Formal Methods in Agent-Oriented Design and Analysis

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

Intelligent 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. The mission of this proposed chapter is to describe an approach for using a formal model in the design of agents. The approach is based on G-nets, which are a type of Petri net defined to support system modeling in terms of a set of independent and loosely-coupled modules. We customize the basic G-net model to define a so-called "agent-oriented G-net" that can serve as a generic model for agent design. To illustrate our formal modeling technique for multi-agent systems, an example of an agent family in electronic commerce is provided. Finally, we discuss our future research plans.

1. An Introduction to Formal Methods for Agent Design

This section of the proposed chapter will introduce the use of formal methods for agent design and highlight related works. The following is a condensed version of this section.

Intelligent agents can be considered as active objects, or objects with mental states [1]. However, intelligent agents are quite different from objects in terms of communication mechanisms and decision-making capabilities. As a result, the object-oriented methodologies are not quite suitable for agent modeling. Especially, they do not directly support asynchronous message-passing and autonomous behavior modeling. Therefore, agent-oriented methodologies are proposed to provide guidelines for agent specification and design. Examples of such work are the AAII methodologies [3] and the Gaia methodologies [11]. Both of these two agent-oriented methodologies are extensions of object-oriented methodologies [12].

Although there are many efforts aimed at developing agent-based systems, there is sparse research on formal specification and design of such systems. As 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 [2]. Previous work on formal modeling agent systems includes the DESIRE model [4], the dMARS model [5], and agent models based on Petri nets [6]. The DESIRE model provides a compositional framework for modeling agents, and the dMARS model is based on Procedure Reasoning System (PRS), which supports formal reasoning. A typical example of agent models based on Petri nets is Moldt and Wienberg's work, in which they proposed a multi-agent system model based on colored Petri nets. The weakness of these models is that they do not explicitly model agent communications, which is one of the key issues for intelligent agent modeling. Another problem is that they do not address the issue of inheritance. Therefore, in our perspective, these models are agent-based rather than agent-oriented.

Unlike the previous work, our proposed agent model supports protocol-based agent communication. Meanwhile, by introducing inheritance mechanisms, and separating the transitional object-oriented features and reasoning mechanisms in our proposed agent-oriented model, we show that reuse can be achieved in terms of functional units defined in an agent model. Furthermore, since we uniformly use net-based formalism for agent modeling, our formal agent designs could be analyzed by using existing Petri net tools.

2. A Net-based Approach for Agent-Oriented Design

This section of the proposed chapter will discuss our Petri net based approach for agent design. The following is a condensed version of this section.

2.1 The 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 [9]. G-nets are an object-based extension of Petri nets [8], which is a graphically defined model for synchronous concurrent systems. A G-net system is composed of a number of G-nets, each of them representing a self-contained module or object. A G-net is composed of two parts: a special place called *Generic Switch Place (GSP)* and an *Internal Structure (IS)*. The *GSP* provides the abstraction of the module, and serves as the only interface between the G-net and other modules. The *IS*, a modified Petri net, represents the detailed design of the module. 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.

The G-net model supports the *Client-Server* paradigm, and it is suitable for object-based design, however, it is not sufficient for agent design because the G-net model does not directly support the following features. First, intelligent agents in multi-agent systems are usually developed by different vendors independently, therefore it is essential for them to have a common communication language and to follow common protocols. Second, the underlying agent communication model is usually asynchronous, and an agent may decide whether to perform actions requested by some other agents. 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.

2.2 A Framework for Agent-Oriented Modeling

To support agent-oriented design, we first need to extend a G-net to support class modeling [14][15]. This can be simply done by interpreting a G-net as a model of agent class; meanwhile we need to define the instantiation of a G-net with the following two steps: to generate a unique agent identifier *G.Aid*, and to initialize the mental state of the resulting agent object. 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. The template for an agent-oriented G-net model is shown in Figure 1. The *Goal*, *Plan* and *Knowledge-base* module are based on the BDI agent model [3], while the *Environment* module is an abstract model of the environment, i.e., the model of the outside world of an agent. The *Planner* module represents the heart of an agent that may decide to ignore an incoming message, to start a new conversation, or to continue with the current conversation. In the *Planner* module, committed plans are achieved, and the *Goal*, *Plan* and *Knowledge-base* modules of an agent are updated after each communicative act [10] or if the environment changes. The *internal structure (IS)* of an agent-oriented 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 correspond to a subset of communicative acts. Each *MPU*, labeled as *action_i* in Figure 1, is used to process incoming/outgoing messages, and may use *ISP*-type modeling for calls to methods defined in its *private utility* section.

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 [2]. In particular, these messages must satisfy standardized communicative (speech) acts, which define the type and the content of the message (e.g., the FIPA agent communication language, or KQML) [10]. Note that in Figure 1, each named *MPU action_i* refers to a communicative act, thus our agent-oriented model supports an agent communication interface. In addition, agents analyze these messages and can decide whether to execute the requested action. As we stated before, agent communications are typically based on asynchronous message passing. Since asynchronous message passing is more

fundamental than synchronous message passing, it is useful for us to introduce a new mechanism, called *Message-passing Switch Place (MSP)*, to directly support asynchronous message passing.

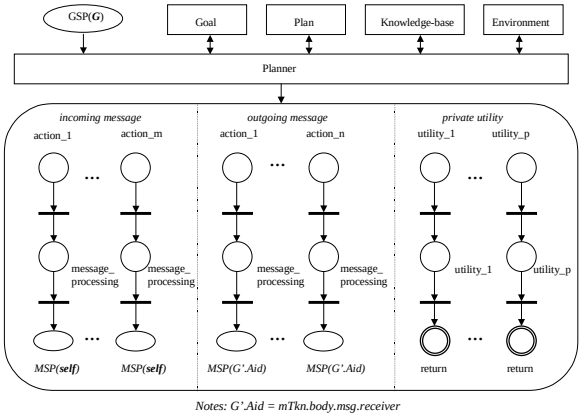

Figure 1. A generic agent-oriented G-net model

A template of the *Planner* module is shown in Figure 2[1]. The modules *Goal*, *Plan*, *Knowledge-base* and *Environment* are represented as four special places (denoted by double ellipses in Figure 2), 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 2 (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. We will give detailed description of the *Planner* module in our proposed chapter.

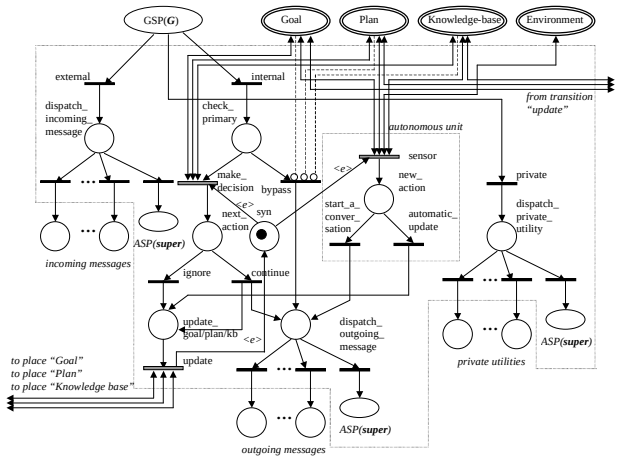

Figure 2. A template for the planner module (initial design)

To support agent-oriented design, we also need to incorporate some inheritance modeling capabilities [16]. 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 [7]. However, we believe that since inheritance happens at the class level, an agent subclass may be initialized with an agent superclass's initial mental

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

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. For simplicity, we assume that an instance of an agent subclass (i.e., a subclass agent) always uses its own reasoning mechanisms, and thus the reasoning mechanisms in the agent superclass should be disabled in some way. On the other hand, to reuse the services (i.e., *MPUs* and *methods*) defined in a subagent (i.e., a part of the agent object that corresponds to the agent superclass model), we need to introduce a new mechanism called *Asynchronous Superclass switch Place (ASP)*. An *ASP* (denoted by an ellipsis in Figure 2) 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. 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.

2.3 Examples of Agent-Oriented Design

Consider an agent family in an electronic marketplace domain. Figure 3 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. Instances of both the buying agent class and selling agent class, as subclasses of an shopping agent class, 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 class is defined as a subclass of both the buying agent class and the selling agent class, and a customer/auctioneer agent class is defined as a subclass of a buying/selling agent class.

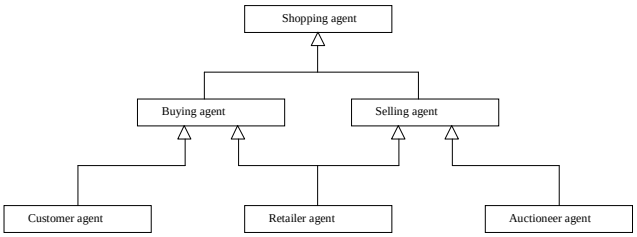

Figure 3. The class hierarchy diagram of agents in an electronic marketplace

Based on the communicative acts (e.g., request-registration, refuse, etc.) needed for the contract net protocol between the shopping agent and the facilitator agent, we may design the shopping agent class and the facilitator agent class by using our agent-oriented G-net model. Similarly, based on the communicative acts (e.g., request-price, propose, etc.) needed for the contract net protocol between the selling agent and the buying agent, we may also design the selling and buying agent class.

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, 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. We will show how the reuse of these functional units can be achieved in our proposed chapter.

2.4 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. A correct design of agent 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. Here, we will use a Petri tool, called INA (Integrated Net Analyzer) [13], to analyze and verify our agent models.

The interaction of one buying agent and two selling agents can be modeled and folded into ordinary Petri nets. By inputting our net model into the INA tool, the result 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 the deadlock is due to a missing arc from transition *start_a_conversation* to place *syn* (Figure 2). After the correction, we can again evaluate the revised net model by using the INA tool. At this time, the result shows that our net model is live.

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. The INA tool allows us to state properties in the form of CTL formulae [13]. 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.

3. Future Research Plans

In addition to proving key behavioral properties of our agent model by using existing Petri net theories and existing Petri net tools, our formal method approach is also of value in creating a clear understanding of the structure of an intelligent agent, and ensuring the correctness of further detailed design for a particular multi-agent system. For our future research, we plan to implement an agent development toolkit to help software engineers to design agents. In this toolkit, while a detailed design is produced, for instance, the abstract transitions in the planner module is refined, we may again use Petri net tools to capture further design errors. Furthermore, we will also try to use our agent framework for mobile agent design. We will model both mobile agents and hostile agents, and to study different forms of attacks.

References

[1] Yoav Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, 60(1): 51-92, March 1993.

[2] 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.

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

[4] Brazier, F.M.T., Dunin Keplicz, B., Jennings, N., and Treur, J., "DESIRE: Modeling 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.

[5] M. d'Inverno, D. Kinny, M. Luck and M. Wooldridge, "A Formal Specification of dMARS," In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, Singh, Rao and Wooldridge (eds.), Lecture Notes in Artificial Intelligence, 1365, 155-176, Springer-Verlag, 1998.

[6] Daniel Moldt and Frank Wienberg, "Multi-Agent-Systems based on Coloured Petri Nets," *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, Toulouse, June 23-27, 1997

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

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

[9] 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.

[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] 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), 2000, pp. 285-312.

[12] M. Wooldridge and P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art," To appear in the *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 2001.

[13] S. Roch and P. H. Starke, *INA:Integrated Net Analyzer*, Version 2.2, Humboldt-Universität zu Berlin, Institut für Informatik, April 1999. http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina/

[14] 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)*, October 2000, Nashville, Tennessee, USA, pp. 3128-3133.

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

[16] H. Xu and S. M. Shatz, "A Framework for Modeling Agent-Oriented Software," To appear in the *Proc. of the 21st International Conference on Distributed Computing Systems (ICDCS)*, April 2001, Phoenix, Arizona.