Presented on Feb. 26 (Monday), 2001 at 3:30 PM in room 1027 SEO

FROM OBJECT TO AGENT: AN APPROACH TO USING FORMAL METHODS IN SOFTWARE DESIGN AND ANALYSIS

BY

HAIPING XU

B.S., Zhejiang University, Hangzhou, China, 1989

M.S., Zhejiang University, Hangzhou, China, 1992

M.S., Wright State University, Dayton, OH, 1998

PH.D. THESIS PROPOSAL

Submitted as partial fulfillment of the requirements

For the degree of Doctor of Philosophy in Computer Science

In the Graduate College of the

University of Illinois at Chicago, 2001

Chicago, Illinois

TABLE OF CONTENTS

CHAPTER

1. INTRODUCTION
   1.1 Background
   1.2 Related Work
      1.2.1 Formal Methods in Object-Oriented Design
      1.2.2 Agent-Oriented Methodology and Formal Approaches
   1.3 Contributions of Our Work

2. INHERITANCE MODELING IN OBJECT-ORIENTED DESIGN
   2.1 Introduction
   2.2 G-net Model Background
   2.3 Extending G-nets for Class Modeling
   2.4 Extending G-net Models to Support Inheritance
   2.5 Modeling Different Forms of Inheritance
   2.6 Modeling Inheritance Anomaly Problem
   2.7 Discussion

3. AN AGENT-BASED G-NET MODEL
   3.1 Introduction
   3.2 Agent-based G-net Model
   3.3 Selling and Buying Agent Design
   3.4 Verifying Agent-Based G-net Models
   3.5 Discussion

4. A FRAMEWORK FOR MODELING AGENT-ORIENTED SOFTWARE
   4.1 Introduction
   4.2 An Agent-Oriented Model
      4.2.1 An Architecture for Agent-Oriented Modeling
      4.2.2 Inheritance Modeling in Agent-Oriented Design
   4.3 Examples of Agent-Oriented Design

4.3.1 A Hierarchy of Agents in an Electronic Marketplace
4.3.2 Modeling Agents in an Electronic Marketplace
4.4 Handling Multiple Inheritance in Agent-Oriented Models
4.5 Discussion

5. ANALYSIS OF AGENT-ORIENTED MODELS
5.1 Introduction
5.2 A Simplified Petri Net Model for a Buying Agent and Two Selling Agents
5.3 Deadlock Detection and Redesign of Agent-Oriented Models
5.4 Property Verification by Using Model Checking
5.5 Discussion

6. FUTURE RESEARCH PLANS
6.1 Introduction
6.2 A Unified Model for Object-Oriented and Agent-Oriented Design
6.3 Extending Agent-Oriented G-net Model for Mobile Agent Design
6.4 Security Issues in Mobile Agent Design

BIBLIOGRAPHY

PUBLICATIONS OF THE AUTHOR

CURRICULUM VITAE

Chapter 1

Introduction

1.1 Background

The development of software systems starts with two main activities, namely software requirements analysis and software design [Sommervile 1995][Pressman 1997]. The purpose of software requirements analysis is to understand the problem thoroughly and reduce potential errors caused from incomplete or ambiguous requirements. The product of the requirements analysis activity is a software requirements specification, which serves as a contract between the customers and the software designers. The purpose of the software design is to follow the software requirements specification and to depict the overall structure of a system by decomposing the system into its logical components. The design activity translates requirements into a representation of the software that can be assessed for quality before coding begins. Like software requirements, the product of the design activity is a design specification, which serves as a contract between the software designers and the programmers.

There are two ways to achieve the purposes of these two activities. One is to specify and analyze systems formally, and the other is to describe and model systems naturally. When specifying, modeling and analyzing the behavior of a critical and complex system, we usually choose a specification language that can formally depict the properties of the system. This is because formal languages can be used to describe system properties clearly, precisely and in detail, and to enable design and analysis techniques to evolve and operate in a systematic manner. Since the 1960's, researchers have been working on formal modeling of critical and complex systems such as concurrent and distributed systems. Among these formal methods, Petri nets [Murata 1989], as a graphical and mathematical modeling tool, are well recognized and widely used in various application domains because of its simplicity and flexibility to depict the dynamic system behaviors, and its strong expressive and analytic power for system modeling. Although Petri nets have been successfully used for system modeling and analysis in various domains, formal methods are still not a popular way for most of the industry/commercial software development. Therefore, many Petri net researchers have devoted efforts to enhance/extend the theory and techniques of Petri nets, including high-

level Petri nets such as CPN (Colored Petri Nets) [Jensen 1992], and tried to build a bridge between formal methods and industry/commercial software development.

Meanwhile, in the industry, there are several transitions of software engineering paradigms during the last few decades. In the seventies, structured programming was the dominant approach to software development. Along with it, software engineering technologies were developed in order to ease and formalize the system development life cycle: from planning, through analysis and design, and finally to system construction, transition and maintenance. In the eighties, object-oriented (OO) languages experienced a rise in popularity, bring with it new concepts such as data encapsulation, inheritance, messaging and polymorphism. By the end of the eighties and beginning of the nineties, a jungle of modeling approaches grew to support the OO market. For instance, the Unified Modeling Language (UML) [Rational 1997], which unifies three popular approaches to OO modeling: the Booch method [Booch 1994], OMT [Rumbaugh *et al.* 1991] and OOSE [Jacobson *et al.* 1992], becomes the most popular modeling language for object-oriented software systems. Although the object-oriented paradigm has achieved a considerable degree of maturity, researchers continually strive for more efficient and powerful software engineering techniques, especially as solutions for even more demanding applications. The emergence of agent techniques is one of the examples of such efforts. In the last few years, the agent research community has made substantial progress in proving a theoretical and practical understanding of many aspects of agents and multi-agent systems [Green *et al.* 1997][Jennings *et al.* 1998]. Agents are being advocated as a next generation model for engineering complex, distributed systems [Jennings 2000]. Yet despite of this intense interest, the concepts of agent-oriented paradigm are still not matured, and the methodology, especially the techniques for agent modeling in practical use, is yet to be researched.

Although there have been many efforts on object and agent modeling, to provide a framework for object-oriented design and agent-oriented design is still a big challenge. Due to the lack of formalisms for practical complex software design, we aim to use and extend a type of high-level Petri nets, called G-nets [Perkusich and de Figueiredo 1997], to model objects and agents in object-oriented design and agent-oriented design respectively. Our proposed formalism has the advantage of being easy to understand, easy to use, and practically it is helpful for designers to design complex software systems, and to use existing Petri net tools to analyze its correctness and to verify its behavior properties such as liveness. In addition, since we view an agent as an extension of an object, i.e., an active object [Shoham 1993], our object models and agent models maybe combined to provide a unified framework for complex software design, especially for Internet applications such as electronic commerce.

1.2 Related Work

1.2.1 Formal Methods in Object-Oriented Design

The concepts of object-oriented paradigm, such as encapsulation and inheritance, have been widely used in system modeling because they allow us to describe a system easily, intuitively and naturally [Rumbaugh *et al.* 1991][Booch 1994][Jacobson *et al.* 1992][Eliens 1995]. With the increasing complexity of nowadays software systems, object-oriented software designers began to understand the usefulness of formal methods. Along with this trend, object-oriented formal methods became one of hot research issues for the last few years. Many researchers have suggested object-oriented formal methods, such as OPN (Object Petri Nets) [Bastide 1995], VDM++ [Lano 1995] and Object-Z [Stepney *et al.* 1992]. Among them, the research on the OPN methods have been actively studied to extend the Petri nets formalism to various forms of object Petri nets, such as OBJSA [Battiston *et al.* 1988], LOOPN++ [Lakos and Keen 1994], CO-OPN/2 [Biberstein *et al.* 1997] and G-nets [Perkusich and de Figueiredo 1997]. Although the results of such studies are promising, these formalisms do not fully support all the major concepts of object-oriented methodology. We now give a brief description of these formalisms.

OBJSA nets, suggested by E. Battiston, define a class of algebraic nets that are extended with modularity features. Their name reflects that they integrate Superposed Automata nets and the algebraic specification language OBJ [Battiston *et al.* 1988][Battiston *et al.* 1995]. OBJSA nets correspond to the semantics model described by algebraic notations, and CLOWN (CLass Orientation With Nets) is a notation developed on the top of OBJSA nets with object-oriented features added [Battiston *et al.* 1996]. CLOWN attributes can be declared as constant (**const**) or variable (**var**), and all the actions that an object can execute are specified by the “method” clauses. In addition, the “interface” clause defines the interaction between a CLOWN object and some other objects, and the inheritance features are extended by the “inherits” clause.

In CLOWN, the data structure of a class is defined by algebraic notations, and the control structure of the class is defined by a class net. Objects in CLOWN are represented as distinguished individual tokens flowing in the corresponding class net. CLOWN does not take the full advantage of this formalism because only the control structure of a system is modeled by Petri nets. Since object-oriented features in CLOWN are not captured at the net level, there are limitations in using existing Petri net tools for system analysis.

O. Biberstein suggests the specification language, called CO-OPN/2 (Concurrent Object-Oriented Petri Nets) [Biberstein *et al.* 1996, Biberstein *et al.* 1997], which is designed to specify and model large scale concurrent systems. The class definition in CO-OPN/2 consists of two parts: “Signature” part is to describe the interface with other classes, and “Body” part is to describe the internal behaviors and operations of a class. The specification method of CO-OPN/2 is similar with that of CLOWN, but the differences are that CO-OPN/2 supports abstract data type in order to reuse its type defined in other classes, and the methods declared in “Signature” part is used as interface transition. The problem of CO-OPN/2 is that the unfolding

mechanism for a CO-OPN/2 specification is not provided, therefore the analysis and simulation method of CO-OPN/2 is unclear.

C. Lakos proposes a class of object-oriented Petri nets, called LOOPN++ (Language for Object-Oriented Petri Nets) [Lakos and Keen 1994, Lakos 1995a, Lakos 1995b]. LOOPN++ uses the text-based grammar to specify systems. In the specification of LOOPN++, the class definition consists of three parts: "Fields" to define data, "Function" to describe expression with parameters and operation, and "Actions" to represent the behavior of a system. The "Fields" part is a declaration of a token in Petri nets, and is used to represent the states of places. The "Functions" and "Actions" part together represent the transitions of Petri nets.

One of the major characteristics of LOOPN++ is the feature for "super places" and "super transitions", used to represent the nesting structure of nets. It becomes a base to support the abstraction of nets. The super place and super transition can be defined by labeling at the corresponding place and transition of nets with the name of an external object. With this feature, "Parent" phrases can be used to represent (multiple) inheritance of classes. Regardless of continuous research on LOOPN++, it has some deficiencies in fully supporting the object-oriented concepts. First, LOOPN++ does not fully reflect the actual concepts of objects because the nets include the global control structure of systems, and tokens are only passive data types [Lakos 1997]. Second, LOOPN++ tries to represent the abstraction by the feature of fusion only, but is not sufficient for abstraction of functional behavior and states. Third, LOOPN++ provides the "Export" phrase, but the message passing mechanism for the interaction among objects is not supported. Finally, LOOPN++ is well applied in the object-oriented software development methodology of Shlaer-Mellor [Lakos and Keen 1994], but not in the methodology of OMT/UML, which is one of the most popular approaches nowadays.

G-nets [Perkusich and de Figueiredo1997][Deng *et al.* 1993] support the concepts of objects better than in CO-OPN/2 or LOOPN++, at least in our concerns. As one form of high-level Petri nets, G-nets are based on the concept of modules corresponding to objects. There are two separate parts to describe the net structure of an object in G-nets. One is called *GSP* (Generic Switch Place), which contains the name of an object, the definition of attributes and methods, and initial marking of the net. The other one is called the *IS* (Internal Structure), which describes the behaviors of methods with a variant of Petri nets. There are special places in the nets, such as *ISP* (Instantiated Switching Place) to make a method call and *GP* (Goal Place) to end a method execution. These features can be unfolded into Pr/T nets.

A fascinating feature of G-nets is its support for encapsulation of objects, message passing for object interactions, and low coupling between objects. The use of the unique identifier for an object makes it possible to represent recursive method calls. Also, the mechanism for method call in G-nets is quite suitable for modeling client-server systems. Although G-nets are useful for object modeling and the

structure of a G-net is similar with that of an object in OMT/UML, it does not support inheritance mechanism. In addition, it is difficult to represent the abstraction hierarchy with net elements of G-nets.

The above object models are widely referenced and compared among high-level object-oriented Petri nets. Other similar research includes: OPNets by Lee [Lee and Park 1993], which are focused on the decoupling of inter-object communication knowledge and the seperation of synchronization constraints from the internal structure of objects; OCoNs (Object Coordination Nets) by Giese [Giese *et al.* 1998] are to describe the coordination of the behavior of a class on a service. Although these formalisms support sufficiently the basic concepts of objects such as encapsulation and modularization, they do not incorporate the concepts of abstraction and/or inheritance, and they do not clearly suggest the analysis or simulation methods.

1.2.2 Agent-Oriented Methodologies and Formal Approaches

Agent technology has received a great deal of attention in the past few years and, as a result, the industry is beginning to get interested in using this technology to develop its own products. In spite of the different developed agent theories, languages, architectures and the successful agent-based applications, very little work for specifying and design techniques to develop agent-based applications using agent technology has been done [Iglesias *et al.* 1998]. The role of agent-oriented methodologies is to assist all the phases of the life cycle of an agent-based application, including its management. A number of groups have reported on methodologies for agent design, touching on representational mechanisms as they support the methodology. Examples of such work are D. Kinny and his colleagues' BDI agent model [Kinny *et al.* 1996] and the Gaia methodology suggested by M. Wooldridge [Wooldridge *et al.* 2000].

Formal methods for agent modeling are mostly concerning about agent specification and agent design. Several formal approaches have tried to bridge the gap between formal theories and implementations. Though formal methods are not so easily scalable in practice, they are especially useful for verifying and analyzing critical applications, prototypes and complex cooperating systems. Traditional formal languages such as Z have been used [Luck et al. 1997], providing an elegant framework for describing an agent system at different levels of abstractions. Since there is no notion of time in Z, it is not quite suitable to specify agent interactions. Another approach has been the use of temporal modal logic [Wooldridge 1998] that allows the representation of dynamic aspects of the agents and a basis for specifying, implementing and verifying agent-based systems. The implementation of the specification can be done by directly executing the agent specification with a language such as *Concurrent Metatem* [Fisher and Wooldridge 1997] or by compiling the agent specification. The usage of formal methods for multi-agent specification such as DESIRE [Brazier *et al.* 1997] is an interesting alternative to be used as a detailed design language

in agent-oriented methodology. DESIRE (framework for Design and Specification of Interacting Reasoning components) proposes a component-based perspective based on task decomposition.

During the last few years, many efforts have been put on developing multi-agent systems, however there is a lack of research on formal specification and design of such systems [Iglesias *et al.* 1998][Rogers *et al.* 2000]. As the multi-agent technology begins to emerge as a viable solution for large-scale industrial and commercial applications, there is an increasing need to ensure that the systems being developed are robust, reliable and fit for purpose. The concept of agent-oriented methodology is still new, and there are different views on this issue [Iglesias *et al.* 1998][Jennings 2000]. In this proposal, we take the view that an agent is an extension of an object. Thus, based on the concepts of object-oriented methodology, we propose our agent-oriented design model, which is a nature approach for most of the object-oriented designers.

1.3 Contributions of Our Work

The work reported in this Ph.D. thesis proposal is aimed at proposing a technique for modeling and analyzing object-oriented and agent-oriented software systems. The concepts of agent-orientation are based on the concepts of object-orientation, but need to be extended with additional features, such as mechanisms for decision-making and asynchronous message passing. The major contributions of our work are listed as follows:

- Extended the original G-net model to support class modeling and inheritance modeling.
- Designed an agent-based G-net model, and proved properties related to *liveness*, *concurrency* and *effectiveness* for agent communication.
- Extended the agent-based G-net model to support inheritance modeling in agent-oriented design.
- Performed experiments with an existing Petri net tool to model and analyze agent-oriented software systems.

Chapter 2

Inheritance Modeling in Object-Oriented Design

2.1 Introduction

One of the key issues in object-oriented (OO) approach is inheritance. The inheritance mechanism allows users to specify a subclass that inherits features from some other class, i.e., its superclass. A subclass has the similar structure and behavior as the superclass, but in addition it may have some other features. As an essential concept of the OO approach, inheritance is both a cognitive tool to ease the understanding of complex systems and a technical support for software reuse and change. With the emergence of formalisms integrating the OO approach and the Petri net (PN) theory, the question arises how inheritance may be supported by such formalism, in order that they benefit from the advantages of this concept and existing Petri net tools. Inheritance has been originally introduced within the framework of data processing and sequential languages, while PNs are mainly concerned with the behavior of concurrent processes. Moreover, it has been pointed out that inheritance within concurrent OO languages entails the occurrence of many difficult problems such as the inheritance anomaly problem [Matsuoka and Yonezawa 1993]. Thus, to incorporate inheritance mechanism into Object Petri Net (OPN) has been viewed as a challenging task.

The concepts of inheritance define both the static features and dynamic behavior of a subclass object. The static feature specifies the structure of a subclass object, i.e., its methods and attributes; while the dynamic behavior of a subclass object refers to its state and its dynamic features such as overriding, dynamic binding and polymorphism [Drake 1998]. Most of the existing object-oriented Petri nets (OOPN) formalism, such as CLOWN, LOOPN++ and CO-OPN/2, fail to provide a uniform framework for class modeling and inheritance modeling in terms of these two features, and they usually use text-based formalism to incorporate inheritance into Petri nets. The problems of these approaches are that they do not take full advantage of the Petri net formalism, and therefore, we cannot use existing Petri net tools to verify the behavior properties of a subclass object in terms of inheritance. Little work has been done to model inheritance of dynamic behavior. Examples of such work are the concept of life-cycle inheritance proposed

by van der Aalst and Basten [Aalst and Basten 1997][Basten and Aalst 2000] and the SBOPN formalism with additional inheritance features suggested by Xie [Xie 2000]. However, these formliasms are either too theoretical to be used in practical software design, or too preliminary to cover all forms of inheritance, such as refinement inheritance [Drake 1998].

In this chapter, we propose a Petri net formalism, called extended G-nets, to model inheritance in concurrent object-oriented design. Based on the original G-net formalism [Perkusich and de Figueiredo 1997], we first extend G-nets into the so-called *standard G-nets* for class modeling, then we introduce new mechanisms to incorporate inheritance into standard G-net models. These new mechanisms are net-based, therefore it would be possible for us to translate our net models into other forms of Petri nets, such as Pr/T net, and use existing Petri net tools for behavior property analysis, e.g., to analyze the inheritance anomaly problem.

2.2 G-net Model Background

A widely accepted software engineering principle is that a system should be composed of a set of independent modules, where each module hides the internal details of its processing activities and modules communicate through well-defined interfaces. The G-net model provides strong support for this principle [Perkusich and de Figueiredo 1997][Deng *et al.* 1993]. 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 [Murata 1989]. But, as a general reminder, we note that Petri nets include three basic entities: place nodes (represented graphically by circles), transition nodes (represented graphically by solid bars), and directed arcs that can connect places to transitions or transitions to places. Furthermore, places can contain markers, called tokens, and tokens may move between place nodes by the "firing" of the associated transitions. The state of a Petri net refers to the distribution of tokens to place nodes at any particular point in time (this is sometimes called the marking of the net). We now proceed to discuss the basics of 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 detailed design of the module. An example of G-nets is shown in Figure 1. Here the G-net models represent two objects – a *Buyer* and a *Seller*. The generic switch places are represented by *GSP(Buyer)* and *GSP(Seller)* enclosed by ellipses, and the internal structures of these models are represented by round-cornered rectangles that contain the detailed design of four methods: *buyGoods()*, *askPrice()*, *returnPrice()* and *sellGoods()*. The

functionality of these methods are defined as follows: *buyGoods()* invokes the method *sellGoods()* defined in G-net *Seller* to buy some goods; *askPrice()* invokes the method *returnPrice()* defined in G-net *Seller* to get the price of some goods; *returnPrice()* is defined in G-net *Seller* to calculate the latest price for some goods and *sellGoods()* is defined in G-net *Seller* to wait for the payment, ship the goods and generate the invoice. A *GSP* of a G-net *G* contains a set of methods *G.MS* specifying the services or interfaces provided by the module, and a set of attributes, *G.AS*, which are state variables. In *G.IS*, the internal structure of G-net *G*, Petri net places represent primitives, while transitions, together with arcs, represent connections or relations among those primitives. The primitives may define local actions or method calls. Method calls are represented by special places called *Instantiated Switch Places (ISP)*. A primitive becomes *enabled* if it receives a token, and an enabled primitive can be executed. Given a G-net *G*, an *ISP* of *G* is a 2-tuple *(G'.Nid, mtd)*, where *G'* could be the same G-net *G* or some other G-net, *Nid* is a unique identifier of G-net *G'*, and $mtd \in G'.MS$. Each *ISP(G'.Nid, mtd)* denotes a method call *mtd()* to G-net *G'*. An example *ISP* (denoted as an ellipsis in Figure 1) is shown in the method *askPrice()* defined in G-net *Buyer*, where the method *askPrice()* makes a method call *returnPrice()* to the G-net *Seller* to query about the price for some goods. Note that we have highlighted this call in Figure 1 by the dashed-arc, but such an arc is not actually a part of the static structure of G-net models. In addition, we have omitted all function parameters for simplicity.

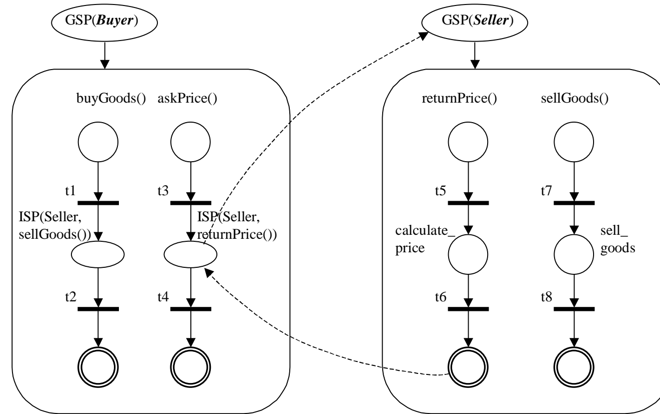


Figure 1. G-Net model of buyer and seller objects

2.3 Extending G-nets for Class Modeling

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 [Xu and Shatz 2000], we have extended the G-net model to support class modeling. The idea of this extension is to generate a unique object identifier, *G.Oid*, and initialize the state variables when a G-net object is instantiated from a G-net *G*.

An *ISP* method invocation is no longer represented as the 2-tuple *(G'.Nid, mtd)*, instead it is the 2-tuple *(G'.Oid, mtd)*, where different object identifiers could be associated with the same G-net class model.

The token movement in a G-net object is similar to that of original G-nets [Perkusich and de Figueiredo 1997]. A token *tkn* is a triple *(seq, sc, mtd)*, where *seq* is the propagation sequence of the token, $sc \in$ {**before**, **after**} is the status color of the token and *mtd* is a triple *(mtd_name, para_list, result)*. For ordinary places, tokens are removed from input places and deposited into output places by firing transitions. However, for the special *ISP* places, the output transitions do not fire in the usual way. Recall that marking an *ISP* place corresponds to making a method call. So, whenever a method call is made to a G-net object, the token deposited in the *ISP* has the status of **before**. This prevents the enabling of associated output transitions. Instead the token is "processed" (by attaching information for the method call), and then removed from the *ISP*. Then an identical token is deposited into the *GSP* of the called G-net object. 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.

More specifically, when a G-net object *G_obj* with *G.Oid* makes a method call *ISP(G'.Oid, mtd(para_list))* in its thread/process with *G.Pid*, the procedure for updating a G-net token *gTkn* is as follows:

1. Call_before: *gTkn.seq* ← *gTkn.seq* + <*G.Oid, G.Pid, mtd*>; *gTkn.msg* ← *(mtd, para_list, NULL)*; *gTkn.sc* ← **before**.
2. Transfer the *gTkn* token to the *GSP* place of the called G-net object with *G'.Oid*.
3. Wait for the result to be stored in *gTkn.msg.result*, and the *gTkn* token to be returned.
4. Call_after: *gTkn.seq* ← *gTkn.seq* – *LAST(gTkn.seq)*; *gTkn.sc* ← **after**.

We call a G-net model that supports class modeling a *standard* G-net model. We now provide a few key definitions for our standard G-net models.

Definition 2.1 *G-net system*

A *G-net system (GNS)* is a triple GNS = (INS, GC, GO), where INS is a set of initialization statements used to instantiate G-nets as G-net objects; GC is a set of G-nets which are used to define classes; and GO is a set of G-net objects which are instances of G-nets.

Definition 2.2 *G-net*

A *G-net* is a 2-tuple G = (GSP, IS), where GSP is a *Generic Switch Place (GSP)* providing an abstraction for the G-net; and IS is the *Internal Structure*, which is a set of modified Pr/T nets. A G-net is an abstract of a set of similarly G-net objects, and it can be used to model a class.

Definition 2.3 *G-net object*

A *G-net object* is an instantiated G-net with a unique object identifier. It can be represented as (G, OID, ST), where G is a G-net, OID is the unique object identifier and ST is the state of the object.

Definition 2.4 *Generic Switching Place (GSP)*

A *Generic Switch Place (GSP)* is a triple of (NID, MS, AS), where NID is a unique identifier (class identifier) of a G-net G; MS is a set of methods defined as the interface of G-net G; and AS is a set of attributes defined as a set of instance variables.

Definition 2.5 *Internal Structure (IS)*

The *internal structure* of G-net *G* (representing a class), *G.IS*, is a net structure, i.e., a modified Pr/T net. *G.IS* consists of a set of *methods*.

Definition 2.6 *Method*

A method is a triple (P, T, A), where P is a set of places with three special places called *entry place*, *ISP place* and *goal place*. Each method can have only one *entry place* and one *goal place*, but it may contain multiple *ISP places*. 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-{*goal place*}) x T) ∪ ((T x (P-{*entry place*}).

2.4 Extending G-nets to Support Inheritance

An example of G-nets is shown in Figure 2. Here the G-net model represents an unbounded buffer class. The generic switch place is represented by *GSP(UB)* enclosed by an ellipsis, and the internal structure of this model is represented by a rounded box which contains the detailed design of four methods: *isEmpty()*, *put(e)*, *get()* and *who()*. The functionality of these methods are defined as follows: *isEmpty()* checks if the buffer is empty and return a boolean value, *put(e)* stores an item *e* into the buffer, *get()* removes an item from the buffer and returns that item, and *who()* prints the object identifier of the unbounded buffer. For clarity, in Figure 2, we put the signatures of these four methods in a rectangle on the right side of the *GSP* place as the interface of G-net UB. An example of *ISP* is shown in the method *get()* (denoted as an ellipsis), where the method *get()* makes a method call *isEmpty()* to the G-net module/object itself to check if the

buffer is empty. Note that we have extended G-nets to allow the use of the keyword **self** to refer to the module/object itself.

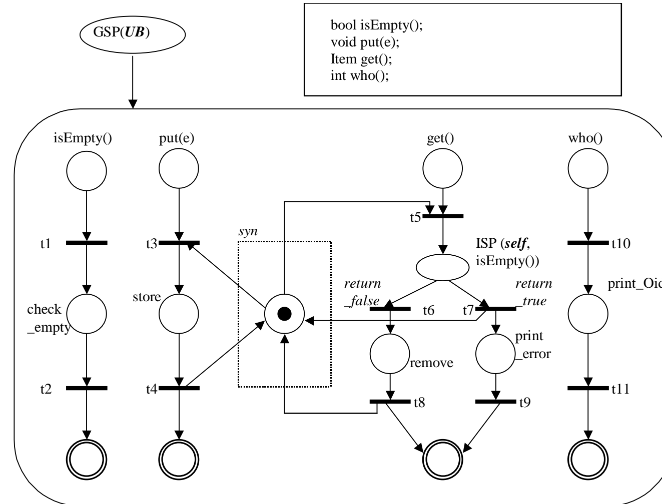


Figure 2. G-net model of unbounded buffer class (UB)

To deal with the concurrency issue in our G-net models, we extended our model by introducing a synchronization module to synchronize methods defined in the internal structure of the G-net. For instance, in the unbounded buffer class model we introduced a synchronization module *syn* to synchronize the methods *get()* and *put(e)*. This mechanism is necessary because these methods need to access the same unbounded buffer and they should be mutually exclusive. Generally, to design the synchronization module, we can either fulfill all synchronization requirements in one synchronization module or distribute them in several synchronization modules. To simplify our model, we follow the second option. Therefore, each class model may contain as many synchronization modules as necessary, and each synchronization module can be used to synchronize among a group of methods. As we will see, the synchronization module can not only be used to synchronize methods defined in a class model, but also can be used to synchronize methods defined in a subclass model and methods defined in its superclass (ancestor) model.

With inheritance, when we instantiate a G-net *Sub_G* (a subclass), it is not enough to just associate an *Oid* with *Sub_G* and initialize the state variables defined in *Sub_G* class. We must associate the same *Oid* with all of *Sub_G*'s superclasses (ancestors) and initialize all state variables defined in those classes. The initialized part corresponding to the subclass and each of the superclasses (ancestors) is called *primary subobject* and *subobject* respectively [Rossie *et al.* 1996][Drake 1998]. When a method call is made to the object *Sub_G_obj* (i.e., an instantiation of class *Sub_G*), it is always the case that only the *GSP* place of the

primary subobject is marked. The subobjects corresponding to the superclasses (ancestors) of *Sub_G* are not activated unless the method call to *Sub_G_obj* is not defined in the subclass model *Sub_G*.

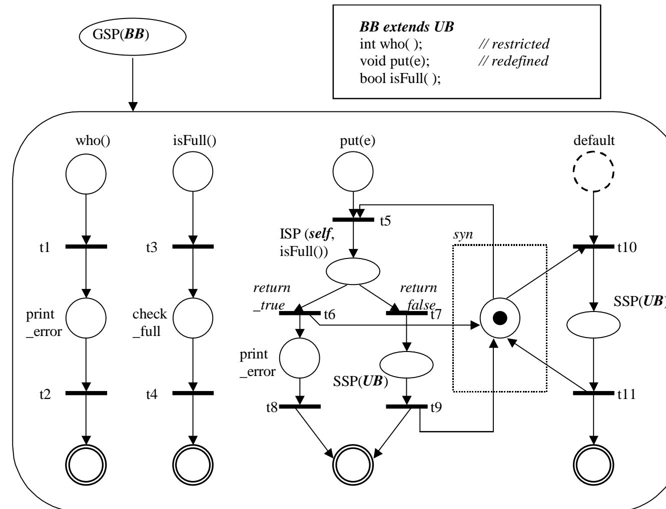


Figure 3. G-net model of bounded buffer class (BB)

When a method call is not found in a subclass model, we need to resolve the problem by searching the methods defined in the superclass models. To do this, we define a new mechanism called a *default place*. A *default place* is a default *entry place* defined in the *internal structure* of a subclass model and is drawn as a dash-lined circle, as shown in Figure 3. When a method is dispatched in a subclass model, the methods defined in the subclass model are searched first. If there is a match, one of the *entry places* of those methods is marked; otherwise, the *default place* is marked instead. After the dispatching, necessary synchronization constraints are established by the *synchronization modules*. If the *default place* is marked, the method call is then forwarded to a named superclass model. At first, it may seem that we can use the *ISP* method invocation mechanism to forward an existing method call. However this is not quite proper. Note that the initial method call will attach information associated with the call to the *gTkn* token. Now the subsequent call to the superclass would again attach the same information to the token, and the method call will actually be invoked more than once. To solve this problem, we introduce a new mechanism called a *Superclass Switch Place (SSP).*

An *SSP* (denoted as an ellipsis in Figure 3) is similar to an *ISP*, but with the difference that the *SSP* is used to forward an existing method call to a *subobject* (corresponding to a superclass model) of the object itself rather than to make a new method call. Essentially, an *SSP* does not update the *gTkn* token because all the information for the method call has already been attached by the original *ISP* method call. In the context of multiple inheritance, we represent an *SSP* mechanism in subclass *Sub_G* as *SSP(G'),* where *G'* is one of the

superclasses of *Sub_G*. Note that the object identifier is not necessary, as in the case of *ISP* method invocation, because the method call will be forwarded to the object itself (i.e., its subobject). When the method call is forwarded to the subobject corresponding to the superclass model *G'*, the *GSP* place of the superclass model *G'* is marked, and the methods defined in the superclass model are searched. If a method defined in the superclass model is matched, as in the case of *ISP* method invocation, the matched method is executed, and the result is stored in *gTkn.msg.result* and the *gTkn* token returns to the *SSP* place. Otherwise, the *default place* (if any) in the superclass is marked, and the methods defined in the grandparent class model are searched. This procedure can be repeated until the called method is found. If the method searching ends up in a class with no methods matched and no *default place* defined, a "method undefined" exception should be raised. This situation can be avoided by static type checking.

Now consider a bounded buffer class example as shown in Figure 3. We define a bounded buffer class BB as a subclass of an unbounded buffer class UB. Since the buffer has a limited size of *MAX_SIZE*, when there is a *put (e)* method call, the size of the buffer needs to be checked to make sure that the buffer capacity is not exceeded. In this case, the method *put (e)* defined in the class model UB is no longer correct, and it needs to be redefined in the subclass model BB. A simple way to redefine the method *put (e)* in subclass BB is to first make an *ISP* method call *isFull()* to the bounded buffer object itself. The method *isFull()* is used to check if the bounded buffer is full and it is added to the BB class model as shown in Figure 3. If it returns true, i.e., the bounded buffer has already been full, an error or exception will be generated; otherwise, the method call *put(e)* will be forwarded to its superclass UB by using an *SSP* mechanism. Here we use an *SSP* to allow reuse of the original method *put(e)* defined in class UB. As we will explain later, we call this situation refinement inheritance. Note that if we use *ISP(**self**, put(e))* in this situation, a dead loop will occur. This is because the methods defined in the subclass will always be searched first; and consequently, the method *put(e)* defined in subclass BB will be called recursively. Again we see the value of introducing the *SSP* mechanism.

It is also important to notice that a synchronization module can be used to synchronize methods defined in a subclass model and methods defined in the superclass model. However, in this case, all methods defined in superclass (ancestor) models must be synchronized as a whole. For instance, in Figure 3, the refined method *put(e)* defined in subclass BB is synchronized with all methods defined in the superclass UB, yet the synchronization between the method *put(e)* and the inherited method *isEmpty()* is unnecessary.

To formally define extended G-nets with inheritance, we need to redefine the *internal structure* and define the concept of *Synchronization Module* and *Abstract Superclass Module*. Based on the formal definitions of standard G-net model in Section 2.2, we now provide a few key definitions for our extended G-net models with inheritance features.

Definition 2.7 *Internal Structure (IS)* *// to replace definition 2.5*

The *internal structure* of G-net *G* is a triple (M, S, A), where M is a set of *methods*, S is a set of *synchronization modules*, and A is an optional *Abstract Superclass Module*. The arcs connecting M and S, or connecting S and A belong to S. There are no direct arcs between M and A.

Definition 2.8 *Synchronization Module*

A synchronization module is 4-tuples (P, A, I, O), where P is a single place used to hold an *sTkn* token, which is a colorless token, and A is a set of arcs defined as: (P x IS.M.T) ∪ (IS.M.T x P); I is a set of arc inscriptions on place incoming arcs, and O is a set of arc inscriptions on place outgoing arcs.

Definition 2.9 *Abstract Superclass Module*

An *Abstract Superclass Module* is a triple (P, T, A), where P is a set of places includes three special places: *default place*, *goal place* and *Superclass Switch Place (SSP)*. T is a set of transitions with optional guards. A is a set of arcs defined as: ((P – {*goal place*}) x T) ∪ (T x (P – {*default place*})).

2.5 Modeling Different Forms of Inheritance

Typically, to create a subclass model, we specialize a superclass by adding new protocols. We call this *augment inheritance* [Drake 1998]. Alternatively, we can restrict or refine a superclass by overriding one or more of its methods. This happens in three cases: method restriction, method replacement and method refinement. We call each of them *restrictive inheritance*, *replacement inheritance* and *refinement inheritance* [Drake 1998].

Augment inheritance is straightforward - new protocols, which are not defined in the superclass model, are added to a subclass model. For instance, consider the design of the subclass BB as shown in Figure 3. We require a service to check if the buffer is already full. This can be done by adding a new method *isFull()* to the subclass BB. Since the method *isFull()* does not override any methods in class UB, we have used augment inheritance.

In some cases, we regard a class as a specialization of another class, with some superclass methods absent from the protocol of the subclass. We call this type of inheritance *restrictive inheritance*. Restrictive inheritance actually runs counter to the semantics and intentions of inheritance, because the "IS-A" relationship between superclass and subclass is broken. However, restrictive inheritance may be necessary when using an existing class hierarchy that cannot be modified. Usually, restrictive inheritance is implemented in the subclass by overriding the disallowed superclass methods to produce error messages or signal exceptions. Here we use a trivial example to illustrate how to model restrictive inheritance. Suppose

we need to disallow the inherited method *who()* in our subclass BB. This can be simply done by redefining method *who()* in class BB; the redefined method *who()* does nothing but prints an error message to indicate that the method call for *who()* is disallowed in subclass model BB.

A subclass can completely redefine the behavior of its superclass for a particular method defined in the superclass. Inheritance in this case is called *replacement inheritance*. With this form of method overriding, we say that the method in the subclass replaces the method defined in the superclass. Replacing a superclass method generally occurs when the subclass can define a more efficient method or needs to define a method in a different way. An example of replacement inheritance would be possible in the bounded buffer example, if we redesign the method *get()* in subclass BB to make the "remove" action more efficient.

More frequently, the semantics of a subclass demand that the subclass respond to a method call by a method that includes the behavior of its superclass, but extends it in some way. In this case, we say that the subclass method refines the superclass method, i.e., there is a *refinement inheritance*. Practically, method refinement is more common than method replacement because it provides a semantic consistence with specialization. When implementing method refinement, we may simply refine the method by *copying* the relevant superclass method into the subclass model. However, we would like our extended G-net formalism to provide a mechanism that supports automatic sharing of the superclass method. This capability is supported by the *SSP* mechanism and it has been illustrated by the method refinement of *put(e)* in bounded buffer BB as shown in Figure 3.

2.6 Modeling Inheritance Anomaly Problem

Inheritance anomaly refers to the phenomenon that synchronization code cannot be effectively inherited without non-trivial re-definitions of some inherited methods [Matsuoka and Yonezawa 1993][Thomas 1994]. As a consequence, some well-known proposals for concurrent object-based languages, such as families of Actor languages, POOL/T, Procol and ABCL/1, chose to not support inheritance as a fundamental language feature [Matsuoka and Yonezawa 1993]. Also some languages like Concurrent Smalltalk or Orient84/K do provide inheritance but do not support intra-object concurrency - that is there is only a single thread of control within an object [Thomas 1994].

There have been previous efforts to solve the inheritance anomaly problem [Mitchell and Wellings 1996], but most of the proposals are based on quasi concurrency, where only one thread at a time is allowed to execute. As stated in [Thomas 1994], this type of inheritance anomaly seems to be almost solved. "True" concurrency refers to cases that more than one thread can be executed in an object at the same time. Reference [Thomas 1994] talked about solutions in this context. The inheritance anomaly problem has

usually been approached in terms of analyzing the causes. The causes have been classified as partitioning of acceptable states, history-only sensitiveness of acceptable states, and modification of acceptable states [Matsuoka and Yonezawa 1993]. Here, we analyze the inheritance anomaly problem based on clarifying the terminology of "synchronization constraints", and we always view a concurrent system as a "true" one.

As we will see, synchronization constraints among methods can be specified explicitly or implicitly. An explicit synchronization constraint refers to the concurrent/mutual exclusive execution between two methods in an object. For instance, in the unbounded buffer example, method *get()* and method *who()* can be executed concurrently, however the execution of method *get()* and method *put(e)* must be mutually exclusive. This type of synchronization constraint creates the inheritance anomaly problem when a method *m1* defined in a subclass module needs to be mutually exclusive with a particular inherited method *m2* that is defined in its superclass (ancestor) module. A simple way to deal with this situation is to refine the method *m2* (e.g., to use the SSP mechanism in our extended G-net model) and to establish mutual exclusion between *m1* and *m2* in the subclass module. In this case the method defined in the superclass (ancestor) module can be reused by a refinement inheritance.

An implicit synchronization constraint refers to cases where acceptance of a method in an object is based on that object's state. The state of an object can be changed by executing a method in that object. For instance, when a buffer is in a state of "empty", the method *get()* is not allowed to execute; however, after executing the method *put(e)*, the state of the buffer is changed from "empty" to "partial," and at this time, the method call of *get()* becomes acceptable. Since the methods *get()* and *put(e)* are indirectly synchronized through the state of the buffer, we called this type of synchronization constraint an implicit synchronization constraint. The implicit constraints can be further classified in terms of two different views of an object's state, namely internal view and external view. Under an internal view, the state of an object can be captured by the evaluation of state variables of the object [Matsuoka and Yonezawa 1993]. For example, the state "empty" of a buffer can be captured by checking if the state variable of *buffer_size* evaluates to "0". This type of synchronization can always be added to a subclass module without redefining inherited methods because it can be easily maintained by checking state variables before allowing the execution of a method.

Another view is the external view, where the state is captured indirectly by the externally observable behavior of the object [Matsuoka and Yonezawa 1993]. For example, a state under external view could be the state of a buffer object when the last executed method is *put(e)*. When synchronization constraints with respect to the external view of an object's state are added to a subclass module, some methods defined in a superclass (ancestor) module must be redefined. Fortunately, in most cases, as long as no deadlocks are introduced, we can again use refinement inheritance to reuse the original method defined in the superclass (ancestor) module. We use the classic example of *gget()* to illustrate this situation. Consider a new bounded

buffer class BB1, defined as a subclass of bounded buffer class BB, and add a new method called *gget()*. The behavior of *gget()* is almost identical to that of *get()*, with the sole exception that it can not be executed immediately after the invocation of *put(e)* [Matsuoka and Yonezawa 1993]. The design of the new bounded buffer BB1 is illustrated in Figure 4. To establish the synchronization between methods *gget()* and *put(e)*, the method *put(e)* must be redefined in the subclass module BB1. Suppose we have an object *bb1*, an instance of class BB1. Initially, the token in the synchronization module *syn* is "0". Whenever there is a method call other than *put(e)* to object bb1, the token will be removed and deposited back to the synchronization module with the same value of "0". However, if there is a method call for *put(e)*, the token in the synchronization module *syn* will be removed first, and then the method call *put(e)* will be forwarded to its superclass BB by using the *SSP(BB)* mechanism. After the method call of *put(e)*, a token with value "1" will be deposited into the synchronization module *syn*. At this time, if there is a method call for *gget()*, the call must wait because a token with value "0" is necessary to enable the transition *t1*. Thus the synchronization between methods *gget()* and *put(e)* is correctly established. Note that we cannot reuse the method *get()* when designing the method *gget()* by using the *SSP(BB)* mechanism. This is inapplicable because *gget()* and *get()* are two different methods. In addition, we need to redefine the methods *isEmpty()* and *isFull()* to avoid deadlocks.

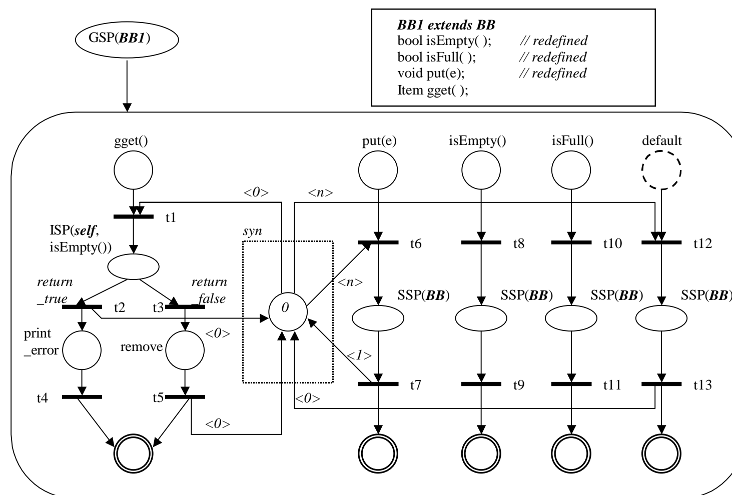


Figure 4. G-net model of bounded buffer class (BB1)

2.7 Discussion

Inheritance has been introduced into several object-oriented net models, such as LOOPN++ [Lakos and Keen 1994] and CO-OPN/2 [Biberstein *et al.* 1997]. However, those methods do not use net-based extensions to capture inheritance properties. Our approach explicitly models inheritance at the net level to

maintain an underlying Petri net model that can be exploited during design simulation or analysis. In future work, we will explore an algorithmic basis for synthesis of subclass models as well as investigate how to analyze extended G-nets at an abstract level, with consideration for the state explosion problem. Issues like design consistency and deadlock avoidance will be of primary concern.

Chapter 3

An Agent-based G-net Model

3.1 Introduction

Agents are becoming one of the most important topics in distributed and autonomous decentralized systems (ADS) [Mendes *et al.* 1997][Arai *et al.* 1999]. With the increasing importance of electronic commerce across the Internet, the need for agents to support both customers and suppliers in buying and selling goods or services is growing rapidly. Most of the technologies supporting today's agent-based electronic commerce systems stem from distributed artificial intelligence (DAI) research [Guttman *et al.* 1998][Green *et al.* 1997]. Applications developed with multi-agent systems (MAS) in electronic commerce are examples of such efforts. A multi-agent system (MAS) is a concurrent system based on the notion of autonomous, reactive, and internally-motivated agents in a decentralized environment. 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 [Green *et al.* 1997]. The notable systems developed with MAS in electronic commerce are Kasbah [Chavez and Maes 1996] and MAGMA [Tsvetovatyy *et al.* 1997]. Kasbah is meant to represent a marketplace where Kasbah agents, acting on behalf of their owners, can filter through ads and find those that their users might be interested in. The agents then proceed to negotiate to buy and sell items. MAGMA moves the marketplace metaphor to an open marketplace involving agents buying/selling physical goods, investments and forming competitive/cooperative alliances. These agents negotiate with each other through a global blackboard.

Notice that the example we provide in Figure 1 (Chapter 2) follows the *Client-Server* paradigm, in which a *Seller* object works as a server and a *Buyer* object is a client. Although the standard G-net model works well in object-based design, it is not sufficient in agent-based design for the following reasons:

1. Agents in multi-agent systems are usually developed by different vendors independently, and those agents will be widely distributed across large-scale networks such as the Internet. To make it possible for those agents to communicate with each other, it is essential for them to have a common

communication language and to follow common protocols. However the standard G-net model does not directly support protocol-based language communication between agents.

2. 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.
3. 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. Standard G-net models can only directly support a predefined flow of control.

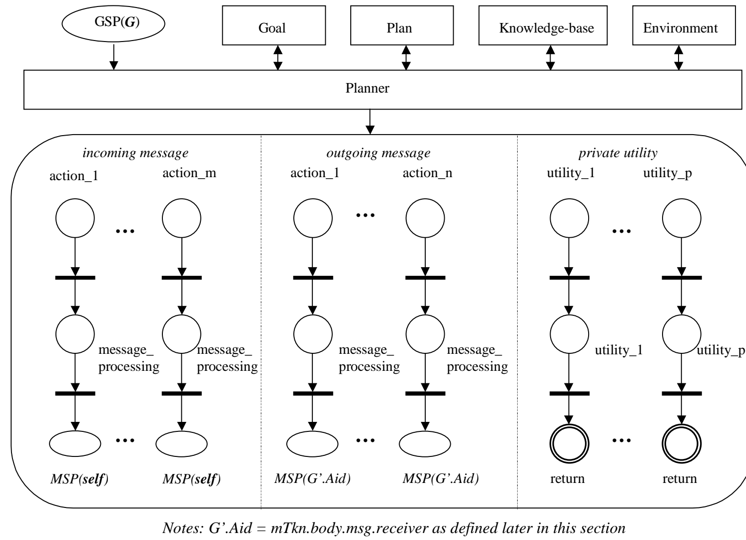
3.2 Agent-based G-net Model

To support agent-based design, we first need to extend a G-net to support modeling an agent class[1]. The basic idea is similar to extending a G-net to support class modeling for object-based design [Xu and Shatz 2000]. 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 [Shoham 1993]) 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. The template for an agent-based G-net model is shown in Figure 5. We describe each of the additional modules as follows:

- A *Goal* module is an abstraction of a goal model [Kinny *et al.* 1996], 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 [Kinny *et al.* 1996] 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 [Kinny *et al.* 1996], 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.
- A *Planner* module is 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

[1] We view the abstract of a set of similar agents as an agent class, and we call an instance of an agent class an agent or an agent object.

plans are achieved, and the *Goal*, *Plan* and *Knowledge-base* modules of an agent are updated after each communicative act [Finin *et al.* 1997][Odell 2000] or if the environment changes.



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

Figure 5. A generic agent-based G-Net model

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 correspond to a subset of communicative acts. Each *MPU*, labeled as *action_i* in Figure 5, 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.

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 [Iglesias *et al.* 1998]. 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) [FIPA 2000][Finin *et al.* 1997]. Note that in Figure 5, 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 5), 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 [Jennings *et al.* 1998], 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 6. Since the modules *Goal, Plan* and *Knowledge-base* have the same interface with the Planner module, for brevity, we represent them as a single special place (denoted by double ellipses in Figure 6), which contains a token *Goal/Plan/KB* that represents a set of goals, a set of plans and a set of beliefs. The *Environment* module is also represented as a special place that contains a token *Environment* as a model of the outside world of the agent. The *Planner* module is goal-driven because the transition *start_a_conversation* may fire whenever an attempt is made to achieve a committed goal. In addition, the *Planner* module is also message-triggered because certain actions may initiate whenever a message arrives (either from some other agent or the agent itself). If the message comes from some other agent, it will be dispatched to a *MPU* defined in the *incoming messages* section of the agent-based G-net's internal structure. After the message is processed, the *MPU* will transfer the processed message as a token to the *GSP* place of the agent itself. This is done by sending a message *MSP(**self**)* to the agent itself. Upon arrival of this internal message, the transition *internal* may fire, and the next action will be determined based on the agent's current mental state. Alternatively, the next action could be to ignore the message or to continue with the current conversation. In either case, a token will be deposited in place *update_goal/plan/kb*, and the transition *update* may fire. As a consequence, the agent's mental state may change. If the next action is to continue the conversation, the tag of the token will be changed from **internal** to **external**, and the token will be deposited in place *dispatch_outgoing_message*. In this case, the corresponding *MPU* will be called before the message is sent to some other agent by using the *MSP* mechanism. In addition, an agent may provide a set of private utility functions for itself and allow other functional units to make synchronous method calls to it. Whenever there is a method call, the token deposited in the *GSP* place will be moved to place *dispatch_utilities* and then will be dispatched to a method defined in the *private utilities* section.

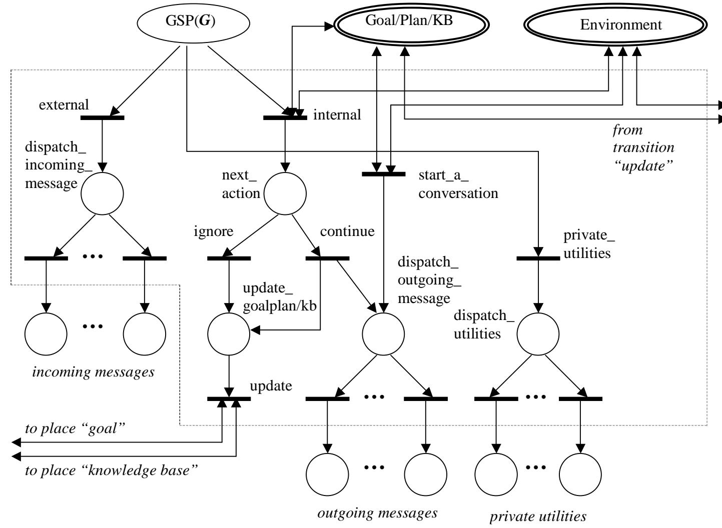


Figure 6. A template of *Planner* module

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 {
    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 [9].
3. After all the *ISP* method calls are finished and the *mTkn* token returns to the original *ISP*, the *mTkn.tag* is set back as *mTkn.body.old_tag*, and both the variables *old_tag* and *miv* are detached.

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

Definition 3.1 *Agent-based G-net*
An *agent-based G-net* is a 7-tuple *AG = (GSP, GL, PL, KB, EN, PN, IS)*, where *GSP* is a *Generic Switch Place* providing an abstract for the agent-based G-net, *GL* is a *Goal* module, *PL* is a *Plan* module, *KB* is a *Knowledge-base* module, *EN* is an *Environment* module, *PN* is a *Planner* module, and *IS* is an *internal structure* of *AG*.

Definition 3.2 *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 3.3 *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 3.4 *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*}) x T) ∪ ((T x (P-{*entry*}).

Definition 3.5 *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*}) x T) ∪ ((T x (P-{*entry*}).

3.3 Selling and Buying Agent Design

To illustrate how to design a selling/buying agent by using our agent-based G-net model, we use an example derived from [Odell 2000]. Figure 7 (a) is a modified example of an FIPA contract net protocol, which depicts a template of protocol expressed as a UML sequence diagram for a price-negotiation protocol between a buying agent and a selling agent. To correctly draw the sequence diagram for this template, we need to introduce two new notations, i.e., the end of protocol operation "•" and the iteration of communicative acts operation "∗". Examples of using these two notations are as follows. In Figure 7 (a), we put a mark of "•" in front of the message name "*refuse*" to indicate that this message ends the protocol. In the same figure, a mark "∗" is put on the right corner of the narrow rectangle for the message "*propose*" to indicate that the communicative actions in this section can be repeated zero or more times.

When a conversation based on this contract net protocol begins, the buying agent sends a request for price to a selling agent. The selling agent can then choose to response to the buying agent by refusing to provide price or submitting a proposal. Here the "x" in the decision diamond indicates an exclusive-or decision. If a proposal is offered, the buying agent has a choice of either accepting or rejecting the proposal. If a selling agent receives a *reject-proposal* message, it may send the buying agent a new proposal or replies the buying agent with a confirmation message. If the selling agent receives an *accept-proposal* message, it will simply send a confirmation message to the buying agent. Whenever a confirmation message is sent, the protocol ends. Figure 7 (b) and 7 (c) shows two actual cases of this protocol template. In Figure 7 (b), the

selling agent's proposal is accepted by the buying agent in one round; while Figure 7 (c) shows the case that the proposal is accepted by the buying agent in the second round.

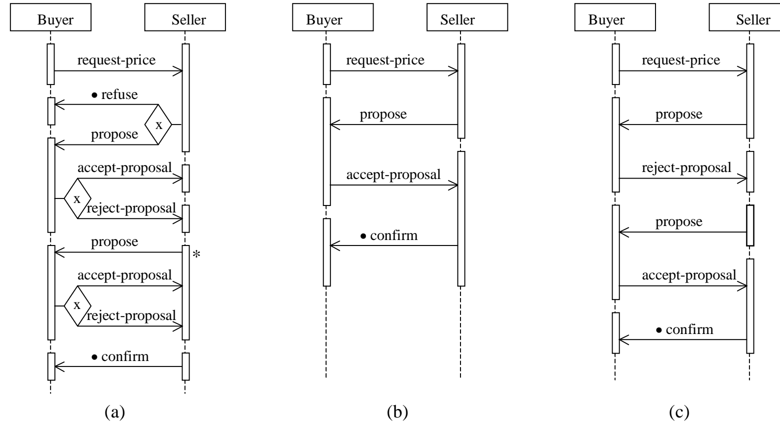
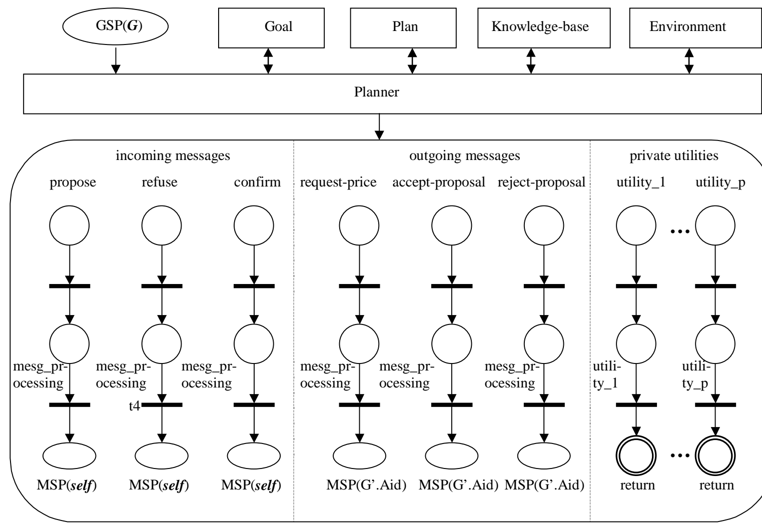


Figure 7. A contract net protocol between buying and selling agent



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

Figure 8. An Agent-based G-net model for buying agent class

Based on the communicative acts (e.g., request-price, propose etc.) needed for this contract net protocol, we may design the buying agent as in Figure 8. In Figure 8, the *Goal* and *Knowledge-base* modules remain as abstract units and can be refined in further detailed design. The *Planner* module may use Figure 6 as a template, with the transition *start_a_conversation* and the place *next_action* left to be refined in further detailed design too. In the *private utilities* section, we may define some necessary functions that can be called by the buying agent itself. Examples of such private utility functions could be: *compare_price*,

update_know-ledge_base etc. The design of the selling agent is similar. We define *MPUs* of *request-price*, *accept-proposal* and *reject-propose* in the *incoming messages* section of the selling agent, and define *MPUs* of *propose*, *refuse* and *confirm* in the *outgoing messages* section of the selling agent.

3.4 Verifying Agent-based G-net models

One of the advantages of building a formal model for agents in agent-based design is to ensure a correct design that meets certain specifications. A correct design of agents at least has the following properties:

- *L3-live*: any communicative act can be performed as many times as needed.
- *Concurrent*: a number of conversations among agents can happen at the same time.
- *Effective*: an agent communication protocol can be correctly traced in the agent models.

To verify the correctness of agent-based G-net models for selling/buying agents with respect to the above properties, we first reduce our agent-based G-net models to an ordinary Petri net as follows: (1) simplify the *Goal* module and *Knowledge-base* module as ordinary places with ordinary tokens; (2) omit the *public services* and *private utilities* sections; (3) simplify *mTkn* tokens as ordinary tokens; (4) use net reduction to simplify the Petri net corresponding to an *MPU/Method* as a single place; and (5) use the close world assumption and make our system only contains two agents, i.e., a buying agent and a selling agent.

The resulting ordinary Petri net is illustrated in Figure 9. To verify the correctness of our agent-based G-net model for agent communication, we utilize some key definitions and theorems as adapted from [Murata 1989].

Definition 3.6 *Incidence Matrix*

For a Petri net *N* with *n* transitions and *m* places, the *incidence matrix* A = $[a_{ij}]$ is an *n* x *m* matrix of integers and its typical entry is given by

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$$

where $a_{ij}^{+} = w(i,j)$ is the weight of the arc from transition *i* to output place *j* and $a_{ij}^{-} = w(j,i)$ is the weight of the arc from input place *j* to transition *i*.

Definition 3.7 *Firing Count Vector*

For some sequence of transition firings in a Petri net *N*, a *firing count vector x* is defined as an n-vector of nonnegative integers, where the *i*th entry of *x* denotes the number of times that transition *i* must fire in that firing sequence.

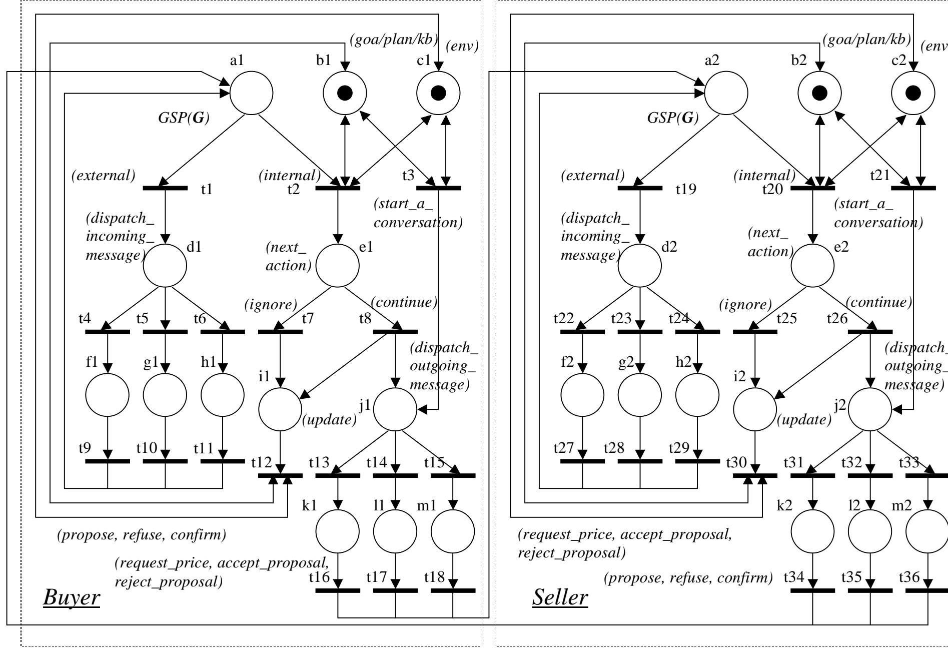


Figure 9. A transformed model of buying and selling agents

Definition 3.8 *T-invariant*

For a Petri net *N*, an n-vector x of integers ($x \neq 0$) is called a *T-invariant* if x is an integer solution of homogeneous equation $A^T x = 0$, where A is the incidence matrix of Petri net *N*.

Definition 3.9 *Support and minimal-support T-invariant*

The set of transitions corresponding to non-zero entries in a T-invariant $x \geq 0$ is called the *support* of a T-invariant and is denoted as $\|x\|$. A support is said to be *minimal* if no proper non-empty subset of the support is also a support. Given a minimal support of a T-invariant, there is a unique minimal T-invariant corresponding to the minimal support. Such a T-invariant is called the *minimal-support T-invariant*.

Definition 3.10 *L3-live Petri net*

A Petri net *N* with initial marking M_0, denoted as (N, M_0), is said to be *L3-live* if for every transition *t* in the net, *t* appears infinitely often in some firing sequence $L(N, M_0)$, where $L(N, M_0)$ is the set of all possible firing sequences from M_0 in the net (N, M_0).

Theorem 3.1 An n-vector x is a T-invariant of a Petri net *N* *iff* there exists a marking M_0 and a firing sequence σ that reproduces the marking M_0, and x defines the firing count vector for σ.

Theorem 3.2 A Petri net *N* with initial marking M_0 is *L3-live* if there exists a set of minimal-support T-invariants that covers all the transitions in the net, and for each minimal-support T-invariant there exists a firing sequence that reproduces the initial marking M_0.

Proof: Let T be the set of transitions in Petri net (N, M_0), Γ be the set of minimal-support T-invariants that covers all the transitions in T. From the given condition, we know that for $\forall t \in$ T, $\exists \chi \in \Gamma$, which covers transition *t*. Since for the minimal-support T-invariant χ, there exists a finite firing sequence ρ that reproduces the initial marking M_0, *t* appears in ρ. Let the infinite firing sequence $\sigma = \rho \bullet \rho \bullet \rho \bullet \rho \ldots$, where "•" is the concatenation operator between finite sequences, *t* appears in σ infinitely often. By definition 4.5, Petri net (N, M_0) is *L3-live*. ◊

	a	b	c	d	e	f	g	h	i	j	k	l	m	a	b	c	d	e	f	g	h	i	j	k	l	m
	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2
t1	-1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t2	-1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t4	0	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t5	0	0	0	-1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t6	0	0	0	-1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t7	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t8	0	0	0	0	-1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t9	1	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t10	1	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t11	1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t12	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t13	0	0	0	0	0	0	0	0	0	-1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t14	0	0	0	0	0	0	0	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t15	0	0	0	0	0	0	0	0	0	-1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
t16	0	0	0	0	0	0	0	0	0	0	-1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
t17	0	0	0	0	0	0	0	0	0	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
t18	0	0	0	0	0	0	0	0	0	0	0	0	-1	1	0	0	0	0	0	0	0	0	0	0	0	0
t19	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	1	0	0	0	0	0	0	0	0	0
t20	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0	0	0
t21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
t22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	1	0	0	0	0	0	0	0
t23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	1	0	0	0	0	0	0
t24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0
t25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0
t26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	1	1	0	0	0
t27	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-1	0	0	0	0	0	0	0
t28	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	-1	0	0	0	0	0	0
t29	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	-1	0	0	0	0	0
t30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0
t31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	1	0	0
t32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	1	0
t33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	1
t34	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0
t35	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0
t36	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1

Table 1. The incidence matrix A of the Petri net in Figure 9

The incidence matrix A of the Petri net in Figure 9 is listed in Table 1. By using Definition 3.6 and 3.9, we can calculate a set of minimal-support T-invariants as follows:

x_1 = [1 1 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0]

x_2 = [0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0]

x_3 = [1 1 1 1 0 0 1 0 1 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 0 1 1 0 0 1 0 0]

x_4 = [1 1 1 0 0 1 1 0 0 0 1 1 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 0 1]

x_5 = [1 1 0 0 1 0 0 1 0 1 0 1 0 0 1 0 0 1 1 1 1 0 0 1 1 0 0 0 1 1 0 1 0 0 1 0]

From Theorem 3.1, for each minimal-support T-invariant x_i in our example, there exists a marking M_0 and a firing sequence σ_i, which reproduces the marking M_0, and x_i defines the firing count vector for σ_i. Obviously, the following firing sequences $\sigma_1, \sigma_2, \ldots \sigma_5$ reproduce the initial marking M_0 = [0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0], and $x_1, x_2, \ldots x_5$ define the firing count vectors for $\sigma_1, \sigma_2, \ldots \sigma_5$ respectively:

σ_1 = <t21, t31, t34, t1, t4, t9, t2, t7, t12>

σ_2 = <t3, t13, t16, t19, t22, t27, t20, t25, t30>

σ_3 = *<t3, t13, t16, t19, t22, t27, t20, t26, t30, t31, t34, t1, t4, t9, t2, t7, t12>*

σ_4 = <t3, t14, t17, t19, t23, t28, t20, t26, t30, t33, t36, t1, t6, t11, t2,t7,t12>

σ_5 = <t21, t32, t35, t1, t5, t10, t2, t8, t12, t15, t18, t19, t24,t29,t20,t25,t30>

Since the above minimal-support T-invariants cover all the transitions in the net, and for each minimal-support T-invariant, there exists a firing sequence that reproduces the initial marking M_0, from Theorem 3.2, we conclude that our Petri net model with initial marking M_0 is *L3-live*, i.e., for any transition *t* in our net model, we can find an infinite firing sequence that *t* appears infinitely often. Consequently, any communicative act can be performed as many times as needed[2].

In Figure 9, it is obvious to see that our net model is unbounded. This is because transitions *t3* and *t21* can fire as many times as needed. This behavior shows that both the buying and selling agent may initiate conversations autonomously and concurrently (as we stated before, the initiation of a new conversation is goal driven). There can be as many conversations as necessary between the buying agent and the selling agent. As an example, a buying agent may request prices of several goods from a selling agent at the same time, and several buying agents may request price of the same goods from a selling agent concurrently.

In addition, we may trace an agent communication protocol *p* in our net model with a firing sequence σ. For a protocol *p*, a corresponding firing sequence σ in our net model has more semantics than the protocol itself because when we actually execute a protocol in our net, we need to do additional work, such as updating the goal or knowledge base after each communicative act. Since a marking M that is reachable from M_0, but $M \neq M_0$, represents that there are still some ongoing conversations in the net, to correctly

[2] One of the limitations for invariant approach is that it is not sufficient to prove a Petri net is *L4-live* or *live*, i.e., from any marking M that is reachable from M_0, it is possible to ultimately fire any transition of the net.

trace a protocol *p* in our net model, it is essential for us to find a firing sequence σ that reproduces the initial marking M_0. In other words, we need to make sure that there will be no residual tokens for a conversation left in the net after that conversation completes. In this case, we say that the protocol *p* can be *effectively* traced as a firing sequence σ in our net model. To show that a protocol *p* can be effectively traced, we use the contract net protocol examples in Figure 7 (b) and Figure 7 (c). These two protocols can be traced in our net model as follows:

σ_b = <t3, t13, t16, t19, t22, t27, t20, t26, t30, t31, t34, t1, t4, t9, t2, t8, t12, t14, t17, t19, t23, t28, t20, t26, t30, t33, t36, t1, t6, t11, t2, t7, t12>

σ_c = <t3, t13, t16, t19, t22, t27, t20, t26, t30, t31, t34, t1, t4, t9, t2, t8, t12, t15, t18, t19, t24, t29, t20, t26, t30, t31, t34, t1, t4, t9, t2, t8, t12, t14, t17, t19, t23, t28, t20, t26, t30, t33, t36, t1, t6, t11, t2, t7, t12>

By Definition 3.7, we calculate their corresponding firing count vectors x_b and x_c as follows:

x_b = [2 2 1 1 0 1 1 1 1 0 1 2 1 1 0 1 1 0 2 2 0 1 1 0 0 2 1 1 0 2 1 0 1 1 0 1]

x_c = [3 3 1 2 0 1 1 2 2 0 1 3 1 1 1 1 1 1 3 3 0 1 1 1 0 3 1 1 1 3 2 0 1 2 0 1]

By Definition 3.8, it is easy to verify that both x_b and x_c are T-invariants because both of the equations $A^T x_b = 0$ and $A^T x_c = 0$ are satisfied. This shows that both firing sequences σ_b and σ_c can reproduce the initial marking M_0. In other words, we prove that both protocols in Figure 7 (b) and 7 (c) can be effectively traced in our agent-based model.

3.5 Discussion

One of the most rapidly growing areas of interest for Internet technology is that of electronic commerce. Consumers are looking for suppliers selling products and services on the Internet, while suppliers are looking for buyers to increase their market share. For convenience and efficiency, we believe that multi-agent system (MAS) is an effective way to automate the time consuming process of looking for buyers or sellers and negotiate in order to obtain the best deal. Although there are several implementations of agent-based electronic marketplaces available [Chavez and Maes 1996][Tsvetovatyy *et al.* 1997], formal framework for such systems are few. It is an increasing need to provide formal methods in MAS specification and design to ensure robust and reliable products.

Chapter 4

A Framework for Modeling Agent-Oriented Software

4.1 Introduction

To avoid building a methodology from scratch, the researchers on agent-oriented methodologies have followed the approach of extending existing methodologies to include the relevant aspects of agents. These extensions have been carried out mainly in two areas: objected-oriented (OO) methodologies and know engineering (KE) methodologies [Iglesias *et al.* 1998]. Now we give a brief introduction to these two ways of extensions.

To extend object-oriented methodologies for agent modeling is a nature way for most of the software engineers. This is because there are similarities between the object-oriented paradigm and the agent-oriented paradigm [Kinny *et al.* 1996]. Since the early times of distributed artificial intelligence (DAI), the close relationship between DAI and Object-Based Concurrent Programming (OBCP) was established [Gasser and Briot 1992]. As stated by Shoham, the agents can be considered as *active objects*, i.e., objects with a mental state [Shoham 1993]. Both paradigms use message passing for communication and can use inheritance and aggregation for defining its architecture. The main difference is the constrained type of messages in the AO paradigm and the definition of a state of an agent in terms of its beliefs, desires and intentions [Iglesias *et al.* 1998].

The popularity of object-oriented methodologies is another potential advantage for this approach. Many object-oriented methodologies are being used in the industry with success. Examples of such methodologies are Object Modeling Technique (OMT) [Rumbaugh *et al.* 1991], Object-Oriented Software Engineering (OOSE) [Jacobson *et al.* 1992], Object-Oriented Design [Booch 1994] and Unified Modeling Language (UML) [Rational 1997]. This experience can be a key to facilitate the integration of agent technology into OO methodologies. This is because, on the one hand, the software engineers can be

reluctant to use and learn a complete new methodology, and on the other hand, the managers would prefer to follow methodologies that have been successfully tested.

Previous work based on this approach includes: agent modeling technique for systems of BDI agents [Kinny *et al.* 1996], agent-oriented analysis and design [Burmeister 1996] and agent unified modeling language (AUML) [Odell 2000].

For the second approach, knowledge engineering methodologies can provide a good basis for multi-agent systems modeling since they deal with the development of knowledge based systems. Since the agents have cognitive characteristics, these methodologies are quite helpful to modeling agent knowledge. The extension of current knowledge methodologies can take advantage of the acquired experience in these methodologies. In addition, both the existing tools and the developed problem solving method libraries can be reused. An example of this approach is the Gaia methodology for agent-oriented analysis and design suggested by Wooldridge and his colleagues [Wooldridge *et al.* 2000].

In this proposal, we adopt the first approach, however unlike previous work, our approach uses the principle of "separation of concerns" in agent-oriented design. We separate the traditional object-oriented features and reasoning mechanisms in our agent-oriented software model as much as possible, and we discuss how reuse can be achieved in terms of functional units, such as message processing units (*MPU*s) and private functions, in agent-oriented design. While some people advocated that inheritance has limited value in conceptual models of agent behavior [Jennings 2000][Wooldridge *et al.* 2000], 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 [Xu and Shatz 2001a], and the extensions that create an agent-oriented model are derived from the framework presented in [Xu and Shatz 2001b].

4.2 An Agent-Oriented Model

4.2.1 An Architecture for Agent-Oriented Modeling

To reuse the design of agent-based G-net model shown in Figure 5 (Chapter 3), we keep our agent-oriented G-net model to have the same structure as an agent-based G-net model. However, to deal with inheritance, we must revise our *Planner* module. In our new *Planner* module, we introduce new mechanisms such as *synchronous Superclass switch Place (ASP)*, and decision-making units such as abstract transitions. The template of the *Planner* module is shown as in Figure 10[3]. Similarly as before, the modules *Goal*, *Plan*, *Knowledge-base* and *Environment* are represented as four special places (denoted by double ellipses in

Figure 10), 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 10 (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 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*.

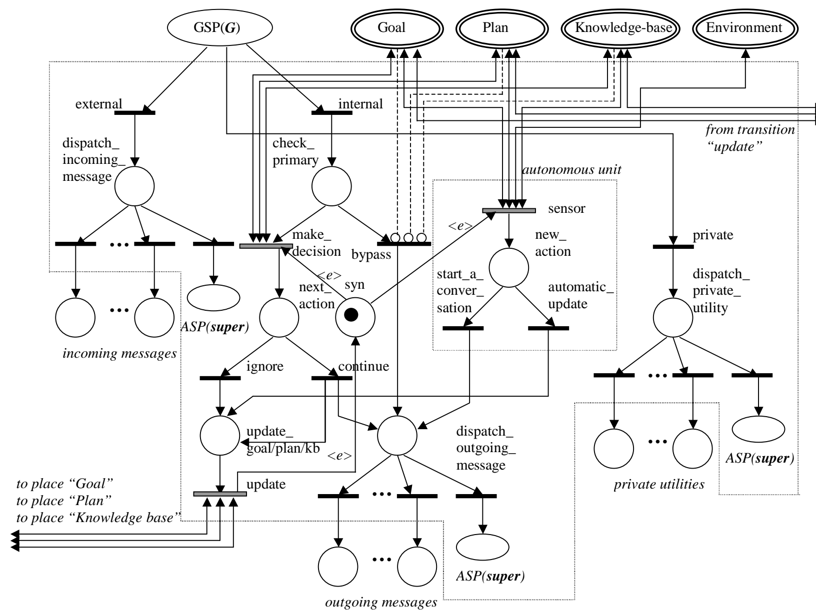


Figure 10. A template for the *Planner* module (initial design)

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

[3] Actually, this module purposely contains a somewhat subtle design error that is used to demonstrate the value of automated verification in Chapter 5.

internal/**external**/**private**. A message token with a tag of **external** represents an incoming message which comes from some other agent, or a newly generated outgoing message before sending to some other agent; while a message token with a tag of **internal** is a message forwarded by an agent to itself with the *MSP* mechanism. In either case, the message token with the tag of **internal/external** should not be involved in an invocation of a method call. On the contrary, a message token with a tag of **private** indicates that the token is currently involved in an invocation of some method call. When an incoming message/method arrives, with a tag of **external/private** in its corresponding token, it will be dispatched to the appropriate *MPU/method* defined in the internal structure of the agent. If it is a method invocation, the method defined in the *private utility* section of the internal structure will be executed, and after the execution, the token will return to the calling unit, i.e., an *ISP* of the calling agent. However, if it is an incoming message, the message will be first processed by a *MPU* defined in the *incoming message* section in the internal structure of the agent. Then the tag of the token will be changed from **external** to **internal** before it is transferred back to the *GSP* of the receiver agent by using *MSP(**self**)*. 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 4.2.2). 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 **external**, 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(Receiver)* mechanism. 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 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

[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.

connecting with the place *syn* to indicate that only ordinary tokens may be removed from or deposited into the place *syn*.

4.2.2 Inheritance Modeling in Agent-Oriented Design

Although there are different views with respect to the concept of agent-oriented design [Iglesias *et al.* 1998] [Jennings 2000], 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 [Kinny and Georgeff 1997][Crnogorac *et al.* 1997]. 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 4.1 *Subagent and Primary Subagent*

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

The result of initializing an agent class is to take the agent class as a template and create a concrete structure of the agent class and initialize its state variables. Since we represent an agent class as an agent-oriented G-net, an initialized agent class is modeled by an agent-oriented G-net with initialized state variables. In particular, the four tokens in the special places of an agent-oriented G-net, i.e., *gTkn*, *pTkn*, *kTkn* and *eTkn*, are set to their initial states. Since different subagents of *AO* may have goals, plans, knowledge and environment models that conflict with those of the primary subagent of *AO*, it is desirable to resolve them in an early stage. In our case, we deal with those conflicts in the instantiation stage in the following way. All the tokens *gTkn*, *pTkn*, *kTkn* and *eTkn* in each subagent of *AO* are removed from their associated special places, and the tokens are combined with the *gTkn*, *pTkn*, *kTkn* and *eTkn* in the primary subagent of *AO*.[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 10) 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 10) from the special places *Goal*, *Plan* and *Knowledge-base*. So the transition *bypass* can only be enabled when there are no tokens in these places. In contrast to this behavior, in the *Planner* module of a primary subagent, tokens do exist in the special places *Goal*, *Plan* and *Knowledge-base*. Thus, the transition *bypass* will never be enabled. Instead, the transition *make_decision* must fire before an outgoing message is dispatched.

To reuse the services (i.e., *MPUs* and *methods*) defined in a subagent, we need to introduce a new mechanism called *Asynchronous Superclass switch Place (ASP)*. An *ASP* (denoted by an ellipsis in Figure 10) 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.

[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.

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

4.3 Examples of Agent-Oriented Design

4.3.1 A Hierarchy of Agents in an Electronic Marketplace

Consider an agent family in an electronic marketplace domain. Figure 11 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.

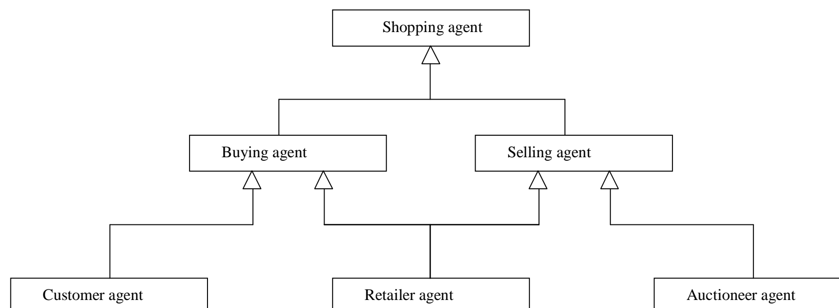


Figure 11. The class hierarchy diagram of agents in an electronic marketplace

4.3.2 Modeling Agents in an Electronic Marketplace

As in Chapter 3, to design an agent, we first need to define the necessary communicative acts of that agent. The communicative acts for a shopping agent, facilitator agent, buying agent and selling agent are shown as agent UML (AUML) sequence diagram in Figure 12. Figure 12 (a) depicts a template of a contract net protocol for a registration-negotiation protocol between a shopping agent and a facilitator agent. Figure 12 (b) is the same example of a contract net protocol as in Figure 7 (a), which depicts a template of a price-negotiation protocol between a buying agent and a selling agent. Figure 12 (c) shows an example of price-negotiation contract net protocol that is instantiated from the protocol template in Figure 12 (b).

Consider Figure 12 (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 12 (b).

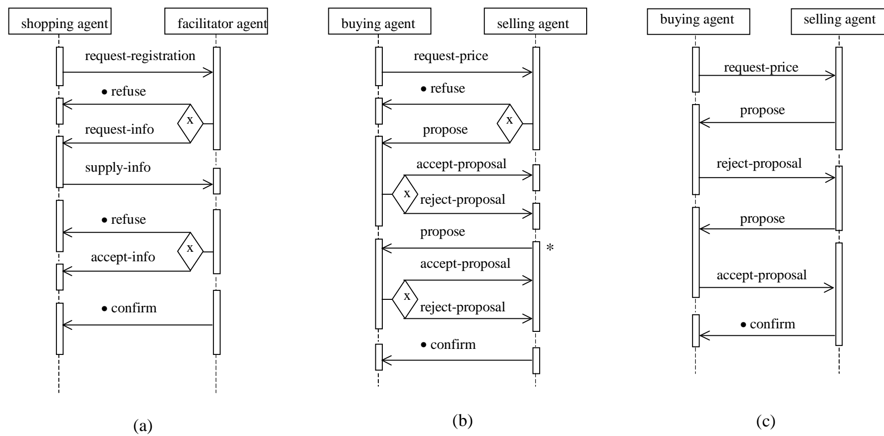


Figure 12. Contract net protocols (a) A template for the registration protocol (b) A template for the price-negotiation protocol (c) An example of the price-negotiation protocol

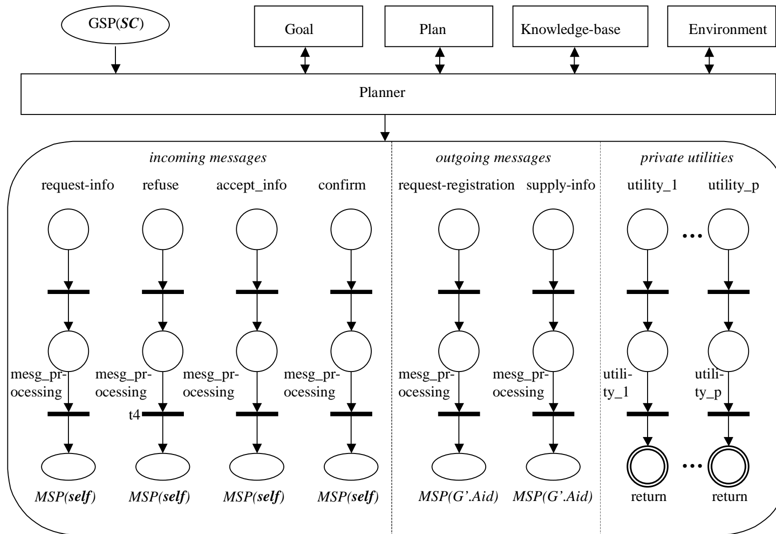


Figure 13. An agent-based G-Net model for shopping agent class (SC)

Based on the communicative acts (e.g., request-registration, refuse, etc.) needed for the contract net protocol in Figure 12 (a), we may design the shopping agent class as in Figure 13. 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 10. 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.

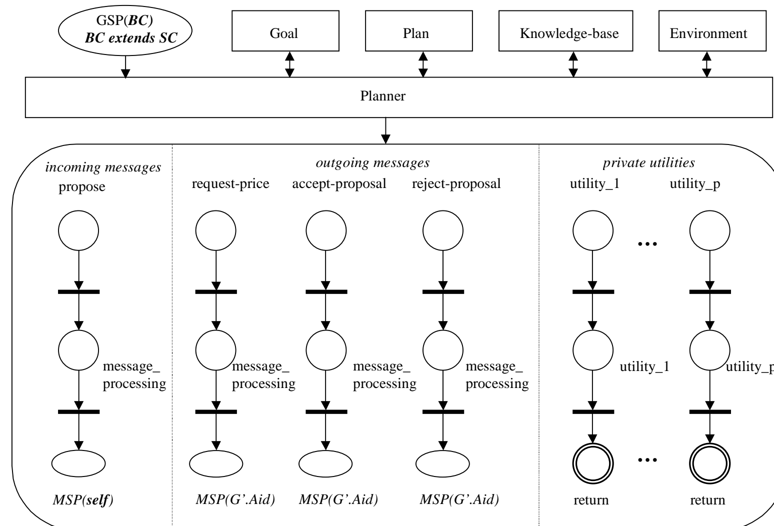


Figure 14. An agent-based G-Net model for buying agent class (BC)

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 12 (b), we may design the buying agent class as in Figure 14. 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 **external** is deposited into place *dispatch_outgoing_message*. The new *mTkn* token has the message name *supply-info*, following the protocol defined in Figure 12 (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 *external* 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 *mTkn* token will be transferred to the *GSP* of the receiver *mTkn.body.msg.receiver*, and in this case, it is a facilitator agent object.

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 four different forms of inheritance that are commonly used, namely augment inheritance, restrictive inheritance, replacement inheritance and refinement inheritance. The usage of these four 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 Section 2.5 or [Xu and Shatz 2000].

4.4 Handling Multiple Inheritance in Agent-Oriented Models

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*. The reference of ***super*** then needs to be resolved. In this section, we propose a modified breadth-first-search algorithm to find the appropriate reference of ***super***. The algorilthm is based on the hierarchy of inheritance diagram and the *MPU/Methods* defined in each agent-oriented G-net. Before presenting our algorithm, we need the following definitions:

Definition 4.2 *Parent Set P(s)*

Let *s* be an agent-oriented G-net, the *parent set*, *P(s),* is a set of agent-oriented G-nets, where each of the elements is a superclass of *s*.

Definition 4.3 *Interface Set Interface(s)*

Let s be an agent-oriented G-net, the *interface set*, *Interface(s),* is a set of *MPU/methods* defined in G-net *s*.

Definition 4.4 *Class Hierarchy Graph G*

A *class hierarchy graph* G=(V, E) is a formal description of the hierarchy of inheritance diagram. The class hierarchy graph *G* is a directed acyclic graph G=(V, E), where V is a set of nodes of agent-oriented G-nets, and E is a set of arcs denotes the inheritance relationship.

The breadth-first-search algorithm is so named because it discovers all the vertices at distance *k* from *s* before processing any vertices at distance *k*+1. To keep track of progress, the breadth-first-search algorithm colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is ***processed*** the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been processed, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. In addition, we assume that we have the following data structures: the color of each vertex $u \in$ V is stored in the variable *color*[*u*], and a first-in, first-out queue *Q* is used to manage the set of gray vertices. The algorithm is presented as follows:

1. **for** each vertex $u \in V - \{s\}$
2. **do** *color*[*u*] ← WHITE
3. *color*[*s*] ← GRAY
4. Q ← {*s*}
5. **while** $Q \neq \phi$
6. **do** *u* ← *head*[*Q*]
7. **for** each $v \in P(u)$
8. **do if** *color*[*v*] = WHITE
9. **then if** *mTkn.body.msg.name* ∈ *Interface*(*v*)
10. **then** ***super*** ← *v*; **return** *true*
11. **else** *color*[*v*] ← GRAY; ENQUEUE(*Q*,*v*)
12. DEQUEUE(*Q*)
13. *color*[*u*] ← BLACK
14. **return** *false*

If a *true* value returns, a *MPU/Method* is discovered, and the *mTkn* can be directly deposited into the *GSP* of ***super***; otherwise, the *MPU/Method* can not be found and an exception occurs. As stated before, we do not consider such exceptions in this paper. Note that this algorithm works correctly for both single and multi-level inheritance, and it has the advantage that the message token can be deposited directly to the appropriate *GSP* of a subagent without going through possible intermediate subagents.

Since a class can have more than one superclass (with multiple inheritance), the inheritance hierarchy has the structure of a directed acyclic graph rather than a tree or forest. In this case, ambiguous or conflicting inheritance can occur. The three issues that must be dealt with are as follows:

- *Name conflict*: two or more ancestors of a class might have messages with the same name, or state variables with the same name and type.
- *Repeated inheritance*: When a class *A* inherits from two superclasses that share a common ancestor, there are two copies of the same ancestor class. In class *A*, the usage of state variables and *MPUs/methods* defined in the common ancestor class is ambiguous.
- *Dominance problem*: When a class *A* inherits from two superclasses that share a common ancestor, and if a *MPU/method* defined in the common ancestor class is redefined by one of its superclasses, the reference of this *MPU/method* in the subclass *A* is ambiguous.

For the name conflict problem, we usually use a qualified name to solve the problem. For instance, if both a selling agent class *SAC* and a buying agent class *BAC* defines *MPU/method m_1*, the intended

message/method called in a retailer agent class *RAC* must be referred to as *SAC::m_1* or *BAC::m_1*, unless *m_1* is redefined in *RAC*. For the repeated inheritance problem, we assume that only one copy of the common ancestor class is maintained. Therefore, if a state variable or *MPU/method* defined in a common ancestor of superclasses of class *A* is referenced, it is always meant to the unique one. Finally, for the dominance problem, we assume that a redefined *MPU/method* has a dominance over the original one. Obviously, our modified breadth-first-search algorithm correctly enforces this rule of dominance.

4.5 Discussion

Multi-agent systems (MAS) have become one of the most rapidly growing areas of interest for distributed computing. Although there are several implementations of MAS available, formal frameworks for such systems are few [Brazier *et al.* 1998][Rogers *et al.* 2000]. In this chapter, 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 functional units defined in an agent superclass.

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 [Deng and Chang 1990][Murata *et al.* 1991a][Murata *et al.* 1991b]. We will also look further into issues like deadlock avoidance and state exploration problems in the agent-oriented design and verification processes.

Chapter 5

Analysis of Agent-Oriented Models

5.1 Introduction

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. In this section, we use a Petri tool, called INA (Integrated Net Analyzer) [Roch and Starke 1999], 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 P/T nets [Roch and Starke 1999]. These methods include analysis of: (1) structural properties, such as structural boundedness, T- and P-invariant analysis; (2) behavioral properties, such as boundedness, safeness, liveness, 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 properties verification and model checking.

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

The interaction between a buying agent and two selling agents can be modeled as a net as in Figure 15. To derive this net model, we use a *GSP* place to represent each selling agent. This is practical because an agent-oriented G-net model can be abstracted as a single *GSP* place, and agent models can only interact with each other through *GSP* places. Meanwhile, for the buying agent, whose class is a subclass of a shopping agent class, we simplify it 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 [Deng *et al.* 1993].

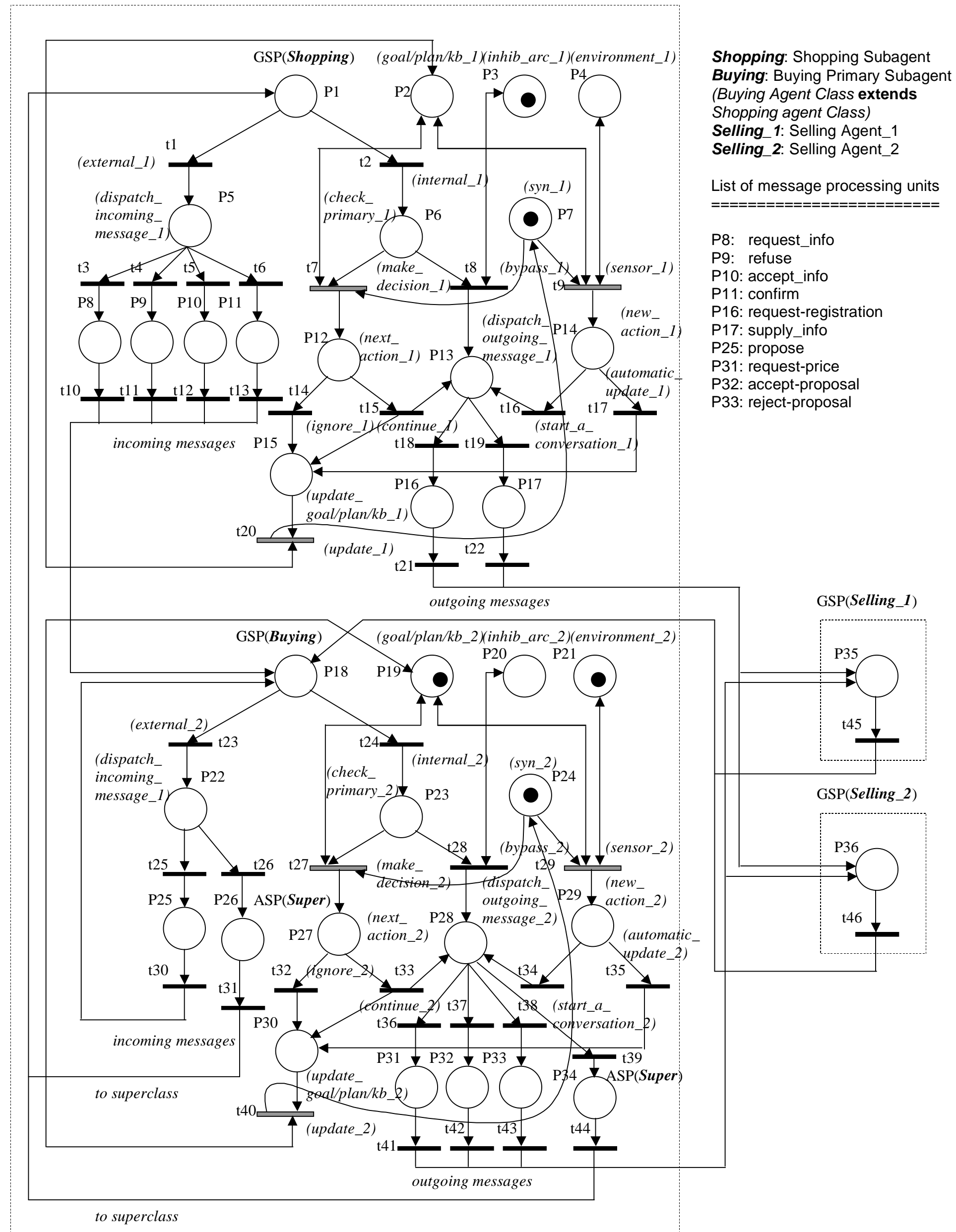


Figure 15. A transformed model of one buying agent and two selling agents

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 5.4).
4. We use net reduction (i.e., net transformation rules [Shatz *et al.* 1996]) to simplify the Petri net corresponding to an *MPU/Method* as a single place. For instance, the *MPU* identified as *propose* in Figure 14 is represented as place *P25* in Figure 15.
5. We use the closed-world assumption and consider a system that only contains three agents, i.e., a buying agent and two selling agents. A system contains more than three agents can be verified in the same way.

5.3 Deadlock Detection and Redesign of Agent-Oriented Models

Now we use the INA tool to analyze the simplified agent model illustrated in Figure 15. 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 [Shatz *et al.* 1996] 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.

To verify the correctness of our agent model, we utilize some key definitions for Petri net behavior properties as adapted from [Murata 1989].

Definition 5.1 *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 5.2 *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 5.3 *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 5.4 *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.

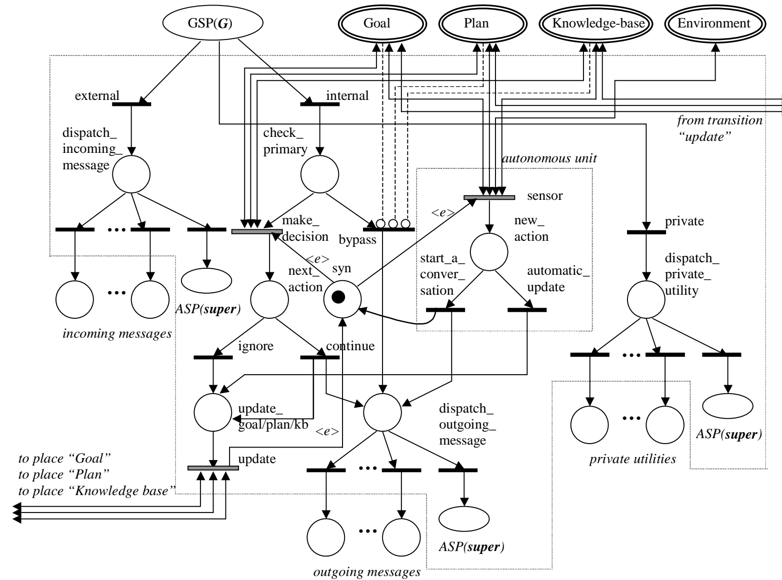


Figure 16. A template for the *Planner* module (revised design)

To correct this error, we need to modify the design of the *Planner* module in Figure 10. The model modification is to add a new arc from transition *start_a_conversation* to place *syn*, and the correct version of our *Planner* module design is shown as in Figure 16. Correspondingly, we add two new arcs in Figure 15: 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.

5.4 Property Verification by Using 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 [Clarke *et al.* 1986] [Clark and Wing 1996].

The INA tool allows us to state properties in the form of CTL formulae [Roch and Starke 1999][Clarke *et al.* 1986]. 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, 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, 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, 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, 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.

5.5 Discussion

In this Chapter, we 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*. In addition, some model checking techniques were used to prove some additional behavioral properties for our agent 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 detailed design, where the abstract transitions in the planner module are refined, we may again use Petri net tools to capture further design errors.

Chapter 6

Future Research Plans

6.1 Introduction

Communication among distributed processes is an essential requisite in nowadays computing systems. A communication paradigm represents the set of rules to be followed in exchanging data and synchronizing the execution of processes. The nature of currently available computing systems is pushing a lot towards a distributed approach which assumes that computing resources and data are no longer located on the same machine, and migration of code and data is executed in order to speed up the whole execution process. The classic client-server paradigm assumes that the client functionalities are somehow disjoint from the execution power of the server. Since a server usually provides service to a large number of clients, the amount of data exchanged may be considerable. Therefore, the work of the server is usually limited to the mere execution of some basic procedures for the data retrieval and storage, while the data processing mainly takes place on the client host. This type of scheme is used when we want to create a very simple system from the management point of view, or structures with a high level of security. An advantage of this architecture is the possibility of controlling the type of message and the ways of communication between clients and servers. In other words, the server only deal with what is expected during the design phase. Consequently, the level of security is very high. Since clients and servers can be viewed as passive objects, the object-oriented paradigm provides the best framework for developing a client-server application.

Unlike the client-server paradigm, a multi-agent system consists of a set of agents, i.e., active objects [Shoham 1993]. Agents usually do not communicate with each other in a way of method invocation, instead, an agent can send meaningful messages, possibly attached with a piece of code, to another agent. The receiver agent may analyze the received message, execute the attached code and decide whether to perform the requested actions. Therefore, an agent must be able to deal with messages that might not be expected during the design phase, and the communication mechanism among agents should be in a way of asynchronous message passing. This way of communication is similar to the remote evaluation (REV) paradigm [Stamos and Gifford 1990], which implies that server receives not only the processing requests

from the client, but also the whole code needed for performing operations on the data. However, agent communication in a multi-agent system is more flexible and more complicated than the server-client interactions in REV paradigm. Although we may use object-oriented approach to design a multi-agent system, it may complicate the design process while we dealing with asynchronous message passing mechanism and those mental decisions of agents. Thus an agent-oriented approach, such as the one we proposed, is necessary to be used to design a multi-agent system.

A third communication paradigm is the mobile agent paradigm. As one of the new agent techniques, mobile agent is becoming a promising paradigm pertinent to the highly distributed, dynamic, heterogeneous and open environment, such as Internet. Mobile agents are autonomous agents that can migrate around a computer network, and execute at different locations during their life spans. A mobile agent consumes fewer network resources in that they transfer the computation to the data rather than the data to the computation, which is adopted by traditional distributed computing. As one of our future plans, we will try to extend our agent-oriented G-net models for mobile agent modeling.

6.2 A Unified Model for Object-Oriented and Agent-Oriented Design

Internet is becoming the most complex environment that provides an open, dynamic and heterogeneous environment for large distributed systems. An Internet application, such as an electronic commerce application, usually consists of a set of both objects and agents. In those situations, an object usually works as a server and provides services to various clients (including agents), while agents may communicate with each other and negotiate to achieve their own goals. Therefore, a unified model for both object modeling and agent modeling might be useful for this type of applications. Based on our previous work, we will try to unify our object model and agent model, and provide a uniform framework for Internet application designs.

The basic idea behind this unified model is to provide both synchronous and asynchronous message passing mechanisms for a distributed system. In a complex software system with both objects and agents, objects usually use synchronous message passing to communicate with each other, while agents communicated with each other by using asynchronous message passing. Moreover, an agent could be a client of an object server, and it may also use synchronous message passing to get services from an object. Synchronous message passing in a form of method invocation is more efficient than asynchronous message passing, however it is not flexible enough for agent communications. Therefore, to provide both mechanisms for a complex software system design is not only a research issue, but also could be a practical attempt.

6.3 Extending Agent-Oriented G-net Model for Mobile Agent Design

In a broad sense, a software agent is any program that acts on behalf of a (human) user, just as different types of agents (e.g., travel agents, insurance agents, secretaries) represent other people in day-to-day transactions in the real world. A mobile agent then is a program which represents a user in a computer network, and is capable of migrating autonomously (under its own control) from node to node in the network, to perform some computation on behalf of the user. Its tasks are determined by the agent application, and can range from online shopping to real-time device control to distributed scientific computing. Applications can inject mobile agents into a network, allowing them to roam the network either on a predetermined path, or one that the agents themselves determine based on dynamically gathered information. Having accomplished their goals, the agents may either terminate or return to their "home site" in order to report their results to the user.

Harrison et al. identified several advantages of the mobile agent paradigm, in comparison with remote procedure calls (RPC) [Tay and Ananda 1990] and message-passing. These advantages include: reduce network usage, increase asynchrony between clients and servers, add client-specified functionality to servers, dynamically update server interfaces and introduce concurrency [Harrison et al. 1995].

The mobile agent paradigm can be exploited in a variety of ways, ranging from low-level system administration to middleware to user-level applications. An example of such application could be an electronic marketplace. Vendors can set up online shops with products, services or information for sale. A customer's agent would carry a shopping list along with a set of preferences, visit various sellers, find the best deal based on the preferences, and purchase the product using digital forms of cash. This application imposes a broad spectrum of requirements on mobile agent systems. Apart from mobility, it needs mechanisms for restricted resource access, secure electronic commerce, protection of agent data, robustness and user control over roving agents. For our future work, we will try to extend our agent-oriented model for agent mobility modeling. This work will be based on previous work [Picco *et al.* 1999][Roman *et al.* 1997][Asperti and Busi 1996][Fan and Xu 2000].

6.4 Security Issues in Mobile Agent Design

Messages sent across an open network like the Internet are inherently insecure. As a mobile agent traverses the network, its code and data are vulnerable to various types of security threats. We consider the following types of attacks on communication links that the system needs to protect against [Ford 1994]:

Passive attacks: In passive attacks, the adversary does not interfere with the message traffic, but only attempts to extract useful information from it. The simplest form of such attack is *eavesdropping*, which

can result in the leakage of sensitive information stored in the message (agent) being transmitted. Even if the adversary is unable to decipher the message contents (because of encryption, for example), useful information may be gleaned from the sizes and frequency of message exchanged, or merely the fact that two principles are in communication. This type of passive attack is usually called *traffic analysis* in the security literature. To counter passive attacks, a *confidentiality* (i.e., privacy) mechanism is therefore necessary.

Active attacks: In the case of open networks like the Internet, we must assume a very general threat model in which the adversary can arbitrarily intercept and modify network-level message, or even delete them altogether and insert forged ones. These are termed as active attacks, since they involve active interference by the adversary. Another type of attack in this category involves *impersonation*, The adversary impersonates one of the legitimate principals in the system and can attempt to intercept messages intended for that principal. Active attacks require greater sophistication on the part of the adversary, but can also be more dangerous than passive attacks. While we can not always prevent all such attacks, the damage caused by them can be minimized if the communication link provides assurances of *data integrity* and *authentication*. Here *data integrity* means that data is either delivered unmodified or a flag is raised to signal if it has been tampered with, and *authentication* requires that the source and destination of the message is unambiguously identified.

If time permitted, we will try to model mobile agents and hostile agents with our agent framework. Our purpose is to study the different forms of attack and to verify that some mobile agent design might be vulnerable to some types of attack. The advantages of this research will be the automated verification of an agent design by using existing Petri net tools.

Bibliography

[Aalst and Basten 1997] W.M.P. van der Aalst and T. Basten, "Life-cycle Inheritance: A Petri-net-based approach," In P. Azema and G. Balbo , editors, *Application and Theory of Petri Nets 1997*, volume 1248 of Lecture Notes in Computer Science, pages 62--81. Springer-Verlag, Berlin, 1997.

[Arai *et al.* 1999] S. Arai, K. Miyazaki, and S. Kobayashi, "Multi-agent Reinforcement Learning for Crane Control Problem: Designing Rewards for Conflict Resolution," *Proceedings of 4th International Symposium on Autonomous Decentralized Systems (ISADS '99)*, Tokyo, Japan, 20-23 March 1999.

[Asperti and Busi 1996] Andrea Asperti and Nadi Busi, "Mobile Petri Nets," Technical Report UBLCS-96-10, University of Bologna, 1996.

[Bastide 1995] R. Bastide, "Approaches in Unifying Petri Nets and the Object-Oriented Approach," *Proceedings of the International Workshop on Object-Oriented Programming and Models of Concurrency*, Turin, Italy, June 1995.

[Basten and Aalst 2000] T. Basten and W.M.P. van der Aalst, "Inheritance of Dynamic Behavior: Development of a Groupware Editor," In G. Agha, F. De Cindo, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, Advances in Petri Nets, Springer-Verlag, Berlin, 2000.

[Battiston *et al.* 1988] E. Battiston, F. De Cindio and G. Mauri, "OBJSA Nets: a Class of High Level Nets Having Objects as Domains", in *Advances in Petri Nets 88*, G. Rozenberg (ed.), LNCS 340, Springer Verlag, 1988.

[Battiston et al. 1995] E. Battiston, A. Chizzoni and F. De Cindio, "Inheritance and Concurrency in CLOWN," *16th International Conference on Application and Theory of Petri nets, 1st Workshop on Object-Oriented Programming and Models of Concurrency*, Turin, Italy, June 1995.

[Battiston *et al.* 1996] E. Battiston, A.Chizzoni and F. De Cindio, "Modeling a Cooperative Development Environment with CLOWN," *17th Int'l Conference on Application and Theory of Petri nets, 2nd Workshop on Object-Oriented Programming and Models of Concurrency*, Osaka, Japan, June 1996.

[Biberstein *et al.* 1996] O. Biberstein, D. Buchs and N. Guelfi, "Modeling of Cooperative Editors Using CO-OPN/2," *17th International Conference on Application and Theory of Petri nets, 2nd Workshop on Object-Oriented Programming and Models of Concurrency*, Osaka, Japan, June 1996.

[Biberstein *et al.* 1997] O. Biberstein, D. Buchs and N. Guelfi, "CO-OPN/2: A Concurrent Object-Oriented Formalism," *Proceedings of the Second IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Canterbury, UK, July 1997, pp. 57-72.

[Booch 1994]G. Booch, *Object-Oriented Analysis and Design, with Applications* (2nd ed.), Benjamin/Cummings, San Mateo, California, 1994.

[Brazier *et al.* 1997] 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.

[Brazier *et al.* 1998] F. Brazier, F. Cornelissen, R. Gustavsson, C. Jonker, O. Lindeberg, B. Polak, and J. Treur, "Agents Negotiating for Load Balancing of Electricity Use," In: M.P. Papazoglou, M. Takizawa, B. Krämer, S. Chanson (eds.), *Proceedings of the 18th International Conference on Distributed Computing Systems*, ICDCS'98, IEEE Computer Society Press, 1998, pp. 622-629.

[Burmeister 1996] Birgit Burmeister, "Models and Methodology for Agent-Oriented Analysis and Design," In K. Fischer, editor, *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*, DFKI Document D-96-06, 1996.

[Chavez and Maes 1996] Anthony Chavez, Pattie Maes, "Kasbah: An Agent Marketplace for Buying and Selling Goods," *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, April 1996.

[Clark and Wing 1996] 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.

[Clarke *et al.* 1986] 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*, 8(2), 1986, pp. 244-263.

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

[Deng *et al.* 1993] 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.

[Deng and Chang 1990] 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.

[Drake 1998] Caleb Drake, *Object-oriented programming with C++ and Smalltalk*. Upper Saddle River, New Jersey, Prentice Hall, 1998.

[Eliens 1995] A. Eliens, *Principles of Object-Oriented Software Development*, Addison-Wesley, 1995.

[Fan and Xu 2000] X. Fan and D. Xu, "SAFIN: An Open Framework for Mobile Agents," *The 2000 International Conference on Artificial Intelligence (IC-AI'2000)*, Las Vegas, June 2000.

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

[FIPA 2000] FIPA, *FIPA ACL Message Structure Specification*, Foundation for Intelligent Physical Agents, Technical Report XC00061, 2000.

[Fisher and Wooldridge 1997] M. Fisher and M. Wooldridge, "On the Formal Specification and Verification of Multi-Agent Systems," *International Journal of Cooperative Information Systems*, 1(6): 37-65, 1997.

[Ford 1994] Warwick Ford, *Computer Communications Security – Principles, Standard Protocols and Techniques*, Prentice Hall, 1994.

[Gasser and Briot 1992] Les Gasser and Jean-Pierre Briot, "Object-Based Concurrent Processing and Distributed Artificial Intelligence," In Nicholas M. Avouris and Les Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 81-108, Kluwer Academic Publishers: Boston, MA, 1992.

[Giese *et al.* 1998] H. Giese, J. Graf and G. Wirtz, "Modeling Distributed Software Systems with Object Coordination Nets," *Proceedings for the Int'l Symposium on Software Engineering for Parallel and Distributed Systems*, Japan, April 1998, pp.39-49.

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

[Guttman *et al.* 1998] R. Guttman, A. Moukas, and P. Maes, "Agent-mediated Electronic Commerce: A Survey," *Knowledge Engineering Review*, June 1998.

[Harrison *et al.* 1995] Colin G. Harrison, David M. Chess and Aaron Kershenbaum, "Mobile Agents: Are They a Good Idea?" *Tech. Rep.*, IBM Research Division, T. J. Watson Research Center, March 1995.

[Iglesias *et al.* 1998] 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.

[Jensen 1992] K. Jensen, *Colored Petri Nets: Basic concepts, Analysis methods, and Practical use*, Vol. 1, No. 2, Springer-Verlag, 1992.

[Jacobson *et al.* 1992] I. Jacobson, et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Publishing Company, 1992.

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

[Jennings 2000] N. R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, 117(2000): 277-296.

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

[Kinny and Georgeff 1997] David Kinny, Michael P. Georgeff, “Modeling and Design of Multi-Agent Systems,” *Proceedings of the 4th Int’l Workshop on Agent Theories, Architectures, and Language (ATAL-97)*, 1997, pp. 1-20.

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

[Lakos and Keen 1994] C. Lakos and C. Keen, “LOOPN++: A New Language for Object-Oriented Petri Nets,” *Technical Report R94-4*, Networking Research Group, University of Tasmania, Australia, April 1994.

[Lakos 1995a] C. Lakos, “Pragmatic Inheritance Issues for Object Petri Nets”, *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) Pacific 1995*, Melbourne, Australia, Prentice-Hall, 1995.

[Lakos 1995b] C. Lakos, “The Object Orientation of Object Petri Nets,” *Proceedings of the International Workshop on Object-Oriented and Models of Concurrency*, Turin, Italy, June 1995.

[Lakos 1997] C. Lakos, “On the Abstraction of Coloured Petri Nets,” *Proceedings of Petri Net Conference 97*, Touloure, France, 1997.

[Lano 1995] K. Lano, *Formal Object-Oriented Development*, Springer-Verlag, 1995.

[Lee and Park 1993] Y. K. Lee and S. J. Park, “OPNets: An Object-Oriented High-Level Petri Net Model for Real-Time System Modeling,” *Journal of Systems and Software*, 20(1): 69-86, 1993.

[Luck et al. 1997] Michael Luck, Nathan Griffiths and Mark d’Inverno, “From Agent Theory to Agent Construction: A Case Study,” In J. P. Muller, M.Wooldridge and N. R. Jennings, editors, *Intelligent Agents III (LNAI 1193)*, Lecture Notes in Artificial Intelligence, Springer-Verlag: Heidelberg, Germany, 1997.

[Matsuoka and Yonezawa 1993] Satoshi Matsuoka and Akinori Yonezawa, “Analysis of inheritance anomaly in object-oriented concurrent programming languages”. In Gul Agha et. al., editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107-150. MIT Press, 1993.

[Mendes *et al.* 1997] M. Mendes, O. Falsarella, I. Fontes, S. Krause, W. Loyolla, C. Mendez, P.S. Silva, C. Tobar, “Architectural Considerations about Open Distributed Agent Support Platforms,” *Proceedings of 3rd Int’l Symp. on Autonomous Decentralized Systems (ISADS '97)*, Berlin, Germany, April 1997.

[Mitchell and Wellings 1996] S. Mitchell and A. Wellings, “Synchronization, Concurrent Object-Oriented Programming and the Inheritance Anomaly”, *Computer Languages*, 1996, Vol. 22, No. 1, pp. 15 - 26.

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

[Murata *et al.* 1991a] 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.

[Murata *et al.* 1991b] 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.

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

[Perkusich and de Figueiredo 1997] 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.

[Picco *et al.* 1999] G. P. Picco, A. L. Murphy and G.-C. Roman, "Lime: Linda meets Mobility," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

[Pressman 1997] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, McGraw-Hill, 1997.

[Rational 1997] Rational Software Corporation, *Unified Modeling Language (UML) version 1.0*, Rational Software Corporation, 1997.

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

[Rogers *et al.* 2000] T. J. Rogers, Robert Ross, V. S. Subrahmanian, “IMPACT: A System for Building Agent Applications,” *Journal of Intelligent Information Systems (JIIS),* 14(2-3): 95-113 (2000).

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

[Rossie *et al.* 1996] J. G. Rossie Jr., D. P. Friedman and M. Wand, “Modeling Subobject-Based Inheritance”, *Proceedings of ECOOP’96*, Vol. 1219, Lecture Notes in Computer Science, pp. 248-274, Springer-Verlag, 1996.

[Rumbaugh *et al.* 1991] J. Rumbaugh, et al., *Object-Oriented Modeling and Design*, Prentice Hall, New York, 1991.

[Shatz *et al.* 1996] 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.

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

[Sommervile 1995] Ian Sommervile, *Software Engineering*, Fifth Edition, Addison-Wesley, 1995.

[Stamos and Gifford 1990] James W. Stamos and David K. Gifford, "Remote Evaluation," *ACM Transactions on Programming Languages and Systems*, 12(4): 537-565, October 1990.

[Stepney *et al.* 1992] Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, Springer-Verlag, 1992.

[Tay and Ananda 1990] B. H. Tay and A. L. Ananda, "A Survey of Remote Procedure Calls," *Operating Systems Review*, 24(3): 68-79, July 1990.

[Thomas 1994] Laurent Thomas, "Inheritance Anomaly in True Concurrent Object Oriented Languages: A Proposal", *IEEE TENCON'94*, August 1994, pp. 541-545.

[Tsvetovatyy *et al.* 1997] M. Tsvetovatyy, M. Gini, B. Mobasher, Z. Wieckowski, "MAGMA: An Agent-Based Virtual Market for Electronic Commerce," *Applied Artificial Intelligence,* special issue on Intelligent Agents, No. 6, September 1997.

[Wooldridge 1998] Michael Wooldridge, "Agents and Software Engineering," *AI*IA Notizie XI*, 3, September 1998.

[Wooldridge *et al.* 2000] 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.

[Xie 2000] X. Xie, *Design Support for State-Based Distributed Object Software*, Ph.D. thesis, EECS Department, The University of Illinois at Chicago, December 2000.

[Xu and Shatz 2000] H. Xu and S. M. Shatz, "Extending G-nets to Support Inheritance Modeling in Concurrent Object-Oriented Design," *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, October 2000, Nashville, Tennessee, USA, pp. 3128-3133.

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

[Xu and Shatz 2001b] 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.

Publications of the Author

[1] H. Xu and S. M. Shatz, "An Approach to Using Formal Methods in Agent-Oriented Design and Analysis," (submitted to journal), January 2001.

[2] H. Xu and S. M. Shatz, "A Framework for Modeling Agent-Oriented Software," To appear in the *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, April 16-19, 2001, Phoenix, Arizona, USA.

[3] H. Xu and S. M. Shatz, "An Agent-based Petri Net Model with Application to Seller/Buyer Design in Electronic Commerce," To appear in the *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, March 26-28, 2001, Dallas, Texas, USA.

[4] 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.

[5] R. K. Gedela, S. M. Shatz and H. Xu, "Compositional Petri Net Models of Advanced Tasking in Ada-95," *Computer Languages*, July 1999, Vol.25, No.2, pp.55-87.

[6] R. K. Gedela, S. M. Shatz and H. Xu, "Formal Modeling of Synchronization Methods for Concurrent Objects in Ada 95," *Proceedings of the ACM Annual International Conference on Ada (SIGAda'99)*, October 17-21, 1999, Redondo Beach, CA, USA, pp. 211-220.

[7] K. Warendorf, H. Xu, and A. Verhoeven, "Case-based Instructional Planning for Learning in a Context," *Proceedings of PACES/SPICIS 97* (24-27 February 1997), Singapore, pp. 354-360.

[8] H. Xu, X. Ruan, Z. Chen, S. Hu and H. Ren, "Hypertext and Multi-knowledge Source Based ICTS (Intelligent Chinese Tutoring System)," *Journal of Chinese Information Processing*, 1992, Vol. 6, No. 2, pp.8-16.

[9] Q. Hu, H. Xu, Y. Zhang and C. Zhou, "Software Design of an Expert Control System in Vaccum Distillation," *Control and Instruments in Chemical Industry*, 1992, Vol. 19, No. 4, pp.25-29.

[10] X. Ruan, S. Hu, Z. Chen and H. Xu, "The Presentation and Inference of Chinese Language Knowledge," *Proceedings of the International Conference on "Information & System"*, A.M.S.E., October 1991, Hangzhou, China.

[11] H. Xu, Z. Chen, and S. Hu, "Design and Implementation Techniques for an Intelligent Chinese Tutoring System," *Proceedings of the Second National Conference on Computer Application*, October 1991, Beijing, China, pp. 988-991.

[12] H. Xu, "Software Design of an Microcomputer-based Nuclear Scaler," *Process Automation Instrumentation*, 1991, Vol. 12, No. 10, pp.13-16.

Curriculum Vitae

Haiping Xu was born in Pinghu, a coastal city of Zhejiang, China. He got his early education in his hometown, and skipped two grades in primary school. As a gifted child, Haiping skipped one more grade in senior high school and was admitted to the "Juvenile Class" of Zhejiang University in April 1985, when he was only 15 years old. In July 1989, Haiping Xu got his B.S. degree in Electrical Engineering.

Owing to Haiping's excellent performance in his undergraduate study, in 1989, he was admitted directly to the Graduate School of Zhejiang University without entrance examination. Haiping's research area was Intelligent Computer Aided Instruction (ICAI), and he worked on a project called "Intelligent Chinese Language Tutoring System" in the Artificial Intelligence Lab at Computer Science Department. In March 1992, Haiping Xu not only got his M.S. degree, but also was awarded the diploma and medal of "Excellent Graduate Student of Zhejiang University".

After graduated from Zhejiang University, Haiping Xu was employed as a software engineer in the Ministry of Electronics Industry in Beijing, China. Then from June 1993 to May 1996, he successively worked as a senior engineer in Shenyan Company and Hewlett-Packard Company in Beijing, China.

In May 1996, Haiping Xu went to Singapore and worked there as a short-term research scholar in the Intelligent Systems Lab at Nanyang Technological University. During the four months there, Haiping built an intelligent tutoring system prototype for the computer science course "Data Structures and Algorithms" and published an international conference paper.

In August 1996, Haiping Xu came to the United States, and began to fulfill his unfinished ambition in his graduate study. Haiping was awarded a research assistantship from the Computer Science and Engineering Department at Wright State University, and worked in the area of parallel and distributed computing. In 1998, due to his excellent performance in his graduate study, Haiping Xu was awarded the Dayton Area Graduate Studies Institute (DAGSI) Scholarship and got his M.S. degree in computer science.

In August 1998, Haiping decided to continue his Ph.D. study at the University of Illinois at Chicago. Now he is a Ph.D. candidate in the Electrical Engineering and Computer Science Department at UIC. During the two and a half years study at UIC, Haiping has published four conference papers and one journal paper. Currently, Haiping's research interest is to apply Petri net formalism to software development, with applications to electronic commerce and Internet security.