavioral property verification by model checking.

4.1 A Simplified Petri net Model for a Buying Agent and Two Selling Agents

The interaction of one buying agent and two selling agents can be modeled as a net as in Figure 8. Table 1 and Table 2 provide a legend that identifies the meaning associated with each place and transition in Figure 8. To derive this net model, we use a *GSP* place to represent each selling agent. This is practical because

an agent-based G-net model can be abstracted as a single *GSP* place, and agent models can only interact with each other through *GSP* places. Meanwhile, the net for the buying agent, whose class is a subclass of a shopping agent class, is simplified as follows:

1. Since the special places of *Goal*, *Plan*, *Knowledge-base* have the same interfaces with the planner module in an agent class, we fuse them into one single place *goal/plan/kb*. Furthermore, we simplify this fused place *goal/plan/kb* and the place of *environment* as ordinary places with ordinary tokens.
2. We omit the *private utilities* sections in both the shopping subagent model and the buying primary subagent model. Thus, to obtain our simplified model, we do not need to translate the *ISP* mechanism, although such a translation to a Petri net form can be found in [21].
3. We simplify *mTkn* tokens as ordinary tokens. Although this simplification will cause the reachability graph of our transformed Petri net to become larger, this simplifies the message tokens, allowing us to ignore message details, which is appropriate for the purpose in this paper (we will explain it further in Section 4.3).
4. We use net reduction (i.e., net transformation rules [33]) to simplify the Petri net corresponding to an *MPU/Method* as a single place. For instance, the *MPU* identified as *propose* in Figure 7 is represented as place *P25* in Figure 8.
5. We use the closed-world assumption and consider a system that only contains three agents, i.e., a buyer agent and two seller agents. We assume that a buying agent initiates a conversation. A system that contains more than three agents can be verified by the same technique.

4.2 Deadlock Detection and Redesign of Agent-Oriented Models

Now we use the INA tool to analyze the simplified agent model illustrated in Figure 8. To reduce the state space, we further reduce the net by fusing the *MPU*s in the same *incoming/outgoing message* section. For instance, in Figure 8, we fuse the places *P8*, *P9*, *P10* and *P11* into one single places. Obviously, this type of net reduction [33] does not affect the properties of liveness, deadlock-freeness and the correctness of inheritance mechanism. In addition, we set the capacity of each place in our net model as 1, which means at any time, some processing units, such as *MPU*s, can only process one message. However, the property of concurrency is still preserved because different transitions can be simultaneously enabled (and not in conflict); providing the standard Petri net notion of concurrency based on the interleaved semantics. For example, transitions t25 and t27 can be simultaneously enabled, representing that message processing for a conversation and decision-making for another conversation can happen at the same time.

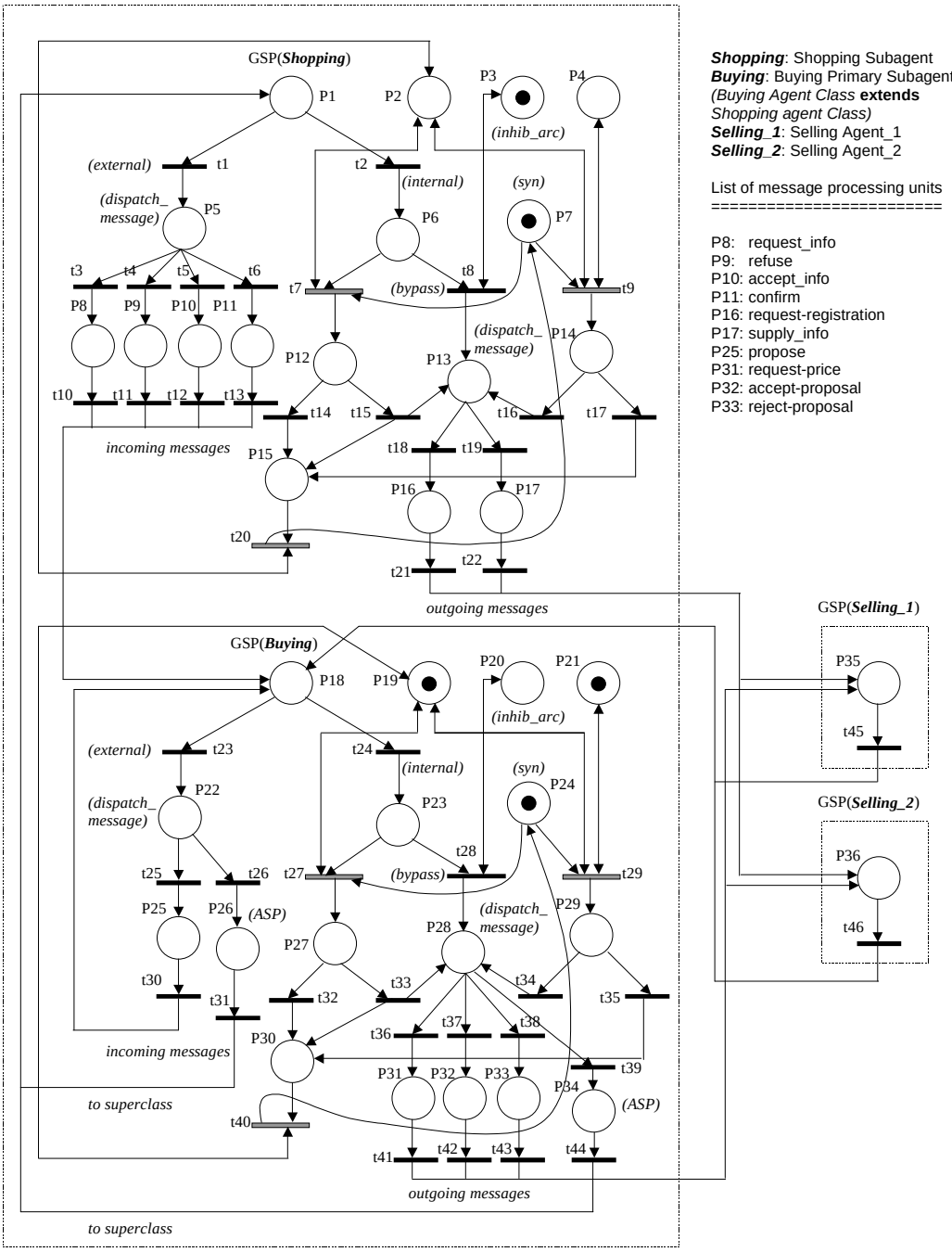


Figure 8. A transformed model of one buying agent and two selling agents

Table 1

LEGEND FOR FIGURE 8 (DESCRIPTION OF PLACES)

Place	Description
P1 / P18	The *GSP* place of the shopping subagent / buying primary subagent.
P2 / P19	The merged place for the *Goal*, *Plan* and *Knowledge-base* module of the shopping subagent / buying primary subagent.
P3 / P20	The complementary place of P2 / P19 introduced to remove the inhibitor arcs.
P4 / P21	The place for the *Environment* module of the shopping subagent / buying primary subagent.
P5 / P22	The place for dispatching incoming messages.
P6 / P23	The place for checking if the current subagent is a primary subagent
P7 / P24	Synchronization place for making decision, updating mental state and capturing internal/external events.
P8 / P9 / P10 / P11	The place for the message processing unit (*MPU*) of *request-info* / *refuse* / *accept-info* / *confirm*.
P12 / P27	The place for choosing the next action: to ignore or to continue with the current conversation.
P13 / P28	The place for dispatching outgoing messages.
P14 / P29	The place for choosing a new action: to start a conversation or to automatically update the agent mental state.
P15 / P30	The place for updating the agent mental state.
P16 / P17	The place for the message processing unit (*MPU*) of *request-registration* / *supply-info*.
P25	The place for the message processing unit (*MPU*) of *propose*.
P26	Asynchronous superclass switch place (*ASP*)
P31 / P32 / P33	The Place for the message processing unit (*MPU*) of *request-price* / *accept-proposal* / *reject-proposal*.
P34	Asynchronous superclass switch place (*ASP*)
P35	The *GSP* place of selling agent_1 (we use the *GSP* place to represent the whole agent).
P36	The *GSP* place of selling agent_2 (we use the *GSP* place to represent the whole agent).

Table 2

LEGEND FOR FIGURE 8 (DESCRIPTION OF TRANSITIONS)

Transition	Description
t1 / t23	The transition *external*, which fires when the token from the *GSP* has a tag of **external**.
t2 / t24	The transition *internal*, which fires when the token from the *GSP* has a tag of **internal**.
t3, t10	Transitions related to the message processing unit (*MPU*) of *request-info*.
t4, t11	Transitions related to the message processing unit (*MPU*) of *refuse*.
t5, t12	Transitions related to the message processing unit (*MPU*) of *accept-info*.
t6, t13	Transitions related to the message processing unit (*MPU*) of *confirm*.
t7 / t27	The abstract transition *make_decision*, which determines the next action to perform.
t8 / t28	The transition *bypass*, which is disabled when there are tokens in place P2 / P19, i.e., there is no token in place P3 / P20. Notice that P3 / P20 is a complementary place of P2 / P19.
t9 / t29	The abstract transition *sensor*, which captures internal and external events.
t14 / t32	The transition *ignore* that ignores the current conversation.
t15 / t33	The transition *continue* that continues with the current conversation.
t16 / t34	The transition *start_a_conversation* that starts a new conversation.
t17 / t35	The transition *automatic_update* that automatically updates the agent's mental state.
t18, t21	Transitions related to the message processing unit (*MPU*) of *request-registration*.
t19, t22	Transitions related to the message processing unit (*MPU*) of *supply-info*.
t20 / t40	The abstract transition *update_goal/plan/kb*, which updates the agent's mental state.
t25, t30	Transitions related to the message processing unit (*MPU*) of *propose*.
t26, t31	Transitions related to the asynchronous superclass switch place (*ASP*) .
t36, t41	Transitions related to the message processing unit (*MPU*) of *request-price*.
t37, t42	Transitions related to the message processing unit (*MPU*) of *accept-proposal*.
t38, t43	Transitions related to the message processing unit (*MPU*) of *reject-proposal*.
t39, t44	Transitions related to the asynchronous superclass switch place (*ASP*) .
t45 / t46	The transition related to the GSP of *Selling Agent_1* / *Selling Agent_2*.

To verify the correctness of our agent model, we utilize some key definitions for Petri net behavior properties as adapted from [18].

Definition 4.1 *Reachability*

In a Petri net *N* with initial marking M_0, denoted as *(N, M_0)*, a marking M_n is said to be *reachable* from a marking M_0 if there exists a sequence of firings that transforms M_0 to M_n. A *firing* or *occurrence sequence* is denoted by $\sigma = M_0\ t1\ M_1\ t2\ M_2 \ldots tn\ M_n$ or simply $\sigma = t1\ t2 \ldots tn$. In this case, M_n is reachable from M_0 by σ and we write $M_0\ [\sigma > M_n$.

Definition 4.2 *Boundedness*

A Petri net *(N, M_0)*, is said to be *k-bounded* or simply *bounded* if the number of tokens in each place does not exceed a finite number *k* for any marking reachable from M_0. A Petri net *(N, M_0)* is said to be *safe* if it is 1-bounded.

Definition 4.3 *Liveness*

A Petri net *(N, M_0)*, is said to be *live* if for any marking *M* that is reachable from M_0, it is possible to ultimately fire any transition of the net by progressing some further firing sequence.

Definition 4.4 *Reversibility*

A Petri net *(N, M_0)* is said to be *reversible* if, for each marking *M* that is reachable from the initial marking M_0, M_0 is reachable from *M*.

With our net model in Figure 8 as input, the INA tool produces the following results:

```
Computation of the reachability graph
States generated: 8193
Arcs generated: 29701

Dead states:
      484, 485,8189
Number of dead states found: 3
The net has dead reachable states.
The net is not live.
The net is not live and safe.
The net is not reversible (resetable).
The net is bounded.
```

```
The net is safe.
The following transitions are dead at the initial marking:
       7, 9, 14, 15, 16, 17, 20, 27, 28, 32, 33
The net has dead transitions at the initial marking.
```

The analysis shows that our net model is not live, and the dead reachable states indicate a deadlock. By tracing the firing sequence for those dead reachable states, we find that when there is a token in place *P29*, both the transitions *t34* and *t35* are enabled. At this time, if the transition *t35* fires, a token will be deposited into place *P30*. After firing transition *t40*, the token removed from place *P24*, by firing transition *t29*, will return to place *P24*, and this makes it possible to fire either transition *t27* or *t29* in a future state. However if the transition *t34* fires, instead of firing transition *t35*, there will be no tokens returned to place *P24*. So, transition *t27* and *t29* will be disabled forever, and a deadlock situation occurs. To correct this error, we need to modify the design of the planner module in Figure 3. The model modification is to add a new arc from transition *start_a_conversation* to place *syn*. Correspondingly, we add two new arcs in Figure 8: an arc from transition *t16* to place *P7*, and another arc from transition *t34* to place *P24*. After this correction, we can again evaluate the revised net model by using the INA tool. Now we obtain the following results:

```
Computation of the reachability graph
States generated: 262143
Arcs generated: 1540095

The net has no dead reachable states.
The net is bounded.
The net is safe.
The following transitions are dead at the initial marking:
       7, 9, 14, 15, 16, 17, 20, 28
The net has dead transitions at the initial marking.

Liveness test:
Warning: Liveness analysis refers to the net where all dead transitions
are ignored.
The net is live, if dead transitions are ignored.
The computed graph is strongly connected.
The net is reversible (resetable).
```

This automated analysis shows that our modified net model is *live*, ignoring, of course, any transitions that are dead in the initial marking. Thus, for any marking M that is reachable from M_0, it is possible to

ultimately fire any transition (except those dead transitions) of the net. Since the initial marking M_0 represents that there is no ongoing (active) conversations in the net, a marking *M* that is reachable from M_0, but where $M \neq M_0$, implies that there must be some conversations active in the net. By showing that our net model is live, we prove that under all circumstances (no matter if there are, or are not, any active conversations), it is possible to eventually perform any needed future communicative act. Consider the dead transitions *t7*, *t9*, *t14*, *t15*, *t16*, *t17* and *t20*. These imply that the decision-making units in the shopping subagent are disabled. The remaining dead transition, *t28*, implies that the primary subagent always makes decisions for the whole buying agent.

Our net model is *safe* because we have set the capacity of each place in our model to 1. A net model with capacity *k* ($k > 1$) for each place can be proved to be k-bounded in the same way. However, the state space may increase dramatically.

In addition, the analysis tells us that our net model is *reversible*, indicating that the initial marking M_0 can be reproduced (recall definition 4.4, given earlier). Since the initial marking M_0 represents that there are no ongoing (active) conversations in the net, the reversible property proves that every conversation in the net can be eventually completed.

4.3 Property Verification by Model Checking

To further prove additional behavioral properties of our revised net model, we use some model checking capabilities provided by the INA tool. Model checking is a technique in which the verification of a system is carried out by using a finite representation of its state space. Basic properties, such as an absence of deadlock or satisfaction of a state invariant (e.g., mutual exclusion), can be verified by checking individual states. More subtle properties, such as guarantee of progress, require checking for specific cycles in a graph representing the states and possible transitions between them. Properties to be checked are typically described by formulae in a branching time or linear time temporal logic [34][35].

The INA tool allows us to state properties in the form of CTL formulae [32][34]. Using this notation, we can specify and verify some key properties of our revised net model, such as concurrency, mutual exclusion, and proper inheritance behavior:

- *Concurrency*

The following formula says that, in the reachability graph of our revised net model, there exists a path that leads to a state in which all the places *P5*, *P13*, *P22* and *P28* are marked.

```
EF(P5 &(P13 &(P22 &P28)))     Result: The formula is TRUE
```

Result explanation: A TRUE result indicates that all the places *P5*, *P13*, *P22* and *P28* can be marked at the same time. From Figure 8 and Table 1, we see that incoming/outgoing messages are dispatched in these places. So the result implies that different messages can be dispatched in our net model concurrently.

- *Mutual Exclusion*

The following formula says that, in the reachability graph of our revised net model, there exists a path that leads to a state in which both places *P27* and *P30*, or both places *P29* and *P30*, are marked.

```
EF((P27 &P30) V (P29 &P30))    Result: The formula is FALSE
```

Result explanation: A FALSE result indicates that it is impossible to mark both places *P27* and *P30*, or both places *P29* and *P30*, at the same time. From Figure 8 and Table 1, we see that place *P27* represents any actions executed after decision-making, and place *P30* is used for updating the plan, goal and knowledge-base. Thus, this result guarantees that decisions can only be made upon the latest mental state, i.e., the latest values in plan, goal and knowledge-base modules. Similarly, the fact that *P29* and *P30* cannot be marked at the same time guarantees the requirement that the sensor can always capture the latest mental state.

- *Inheritance Mechanism (decision-making in subagent)*

The following formula says that, in the reachability graph of our revised net model, *P12*, *P14* and *P15* are not marked in any state on all paths.

```
AG(-P12 &(-P14 &-P15))        Result: The formula is TRUE
```

Result explanation: A TRUE result indicates that places *P12*, *P14* and *P15* are not marked under any circumstance. From Figure 8 and Table 1, we see that *P12*, *P14* and *P15* belong to decision-making units in the shopping subagent. As we stated earlier, all decision-making mechanisms in subagents should be disabled, with all decision-makings for an agent being achieved by the primary subagent. So, the result implies a desirable feature of the inheritance mechanism in our net model.

- *Inheritance Mechanism (ASP message forwarding I)*

The following formula says that, in the reachability graph of our revised net model, *P26* or *P34* are always marked before *P5* or *P6* is marked.

```
A[(P26 VP34)B(P5 VP6)]        Result: The formula is TRUE
```

Result explanation: A TRUE result indicates that neither place *P5* nor *P6* can become marked before the place *P26* or *P34* is marked. From Figure 8 and Table 1, we see that place *P26* and *P34* represent *ASP* places, and *P5* and *P6* represent the message dispatching units. The result implies that messages will never be dispatched in a shopping subagent unless a *MPU* is not found in the primary buying subagent, in which case, either the *ASP* place *P26* or *P34* will be marked.

- *Inheritance Mechanism (ASP message forwarding II)*

The following formula says that, in the reachability graph of our revised net model, *P26* (*P34*) is always marked before *P5* (*P6*) is marked.

```
A[P26 BP5]VA[P34 BP6]         Result: The formula is FALSE
```

Result explanation: We expect that for every incoming (outgoing) message, if it is not found in the primary buying subagent, it will be forwarded to the shopping agent, and dispatched into a *MPU* of the incoming (outgoing) message section. However, the FALSE result indicates that our net model does not work as we have expected. By looking into the generic agent model, we can observe that when we created the net model in Figure 8, we simplified all message tokens as ordinary tokens, i.e., black tokens. This simplification makes it possible for an incoming (outgoing) message to be dispatched into an outgoing (incoming) message section. Therefore, a message might be processed by a *MPU* that is not the desired one. To solve this problem, we may use colored tokens, instead of ordinary tokens, to represent message tokens, and attach guards to transitions. However, in this paper, by using ordinary place/transition net (not a colored net), we obtain a simplified model that is sufficient to illustrate our key concepts.

5. Conclusion and Future Work

One of the most rapidly growing areas of interest for distributed computing is that of distributed agent systems. Although there are several implementations of multi-agent systems available, formal frameworks for such systems are few. Formal methods in multi-agent system specification and design can help to ensure robust and reliable products.

In this paper, we introduced an agent-oriented model rooted in the Petri net formalism, which provides a foundation that is mature in terms of both existing theory and tool support. An example of an agent family in electronic commerce was used to illustrate the modeling approach. Models for a shopping agent, selling agent, buying agent and retailer agent were presented, with emphasis on the characteristics of being autonomous, reactive and internally-motivated. Our agent-oriented models also provide a clean interface between agents, and agents may communicate with each other by using contract net protocols. By the example of registration-negotiation protocol between shopping agents and facilitator agents, and the example of a price-negotiation protocol between shopping agents and buying agents, we illustrated how to create agent models and how to reuse functionality defined in an agent superclass. We also discussed how to verify liveness properties of our net model by using an existing Petri net tool, the INA tool. The value of such an automated analysis capability was demonstrated by detection of a deadlock situation due to a design error. The revised model was then proved to be both *live* and *reversible*. Finally, some model checking techniques were used to prove some additional behavioral properties for our model, such as concurrency, mutual exclusion, and correctness of the inheritance mechanism. Although we proved some key behavioral properties of our agent model, our formal method approach is also of value in creating a clear understanding of the structure of an agent, which can increase confidence in the correctness of a particular multi-agent system design. Also, in producing a more detailed design, where the abstract transitions in the planner module are refined, we may again use Petri net tools to capture further design errors.

For our future work, we will consider the refinements of the *Goal*, *Plan*, *Knowledge-base* and *Environment* modules. Also, the abstract transitions defined in the *Planner* module, i.e., *make_decision*, *sensor* and *update*, can be refined into correct sub-nets that capture action sequences specific to those activities. This work will provide a bridge to other work concerned with such agent activities [36][37][38]. We will also look further into issues like deadlock avoidance and state exploration problems in the agent-oriented design and verification processes.

References

[1] N. R. Jennings, "An Agent-Based Approach for Building Complex Software Systems," *Communications of the ACM*, vol. 44, no. 4, April 2001, pp. 35-41.

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

[3] J. Yin, M. S. Miller, T. R. Ioerger, J. Yen, and R. A. Volz, "A Knowledge-Based Approach for Designing Intelligent Team Training Systems," *Proceedings of the Fourth International Conference on Autonomous Agents* (Agents 2000), Barcelona, Spain, June 2000, pp. 427-434.

[4] S. Green, L. Hurst, B. Nangle, P. Cunningham, F. Somers, and R. Evans, "Software Agents: A Review," *Intelligent Agents Group (IAG) report TCD-CS-1997-06*, Trinity College Dublin, May 1997.

[5] T. J. Rogers, R. Ross, V. S. Subrahmanian, "IMPACT: A System for Building Agent Applications," *Journal of Intelligent Information Systems* (JIIS), vol. 14, no. 2-3, 2000, pp. 95-113.

[6] E. A. Kendall, "Role Modeling for Agent System Analysis, Design, and Implementation," *IEEE Concurrency*, April-June, 2000, pp. 34-41.

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

[8] M. Luck and M. d'Inverno, "A Formal Framework for Agency and Autonomy," *Proceedings of the First International Conference on Multi-Agent Systems* (ICMAS-95), AAAI Press / MIT Press, 1995, pp. 254-260.

[9] M. Fisher, "Representing and Executing Agent-Based Systems," *Intelligent Agents -- Proceedings of the International Workshop on Agent Theories, Architectures, and Languages*, M. Wooldridge and N. Jennings, eds., Lecture Notes in Computer Science, vol. 890, Springer-Verlag, 1995, pp. 307-323.

[10] F.M.T. Brazier, B. Dunin Keplicz, N. R. Jennings, and J. Treur, "DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework," *International Journal of Cooperative Information Systems*, vol. 6, Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, M. Huhns and M. Singh, eds., 1997, pp. 67-94.

[11] H. Zhu, "SLABS: A Formal Specification Language for Agent-Based Systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 5, 2001, pp. 529-558.

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

[13] V. Hilaire, A. Koukam, P. Gruer, and J.-P. Müller, "Formal Specification and Prototyping of Multi-agent Systems," *Engineering Societies in the Agent World, First International Workshop* (ESAW 2000), Berlin, Germany, August 2000, A. Omicini, R. Tolksdorf, and F. Zambonelli, eds., Lecture Notes in Computer Science, vol. 1972, Springer-Verlag, 2000, pp. 114-127.

[14] E. Huang and T. Elrad, "Reflective Decision Control for Autonomous Distributed Objects," *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems* (ISADS), March 2001, pp. 212-219.

[15] 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*, vol. 3, no.3, 2000, pp. 285-312.

[16] H. Xu and S. M. Shatz, "An Agent-based Petri Net Model with Application to Seller/Buyer Design in Electronic Commerce," *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems* (ISADS), March 2001, Dallas, Texas, pp. 11-18.

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

[18] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, April 1989, pp. 541-580.

[19] J. Yen, J. Yin, T. Ioerger, M. Miller, D. Xu, and R. Volz, "CAST: Collaborating Agents for Simulating Teamwork," *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (IJCAI), August 2001, Seattle, Washington, pp. 1135-1144.

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

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

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

[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 8-11, 2000, Nashville, Tennessee, USA, pp. 3128-3133.

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

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

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

[27] N. R. Jennings, K. Sycara and M. Wooldridge, "A Roadmap of Agent Research and Development," *International Journal of Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, 1998, pp. 7-38.

[28] L. Crnogorac, A. S. Rao, and K. Ramamohanarao, "Analysis of Inheritance Mechanisms in Agent-Oriented Programming," *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (IJCAI –97), 1997, pp. 647-654.

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

[30] F. Bergenti and A. Poggi, "Exploiting UML in the Design of Multi-Agent Systems," *Engineering Societies in the Agent World*, *First International Workshop* (ESAW 2000), Berlin, Germany, August 2000, A. Omicini, R. Tolksdorf, and F. Zambonelli, eds., Lecture Notes in Computer Science, vol. 1972, Springer-Verlag, 2000, pp. 106-113.

[31] H. Yim, K. Cho, J. Kim, and S. Park, "Architecture-Centric Object-Oriented Design Method for Multi-Agent Systems," *Proceedings of the Fourth International Conference on Multi-Agent Systems* (ICMAS 2000), Boston, Massachusetts, July 2000.

[32] S. Roch and P. H. Starke, *INA:Integrated Net Analyzer*, Version 2.2, Humboldt-Universität zu Berlin, Institut für Informatik, April 1999.

[33] S. M. Shatz, S. Tu, T. Murata, and S. Duri, "An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 12, December 1996, pp. 1307-1322.

[34] E. M. Clarke, E. A. Emerson and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, 1986, pp. 244-263.

[35] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, December 1996, pp. 626-643.

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

[37] T. Murata, V. S. Subrahmanian, and T. Wakayama, "A Petri Net Model for Reasoning in the Presence of Inconsistency", *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 3, September 1991, pp. 281-292.

[38] T. Murata, P. C. Nelson, and J. Yim, "A Predicate-Transition Net Model for Multiple Agent Planning," *Information Sciences*, 57-58, 1991, pp. 361-384.