A MODEL-BASED APPROACH FOR DEVELOPMENT OF MULTI-AGENT SOFTWARE SYSTEMS

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

THESIS

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, 2003

Chicago, Illinois

This thesis is dedicated to

my dear parents, Nianxiang and Ming,

without whom it would never have been accomplished.

ACKNOWLEDGMENTS

I would like to acknowledge many people for helping me during my doctoral work. I would especially like to thank my advisor, Dr. Sol M. Shatz, for his generous time and commitment. Throughout my doctoral work he encouraged me to develop independent thinking and research skills. He continually stimulated my analytical thinking and greatly assisted me with scientific writing. Dr. Shatz has been a great source of encouragement and inspiration to me. Without his support this dissertation would not have been written.

I am also very grateful for having an exceptional thesis committee and wish to thank Dr. Ugo Buy, Dr. Tadao Murata, Dr. Peter Nelson and Dr. Aris Ouksel for their unwavering support and assistance. I would like to thank Dr. Jeffrey Tsai, who was one of the committee members for my Ph.D. preliminary examination, and had provided valuable suggestions for my research directions.

I owe a special note of gratitude to Dr. Prabhaker Mateti at the Computer Science and Engineering Department at Wright State University, who led me into the area of distributed computing before I joined the Computer Science Department at the University of Illinois at Chicago. He has always been a great support and encouragement to my Ph.D. study.

I extend many thanks to my colleagues and friends at the Concurrent Software Systems Laboratory (CSSL), especially Dr. Ajay Kshemkalyani, Dr. Prasad Sistla, Dr. Xiande Xie, Zhaoxia Hu, Yan Pan, and Chaoyue Xiong. Each of them has contributed valuable insights and suggestions.

Finally, I thank the Department of Computer Science at the University of Illinois at Chicago for giving me this opportunity and providing an environment to do research.

HX

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

ADK	Agent Development Kit
AOSE	Agent-Oriented Software Engineering
ASM	Abstract Superclass Module
ASP	Asynchronous Superclass switch Place
AUML	Agent Unified Modeling Language
AVM	Agent Virtual Machine
AW	Agent World
BB	Bounded Buffer class
BDI	Belief, Desire and Intention
BMA	Buying Mobile Agent class
CLOWN	Class Orientation With Nets
CO-OPN/2	Cocurrent Object-Oriented Petri Nets
CTL	Computation Tree Logic
DAI	Distributed Artificial Intelligence
DP	Default Place
EP	Entry Place
FIPA	Foundation for Intelligent Physical Agents
GSP	Generic Switch Place
IFA	Intelligent Facilitator Agent
IMA	Intelligent Mobile Agent
INA	Integrated Net Analyzer
IS	Internal Structure
ISP	Instantiated Switch Place
KE	Knowledge Engineering

LIST OF ABBREVIATIONS (continued)

LOOPN++	Language for Object-Oriented Petri Nets
MA	Mobile Agent
MAS	Multi-Agent System
MPU	Message Processing Unit
MSP	Message Switch Place
OBCP	Object-Based Concurrent Programming
OMT	Object Modeling Technique
OO	Object-Oriented
OOPN	Object-Oriented Petri Nets
OOSE	Object-Oriented Software Engineering
OPN	Object Petri Nets
PM	Planner Module
PN	Petri Nets
Pr/T Nets	Predicate/Transition Nets
P/T Nets	Place/Transition Nets
RP	Return Place
SBOPN	State-Based Object Petri Nets
SM	Synchronization Module
SMA	Selling Mobile Agent class
SSP	Superclass Switch Place
UB	Unbounded Buffer class
U-Method	Utility Method
UML	Unified Modeling Language

SUMMARY

The advent of multi-agent systems has brought opportunities for the development of complex software that will serve as the infrastructure for advanced distributed applications. During the past decade, there have been many agent architectures proposed for implementing agent-based systems, and also some efforts to formally specify agent behaviors. However, research on narrowing the gap between agent formal models and agent implementation is rare. In this thesis, we present a model-based approach to designing and implementing multi-agent software systems. Instead of using formal methods only for the purpose of specifying agent behavior, we bring formal methods into the design phase of the agent development life cycle. Our approach is based on the G-net formalism, which is a type of high-level Petri net defined to support modeling of a system as a set of independent and loosely-coupled modules.

We first introduce how to extend G-nets to support class modeling and inheritance modeling for concurrent object-oriented design. Then, by viewing an agent as an extension of an object with mental states, we derive an agent-oriented G-net model from our extended G-nets that support class modeling. The agent-oriented G-net model serves as a high-level design for intelligent agents in multi-agent systems. To illustrate our formal modeling technique for agent-oriented software, an example of an agent family in electronic commerce is provided. We show how an existing Petri net tool can be used to detect design errors, and how model checking techniques can support the verification of some key behavioral properties of our agent models. In addition, we adapt the agent-oriented G-net model to support basic mobility concepts, and present design models of intelligent mobile agents. Finally, based on the high-level design, we derive the agent architecture and the detailed design needed for agent implementation. To demonstrate the feasibility of our approach, we describe a toolkit called ADK (Agent Development Kit) that supports rapid development of application-specific agents for multi-agent systems.

1. INTRODUCTION

1.1 Background and Motivations

The development of software systems starts with two main activities, namely software requirements analysis and software design [Sommerville 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 document, which serves as a contract between the software designers and the programmers.

The purpose of software requirements analysis can be achieved in two ways. One is to specify and analyze systems formally, and the other is to describe and model systems naturally. Conventionally, software requirements specifications are written in natural languages, e.g., English. However, when specifying, modeling and analyzing the behavior of a critical and complex system, choosing a specification language that can formally depict the properties of the system is preferred. 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, and as a result, a number of formal specification languages and tools have been developed as a replacement for natural languages specification techniques. 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. Further efforts to enhance/extend the theory and techniques of Petri nets, including high-level Petri nets such as CPN (Colored Petri Nets) [Jensen 1992], have also been devoted to make formal methods more useful in industry/commercial software development.

Although formal methods have been widely used in specifying and verifying complex software systems [Clark and Wing 1996], to bridge the gap between formal models and implemented systems is still a big challenge. Formal methods have been frequently adopted in the requirements analysis phase to specify a system and its desired properties, e.g., behavioral properties; however, to create formal models in the design phase and therefore verify their correctness is rare. This is not only because of the infancy of the techniques and the apparent difficulty of the notations used, but also due to a lack of support for modularization in most of the formal approaches, e.g., the temporal logic [Manna and Pnueli 1992]. Meanwhile, software design is the technical kernel of software engineering, and to develop critical and complex software systems not only requires a complete, consistent and unambiguous specification, but also a correct design that meets certain requirements. This observation has motivated our initial work of using formal methods in concurrent object-oriented design, and further derived our model-based approach for development of agent-oriented software systems.

On the other hand, in both academic and industrial histories, 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, bringing with it new concepts such as data encapsulation, inheritance, messaging and polymorphism. By the end of the eighties and the 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 (Object Modeling Technique) [Rumbaugh *et al.* 1991], and OOSE (Object-Oriented Software Engineering) [Jacobson *et al.* 1992], became 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 software 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, many concepts of the agent-oriented paradigm are still not mature, and the methodology, especially the techniques for agent modeling in practical use, is yet to be improved.

To provide a practical agent-oriented methodology for agent development, we view an agent as an extension of an object, i.e., an active object [Shoham 1993], and propose a model-based approach for development of multi-agent software systems. Instead of using formal methods only for the purpose of specifying agent behavior, we bring formal methods into the design phase of the agent development life cycle. Our approach is based on the G-net formalism [Deng *et al.* 1993] [Perkusich and de Figueiredo 1997], which is a type of high-level Petri net defined to support modeling of a system as a set of independent and loosely-coupled modules. We select Petri nets as our base model because Petri net models are a mature graph-based model with intuitively appealing rules for defining the structure and dynamic behavior of general systems with concurrent components.

1.2 Related Work

1.2.1 Object-Oriented Petri Nets

The concepts of the 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 contemporary software systems, object-oriented software designers began to understand the usefulness of formal methods. Along with this trend, object-oriented formal methods have become one of the 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 net 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 [Deng *et al.* 1993][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]. An OBJSA net can be viewed as a semantics model described by algebraic notations; while 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 interface for interactions between a CLOWN object and other objects, and the `inherits` clause defines the inheritance features.

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 the Petri net 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 suggested 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: the `Signature` part is used to describe the interface with other classes, and the `Body` part is used 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 types in order to reuse its type defined in other classes, and the methods declared in the `Signature` part are used as interface transitions. The weakness of the CO-OPN/2 approach is that the unfolding mechanism for a CO-OPN/2 specification is not suggested; therefore the analysis and simulation method for CO-OPN/2 is not explicitly defined.

C. Lakos proposed 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 a text-based grammar to specify systems. In a specification of LOOPN++, the class definition consists of three parts: `Fields` to define data, `Functions` to describe expressions with parameters and operations, 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, and it becomes a base to support the abstraction of nets. The super place and super transition can be defined by labeling 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++, this approach has some deficiencies in fully supporting the object-oriented concepts. For instance, 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]. In addition, although a LOOPN++ program can be used to simulate a system, using existing Petri net tools for system analysis is not supported.

G-nets [Deng *et al.* 1993][Perkusich and de Figueiredo1997] support the concepts of objects better than CO-OPN/2 or LOOPN++ in terms of simplicity of expressing modularity and information hiding. 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 *RP* (Return Place) to end a method execution. These features can be unfolded into Pr/T (Predicate/Transition) nets for system analysis [Deng *et al.* 1993].

A fascinating feature of G-nets is its support for encapsulation of objects, synchronous 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 invocations. Also, the mechanism for a 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, it does not support inheritance mechanism. In addition, it is difficult to represent an 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: the OPNets [Lee and Park 1993] that focus on the decoupling of inter-object communication knowledge and the separation of synchronization constraints from the internal structure of objects; and the OCoNs (Object Coordination Nets) [Giese *et al.* 1998], which are used to describe the coordination of class behaviors on a service. Although these formalisms support 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 analysis or simulation methods.

1.2.2 Formal Methods in Agent-Oriented Software Engineering

Agent technology has received a great deal of attention in the past few years and, as a result, industry is becoming interested in using this technology to develop its own products. In spite of the different developed agent theories, languages, architectures and successful agent-based applications, very little work has been aimed at specifying agent architectures and creating design techniques to develop agent-based applications using agent technology [Iglesias *et al.* 1998]. The role of agent-oriented methodologies is to assist all the phases of the development life cycle for 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].

There are three main strands of work to which our research is related, i.e., work on formal modeling of agent systems, work on building practical agent-based systems or developing tool kits for rapid development of agent systems, and work on narrowing the gap between agent formal models and implementation of agent-based systems.

Previous work on formal modeling of agent systems has been based on formalisms, such as Z [Davies and Woodcock 1996], temporal logic [Manna and Pnueli 1992], and Petri nets [Murata 1989], to specify agent systems or agent behaviors. Luck and d'Inverno tried to use the formal language Z to provide a framework for describing the agent architecture at different levels of abstraction. They proposed a four-tiered hierarchy comprising entities, objects, agents and autonomous agents [Luck and d'Inverno 1995]. The basic idea for this is that all components of the world are entities with attributes. Of these entities, objects are entities with capabilities of actions, agents are objects with goals, and autonomous agents are agents with motivations. Fisher used temporal logic to represent dynamic agent behavior [Fisher 1995]. Such a temporal logic is more powerful than the corresponding classic logic and is useful for the description of dynamic behavior in reactive systems. Fisher took the view that a multi-agent system is simply a system consisting of concurrently executing objects. Xu and his colleagues used Predicate/Transition (Pr/T) nets, which is a high-level formalism of Petri net, to model and verify multi-agent behaviors [Xu *et al.* 2002]. Based on the Pr/T model, certain properties, such as parallel execution of multi-plans and guarantee for the achievement of a goal, can be verified by analyzing the dependency relations among the transitions. More recently, Pr/T nets were used to model logical agent mobility [Xu *et* *al.* 2003]. The proposed model for logical agent mobility specifies a mobile agent system that consists of a set of components and a set of (external) connectors. Pr/T nets were used because in a Pr/T net a token may carry structured data – the mobility modeling was based on the idea of "agent nets" being able to be routed, as tokens, within "system nets." Other efforts on formal modeling of agents focus on the design of modeling languages for conceptual design and specification of multi-agent systems. For instance, the modeling language DESIRE (framework for Design and Specification of Interacting Reasoning component) is based on the philosophy of viewing a complex software system as a series of interacting components; therefore it is suited to the specification of multi-agent systems [Brazier *et al.* 1997]. Similarly, SLABS (formal Specification Language for Agent-Based Systems) provides a way of specifying agent behaviors to enable software engineers to analyze agent-based systems before they are implemented [Zhu 2001].

In summary, formal methods are typically used for specification of agent systems and agent behaviors. The primary purpose of the resulting formal agent models is to define *what* properties are to be realized by the agent system, e.g., behavioral properties. In contrast, the formal agent model that we present in Chapter 4 provides a high-level design of multi-agent software systems [Xu and Shatz 2003] – it not only provides a conceptual framework for agent development, but it also aids a software engineer in understanding *how* to structure and implement an agent system. This is accomplished by explicitly identifying the major components and mechanisms in the design and showing how to derive a detailed design and corresponding implementation. A direct benefit of this approach is that it brings formal methods into the design phase, providing opportunities for formal verification of correctness of an agent design. We also show ways of using analysis techniques, including model checking, to verify the correctness and key properties of the formal agent model in Chapter 5 [Xu and Shatz 2003]. Ideally, formal methods can be applied in each phase of a software development life cycle; however, to bring formal methods into the later phases (e.g., design and implementation) of a software development life cycle is not an easy task. For instance, Rao and Georgeff presented an algorithm for model checking BDI systems [Rao and Georgeff 1993]; however, since there is no clear relationship between the BDI logic and the concrete computational models used to implement agents, it is not clear how such a model can be derived [Wooldridge and

Ciancarini 2001]. Thanks to Petri nets' graphical modeling approach and its similarity with the UML modeling technique [Saldhana *et al.* 2001], we argue that Petri nets provide a reasonable way of bringing formal methods into the design phase. With refinement of our original agent-oriented G-net models in further detailed design [Yan 2002], our approach supports formal design of agent-oriented software.

A second strand of related work is the development of practical agent-based systems or tools for rapid development of agent systems. During recent years, many agent architectures have been proposed. For instance, JAM (Java Agent Model) is a hybrid intelligent agent architecture that draws upon the theories and ideas of the Procedural Reasoning System (PRS), Structured Circuit Semantics (SCS), and Act plan interlingua [Huber 1999]. Based on the BDI theories [Kinny *et al.* 1996], which models the concepts of beliefs, goals (desires), and intentions of an agent, JAM provides strong goal-achievement syntax and semantics, with support for homeostatic goals and a much richer, more expressive set of procedural constructs. The JACK (Java Agent Kernel) intelligent agent framework proposed by the *Agent Oriented Software Group* brings the concept of intelligent agents into the mainstream of commercial software engineering and Java technology [Howden *et al.* 2001]. It is designed as a set of lightweight components with high performance and strong data typing. Paradima has been implemented to support the development of agent-based systems [Ashri and Luck 2000]. It relies on a formal agent framework, i.e., Luck and d'Inverno's formal agent framework [Luck and d'Inverno 1995], and is implemented by using recent advances in Java technology. Although the above agent architectures use formal agent models as conceptual guidelines, the formal methods serve as agent specifications rather than formal designs.

Some other efforts had tried to provide a rapid prototyping development environment for the construction and deployment of agent-oriented applications. A typical example is the Zeus MAS (Multi-Agent System) framework developed by British Telecom labs [Nwana *et al.* 1999]. The MAS development environment based on Zeus MAS framework consists of an API (Application Programming Interface), code generator, agent and society monitoring tools, and programming documentation. A complete Zeus agent has a coordination engineer enabling functional behavior organized around conversation protocols, a planner that schedules sub-goal resolution, an engine for rule-based behavior, and databases to manage resources, abilities, relationships between agents, tasks, and protocols. More recently, many agent frameworks have been proposed for developing agent applications in compliance with the FIPA (Foundation for Intelligent Physical Agents) specifications [FIPA 2000] for interoperable intelligent agents in multi-agent systems. Examples of such efforts are JADE (Java Agent Development Framework) [Bellifemine *et al.* 1999], FIPA-OS (FIPA Open Source) agent platform [Poslad *et al.* 2000], and the current Zeus platform [Nwana *et al.* 1999]. The major difference between the above work and our approach is that most of the existing agent architectures attempt to provide a comprehensive set of agent-wide services that can be utilized by application programmers; however, these services are usually made available through an ad-hoc architecture that is highly coupled. Application programmers must face a steep learning curve for such systems due to a lack of explicit control flow and modularization. In contrast, our approach provides programmers a set of loosely coupled modules, an explicit control flow, and a clean interface among agents. We believe that our approach can significantly flatten a programmer’s learning curve, and ease the workload for developing application-specific agents. Another difference between the above works and our approach is that most of the agent architectures originated from industry aim to provide practical platforms or toolkits for agent development; therefore, unlike our approach there is not the direct motivation for an agent design that supports formal analysis and verification. Meanwhile, most of the existing systems use object-oriented languages, such as Java, but without considering how to use object-oriented mechanisms effectively in developing agent-oriented software. In contrast, our approach carefully considers the role of inheritance in agent-oriented development, and discusses which components of an agent could be reused in a subclass agent. This treatment of inheritance in agent-oriented software engineering is based on previous work [Crnogorac *et al.* 1997], but our approach emphasizes on reuse of functional components rather than mental states. Finally, to demonstrate the feasibility of our approach, we developed the toolkit called ADK (Agent Development Kit) that supports rapid development of intelligent agents for multi-agent systems. Although our current version of ADK does not strictly follow the FIPA specifications, we have designed our agent model with standardization in mind. Further work on this prototype has shown that it is fairly straightforward to extend our agent design and development kit to a level of detail that is compliant with the FIPA specifications [Yan 2002].

Previous efforts on narrowing the sizable gap between agent formal models and agent-based practical systems can be summarized as follows. Some researchers aimed at constructing directly executable formal agent models. For instance, Fisher's work on Concurrent METATEM has attempted to use temporal logic to represent individual agent behaviors where the representations can be executed directly, verified with respect to logical requirements, or transformed into some refined representation [Fisher 1995]. Vasconcelos and his colleagues have tried to provide a design pattern for skeleton-based agent development [Vasconcelos *et al.* 2002], which can be automatically extracted from a given electronic institution. The electronic institutions have been proposed as a formalism with which one can specify open agent organizations [Rodriguez-Aguilar *et al.* 1999]. These types of work seem to be an ideal way for seaming the gap between theories and implemented systems; however, an implementation automatically derived from a formal model tends to be not practical. This is because a formal model is an abstraction of a real system, and thus an executable formal model ignores most of the components and behaviors of a specific agent. Therefore, as stated in a survey paper [D'Inverno *et al.* 1997], executable models based on formalisms, such as temporal logic, are quite distant from agents that have actually been implemented. Other efforts have attempted to start with specific deployed systems and provide formal analyses of them. For instance, d'Inverno and Luck tried to move backwards to link the system specification based on a simplified version of dMARS (distributed Multi-Agent Reasoning System) to the conceptual formal agent framework in Z, and also to provide a means of comparing and evaluating implemented and deployed agent systems [D'Inverno and Luck 2001].

In contrast to the above approaches, we have tried to bring formal methods directly into the design phase, and to let the formal agent model serve as a high-level design for agent implementation. In particular, we use the agent-oriented G-net model to define the agent structure, agent behavior, and agent functionality for intelligent agents. A key concept in our work is that the agent-oriented G-net model itself serves as a design model for an agent implementation. We will see that our architectural design of intelligent agents closely follows the agent-oriented G-net model. By supporting design reuse, our approach also follows the basic philosophy of *Model Driven Architecture (MDA)* [Siegel *et al.* 2001] that is gaining popularity in many communities, for example UML.

1.3 Contributions of Our Work

The work reported in this thesis is aimed at proposing a technique for modeling and analyzing object-oriented and agent-oriented software systems, and attempting to bridge the gap between formal agent models and agent implementation. 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 can be listed as follows:

1. Extended the original G-net model to support class modeling and inheritance modeling, and proposed a formal model – extended G-nets, for concurrent object-oriented design (Chapter 2).
2. Proposed an agent-based G-net model, and proved the properties of *L3-liveness*, *concurrency* and *effectiveness* for agent communication (Chapter 3).
3. Proposed an agent-oriented G-net model by introducing an inheritance mechanism into the agent-based G-net model. Used an example of agent family in electronic commerce to show how agent-oriented software systems can be designed (Chapter 4).
4. Performed experiments with an existing Petri net tool to detect design errors, and used model checking techniques to verify some key properties of agent-oriented models (Chapter 5).
5. Adapted the agent-oriented G-net model to support basic mobility concepts, and presented design models for intelligent mobile agents (Chapter 6).
6. Derived an agent design architecture and a detailed design needed for agent implmentation from the agent-oriented G-net model. Developed an agent dvelopment kit (ADK) that facilitates development of application-specfic agents in multi-agent systems (Chapter 7).

2. A FORMAL MODEL FOR CONCURRENT OBJECT-ORIENTED DESIGN

2.1 Introduction

One of the key issues in object-oriented (OO) approaches 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 with the Petri net (PN) theory, the question arises how inheritance may be supported by such formalisms, 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, e.g., Concurrent Smalltalk, 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, to directly use existing Petri net tools to verify behavioral properties of a subclass object is not supported. Little work has been done so far 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 formalisms are either too theoretical to be used in practical software design, or too preliminary to cover all forms of inheritance including refinement inheritance.

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 [Deng *et al.* 1993][Perkusich and de Figueiredo 1997], we first extend G-nets into a 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 behavioral property analysis, e.g., to analyze the inheritance anomaly problem.

2.2 <u>G-Net Model Background</u>

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 [Deng *et al.* 1993][Perkusich and de Figueiredo 1997]. 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 the original G-net models.

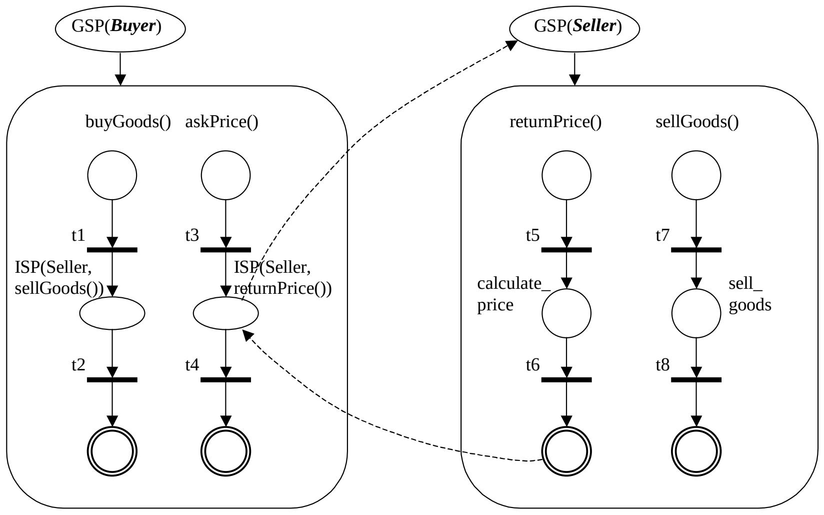


Figure 1. G-net model of buyer and seller objects

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 *GSP* (Generic Switch Place) and an *IS* (Internal Structure). 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 a design of the module. An example of G-nets is shown in Figure 1. Here the G-net models represent two objects – a *Buyer* and a *Seller*. The generic switch places are represented by *GSP(Buyer)* and *GSP(Seller)* enclosed by ellipses, and the internal structures of these models are represented by round-cornered rectangles that contain four methods: *buyGoods()*, *askPrice()*, *returnPrice()* and *sellGoods()*. The functionality of these methods is 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* that defines the 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 *ISP (Instantiated Switch Place)*. 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 and variable declarations for simplicity.

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. To support modeling object-oriented software, we first need to extend the G-net model to support class modeling [Xu and Shatz 2000]. The idea of this extension is to generate a unique object identifier, *G.Oid*, and initialize the state variables defined in *G.AS* 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 [Deng *et al.* 1993][Perkusich and de Figueiredo 1997]. The only difference is that we allow two types of tokens, namely *sTkn* tokens and *mTkn* tokens. An *sTkn* token is a colored or colorless token used in synchronous modules, which we will introduce shortly. An *mTkn* token is a message token deifned as 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, m1(para_list))* in its thread/process with process id of *G.Pid* to a G-net object *G'_obj* with *G'.Oid*, the procedure for updating an message token *mTkn* is as follows:

1. Call_before: *mTkn.seq* ← *mTkn.seq* + <*G.Oid, G.Pid, m1*>; *mTkn.mtd* ← *(m1, para_list, NULL)*; *mTkn.sc* ← **before**.
2. Transfer the *mTkn* token to the *GSP* place of the called G-net object *G'_obj* with *G'.Oid*.
3. Wait for the result to be stored in *mTkn.mtd.result*, and the *mTkn* token to be returned.
4. Call_after: *mTkn.seq* ← *mTkn.seq* – *LAST(mTkn.seq)*; *mTkn.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 into 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* 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*; 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 (EP)*, *instantiated switch place (ISP)* and *return place (RP)*. Each method can have only one *EP* and one *RP*, 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-{*RP*}) x T) ∪ ((T x (P-{*EP*}).

2.4 Extending G-Nets to Support Inheritance

Figure 2 shows an example of G-net model that 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 design of four methods: *isEmpty(), put(e), get()* and *who()*. The functionalities of these methods are defined as follows: *isEmpty()* checks if the buffer is empty and returns 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.

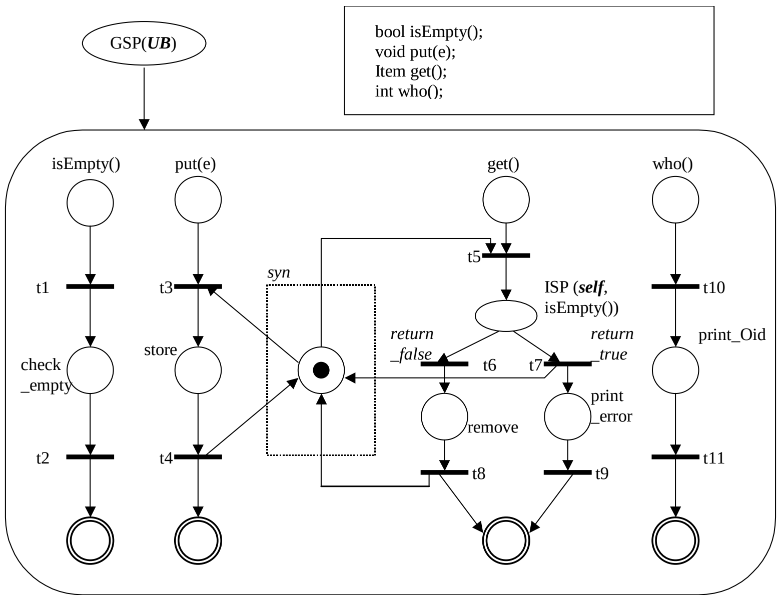


Figure 2. G-net model of unbounded buffer class (UB)

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*.

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 (DP)*. 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 *mTkn* 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)*.

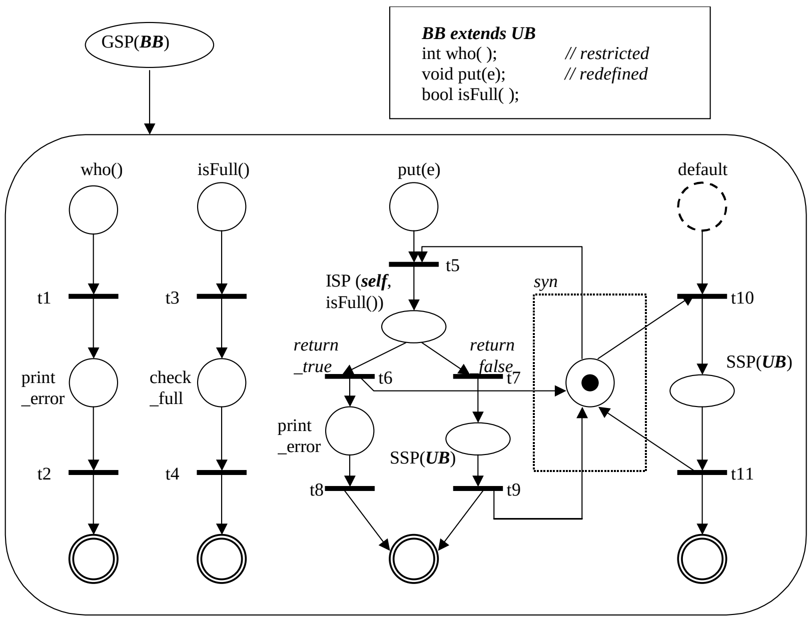


Figure 3. G-net model of bounded buffer class (BB)

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 *mTkn* 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 *mTkn.msg.result* and the *mTkn* 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 (SM)* and *Abstract Superclass Module (ASM)*. Based on the formal definitions of *standard G-net* model (Section 2.3), 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 (SM)*

A *Synchronization Module (SM)* is 4-tuples (P, A, I, O), where P is a set only containing a single place that is used to hold an *sTkn* token - a colored or 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 (ASM)*

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

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 <u>Modeling Inheritance Anomaly Problem</u>

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 they 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.

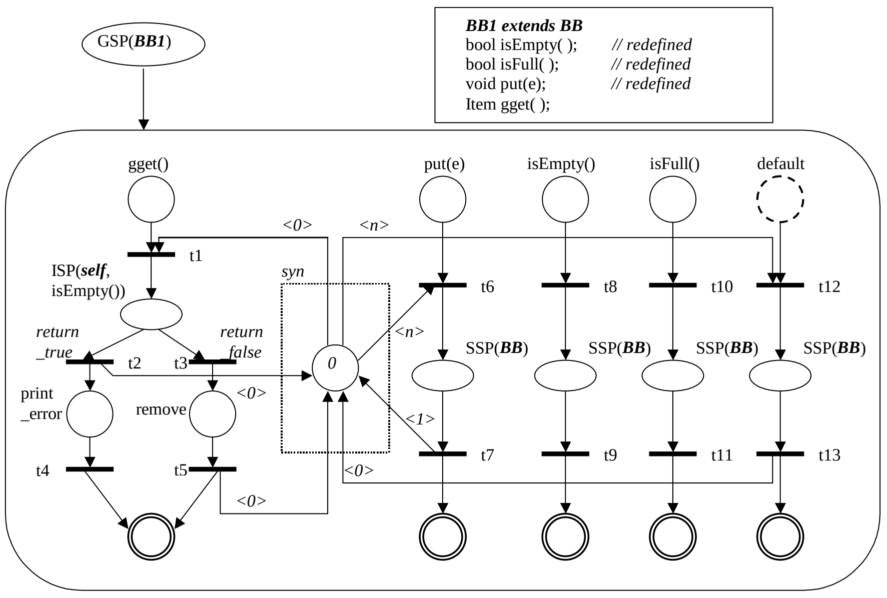


Figure 4. G-net model of bounded buffer class (BB1)

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 cannot 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.

2.7 Summary

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. In contrast, 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 this chapter, we presented a formal model, called *extended G-net* model, that can be used to model different forms of inheritance for concurrent object-oriented design. The formal model is derived from the *standard G-net* model by introducing an inheritance mechanism. As an example of net-based analysis, we investigated the inheritance anomaly problem in concurrent object-oriented design. In Chapter 3, we will introduce a new concept – multi-agent systems (MAS), and by incorporating additional modules into the *standard G-nets*, we derive an *agent-based G-net* model that supports agent modeling.

3. FROM OBJECT TO AGENT: AN AGENT-BASED G-NET MODEL

3.1 Introduction

The term "agent" comes from the Greek word "agein", which means to drive or to lead. Today, the term "agent" is used to describe something that can produce an effect, e.g., a "drying agent" or a "shipping agent". In computer science, an "agent" denotes a computer system that is situated in some *environment* and is capable of *autonomous* actions, e.g., a software agent that can search and buy air tickets over the Internet. In this thesis, we always use the word "agent" to refer to "software agent".

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 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, and 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:

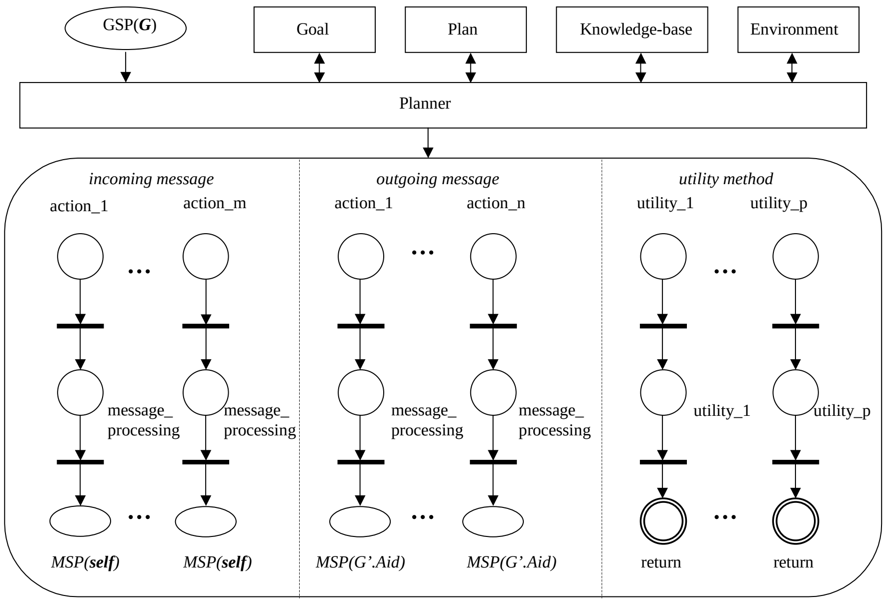
1. Agents that form a multi-agent system may be developed by different vendors independently, and those agents may be widely distributed across large-scale networks such as the Internet. To make it possible for those agents to communicate with each other, it is desirable for them to have a common communication language and to follow common protocols. However the standard G-net model does not directly support protocol-based language communication between agents.
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 as discussed in Chapter 2. 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

[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.

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. Each plan is associated with a goal or a subgoal; however a goal/subgoal may associated with one or more than one plans, and the most suitable one will be selected to achieve that goal/subgoal. 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.



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

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

In the *Planner* module, committed goals or subgoals can be 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. Thus, the *Planner* module can be viewed as the heart of an agent that may decide to ignore an incoming message, to start a new conversation, or to continue with the current conversation based on the agent's mental state.

The *IS* (Internal Structure) of an agent-based G-net consists of three sections: *incoming message*, *outgoing message*, and *utility method*. 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 *utility method* section. Unlike with the methods defined in a standard G-net model, the methods defined in the *utility method* 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 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.

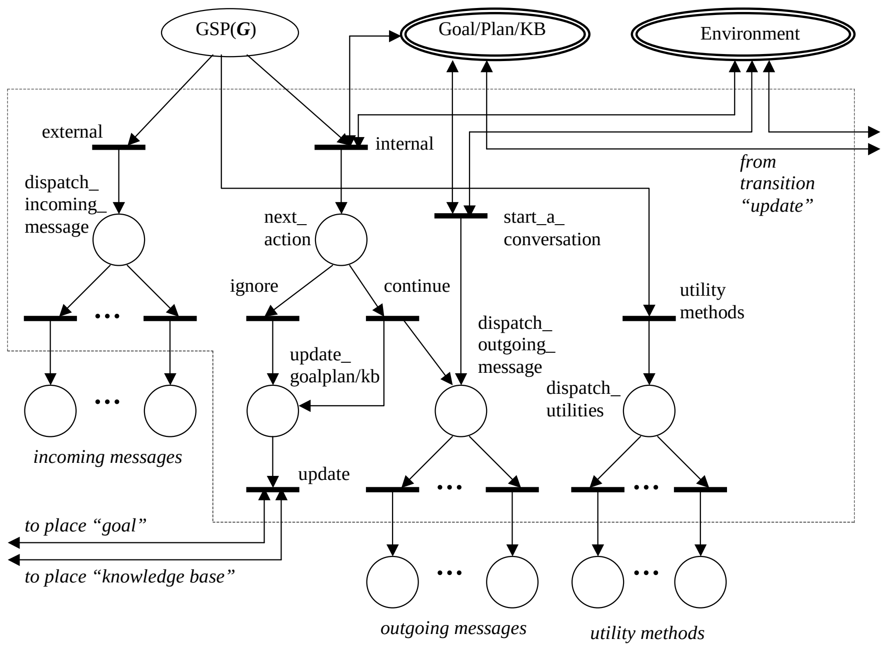


Figure 6. A template of the *Planner* module

The *Planner* module represents the heart of an agent. It 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 utility methods 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 *utility method* section.

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 colored or colorless token *sTkn* defined in optional synchronization modules, 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* tokens is to make them 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 chapter.

An *mTkn* token (originally deifned in Section 2.3) is redefined as a 2-tuple *(tag, body)*, where *tag* $\in$ {**internal**, **external**, **method**} 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, method};

struct MtdInvocation {
    Triple (seq, sc, mtd);  // as defined in Section 2.3
}

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. Two variables of *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 **method**.
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 [Deng *et al.* 1993] [Perkusich and de Figueiredo 1997].
3. After all the *ISP* method calls are finished and the *mTkn* token returns to the original *ISP*, the *mTkn.tag* is set back as *mTkn.body.old_tag*, and both the variables *old_tag* and *miv* are detached.

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

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

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

Definition 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 (PM)*

A *Planner Module (PM)* 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 *utility method* 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 (EP), instantiated switch place (ISP)* and *message switch place (MSP)*. Each *MPU* has only one *EP* 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-{*EP*}).

Definition 3.5 *Utility Method (U-Method)*

A *Utility Method (U-Method)* or a *Method* is a triple (P, T, A), where P is a set of places with three special places: *entry place (EP), instantiated switch place (ISP)* and *return place (RP)*. Each method has only one *EP* and one *RP*, 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-{*RP*}) x T) ∪ ((T x (P-{*EP*}).

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 reference [Odell 2000]. Figure 7 (a) is a modified example of an FIPA contract net protocol, which depicts a protocol template 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.

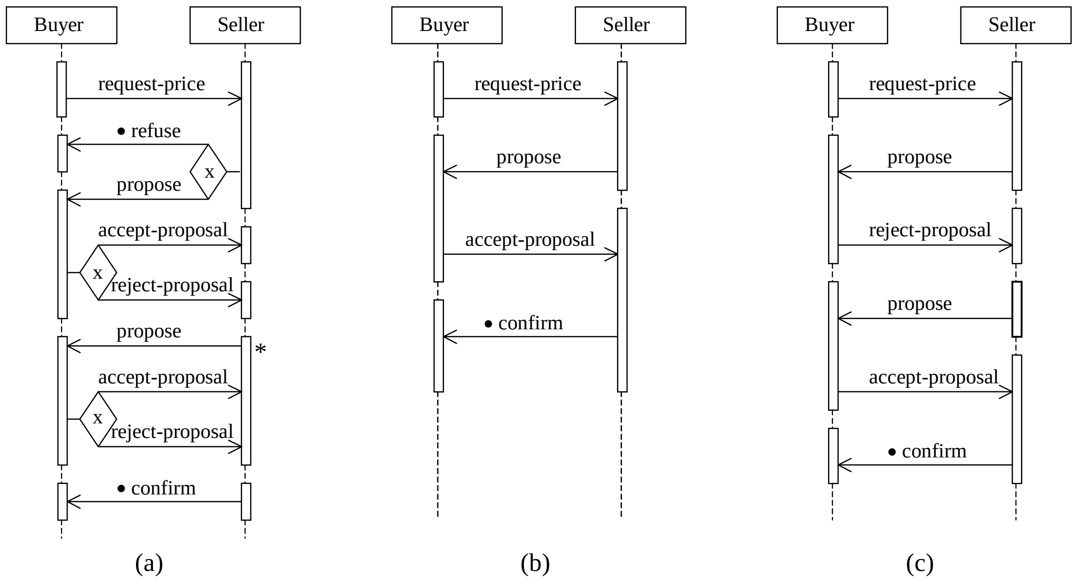
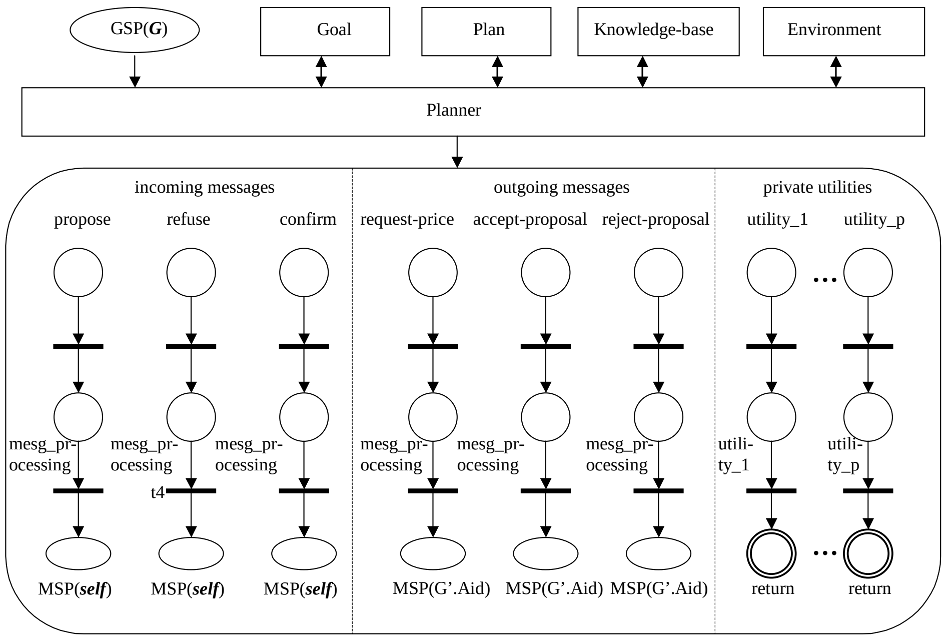


Figure 7. A contract net protocol between buying and selling agent

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*, *Plan* 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 *utility method* section, we may define some necessary functions that can be called by the buying agent itself. Examples of such utility methods could be: *compare_price*, *update_knowledge_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.



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

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, *Plan* module and *Knowledge-base* module as ordinary places with ordinary tokens; (2) omit *utility method* 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. Table I and Table II provide a legend that identifies the meaning associated with each place and transition 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.

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.

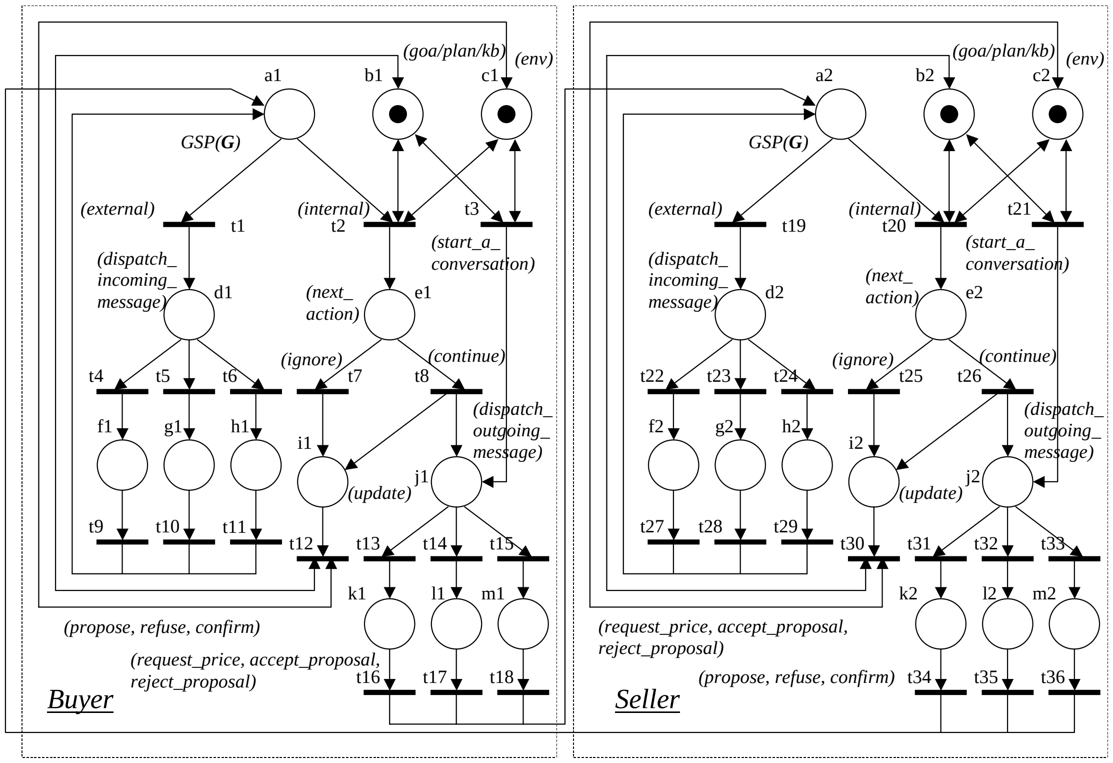


Figure 9. A transformed model of buying and selling agents

Table I

LEGEND FOR FIGURE 9 (DESCRIPTION OF PLACES)

Place	Description
a1 / a2	The *GSP* place of the buying / selling agent.
b1 / b2	The merged place for the *Goal*, *Plan* and *Knowledge-base* module of the buying / selling agent.
c1 / c2	The place for the *Environment* module of the buying / selling agent.
d1 / d2	The place for dispatching incoming messages.
e1 / e2	The place for choosing the next action: to ignore or to continue with the current conversation.
f1 / g1 / h1	The place for the message processing unit (*MPU*) of *propose* / *refuse* / *confirm*.
i1 / i2	The place for updating the agent mental state.
j1 / j2	The place for dispatching outgoing messages.
k1 / l1 / m1	The place for the message processing unit (*MPU*) of *request-price* / *accept-proposal* / *reject-proposal*.
f2 / g2 / h2	The place for the message processing unit (*MPU*) of *request-price* / *accept-proposal* / *reject-proposal*.
k2 / l2 /m2	The place for the message processing unit (*MPU*) of *propose* / *refuse* / *confirm*.

Table II

LEGEND FOR FIGURE 9 (DESCRIPTION OF TRANSITIONS)

Transition	Description
t1 / t19	The transition *external*, which fires when the token from the *GSP* has a tag of **external**.
t2 / t20	The transition *internal*, which fires when the token from the *GSP* has a tag of **internal**.
t3 / t21	The transition *start_a_conversation* that starts a new conversation.
t4, t9	Transitions related to the message processing unit (*MPU*) of *propose*.
t5, t10	Transitions related to the message processing unit (*MPU*) of *refuse*.
t6, t11	Transitions related to the message processing unit (*MPU*) of *confirm*.
t7 / t25	The transition *ignore* that ignores the current conversation.
t8 / t26	The transition *continue* that continues with the current conversation.
t12 / t30	The transition *update_goal/plan/kb*, which updates the agent's mental state.
t13, t16	Transitions related to the message processing unit (*MPU*) of *request-price*.
t14 / t17	Transitions related to the message processing unit (*MPU*) of *accept-proposal*.
t15 / t18	Transitions related to the message processing unit (*MPU*) of *reject-proposal*.
t22 / t27	Transitions related to the message processing unit (*MPU*) of *request-price*.
t23 / t28	Transitions related to the message processing unit (*MPU*) of *accept-proposal*.
t24, t29	Transitions related to the message processing unit (*MPU*) of *reject-proposal*.
t31, t34	Transitions related to the message processing unit (*MPU*) of *propose*.
t32, t35	Transitions related to the message processing unit (*MPU*) of *refuse*.
t33, t36	Transitions related to the message processing unit (*MPU*) of *confirm*.

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 3.10, Petri net *(N, M_0)* is *L3-live*. ◊

Table III

THE INCIDENCE MATRIX A OF THE PETRI NET IN FIGURE 9

	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

The incidence matrix A of the Petri net in Figure 9 is listed in Table III. 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 0 1 0 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 σ_1, σ_2, … σ_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, … x_5 define the firing count vectors for σ_1, σ_2, … σ_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

[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.

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 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 Summary

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. This observation motivated our work on modeling agent-based systems. In this chapter, we proposed an agent-based G-net model to support agent modeling. We used an example of price-negotiation protocol to show how agents in an electronic marketplace can be formally designed. By using some analysis technique – calculating the minimal support T-invariants of a transformed model of a buying agent and a selling agent, we further proved that our net model meets the requirements of *L3-live*, *concurrent* and *effective* properties.

In addition to a useful role in electrical commerce applications, MAS technology has also been applied in a wide range of realistic application domains, including human-computer interface, telecommunications, transportation systems and concurrent engineering [Jennings *et al.* 1998]. With more and more practical agent systems being built, it becomes an increasing need to provide formal methods in MAS specification and design to ensure robust and reliable products. In Chapter 4, we show how to introduce an inheritance mechanism into our *agent-based G-net* model and derive an *agent-oriented G-net* model for agent-oriented software design.

4. A FRAMEWORK FOR MODELING AGENT-ORIENTED SOFTWARE

4.1 Introduction

To avoid building a methodology from scratch, 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 knowledge 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 natural way for most of the software engineers. This is because there are similarities between the object-oriented paradigm and the agent-oriented (AO) 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 engineering 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 dissertation, 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 models 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 utility methods (*U-Methods*), 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 agent-oriented desugn.

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 *Asynchronous Superclass switch Place (ASP)*, and decision-making units, e.g., *abstract transitions*.

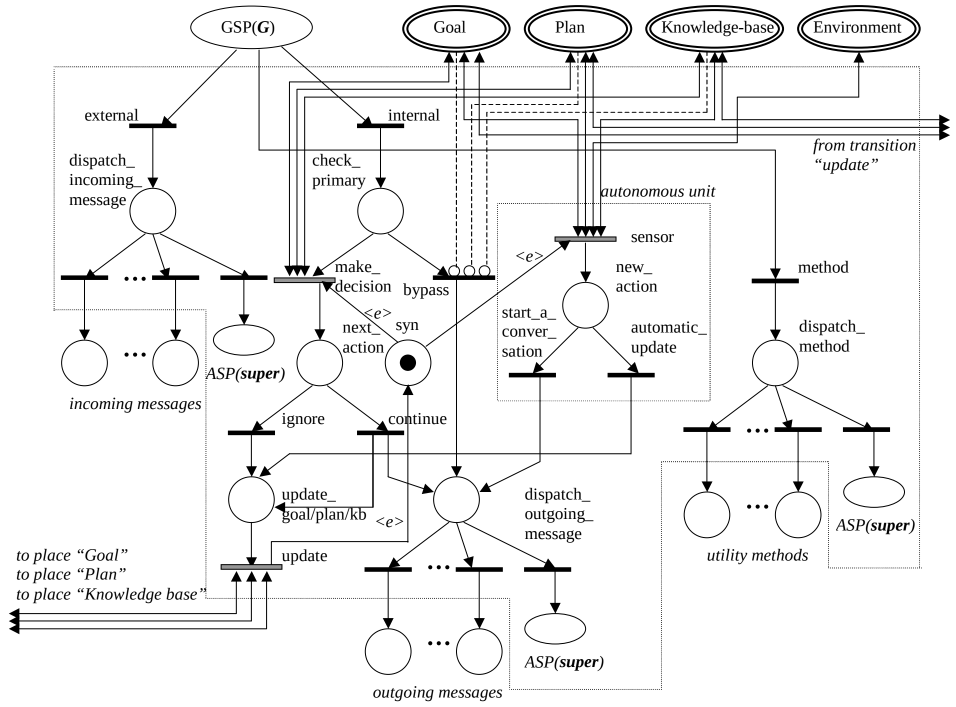


Figure 10. A template for the *Planner* module (initial design)

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 can be refined into correct sub-nets that capture action sequences specific to those activities; however how to make decisions and how to update an agent's mental state is beyond the scope of this dissertation, and will be considered in our future work. As a side

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

note, such work will provide a bridge to other work concerned with modeling agent mental states [Deng and Chang 1990][Murata *et al.* 1991a][Murata *et al.* 1991b].

There is also a newly defined unit in the *Planner* module 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 goals 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*.

Note that the *Planner* module is both goal-driven and event-driven because the transition *sensor* may fire when any committed goal is ready to be achieved or any new event happens. In addition, the *Planner* module is also message-triggered because certain actions may initiate whenever a message arrives (either from some other agent or from the agent itself). A message is represented as a message token with a tag of **internal**/**external**/**method**. A message token with a tag of **internal** represents a message forwarded by an agent to itself with the *MSP* mechanism, or a newly generated outgoing message before sending to some other agent; while a message token with a tag of **external** is an incoming message which comes from some other agent. In either case, the message token with the tag of **internal**/**external** should not be involved in an invocation of a method call. In contrast, a message token with a tag of **method** indicates that the token is currently involved in an invocation of some method call. When an incoming message/method arrives, with a tag of **external**/**method** in its corresponding token, it will be dispatched to the appropriate *MPU/method* defined in the internal structure of the agent. If it is a method invocation, the method defined in the *utility method* 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 an *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 **internal**, will be dispatched into the appropriate *MPU* in the *outgoing message* section of the internal structure of the agent. After the message is processed by the *MPU*, the message will be sent to a receiver agent by using the *MSP(Receiver)* mechanism, and the tag of the message token will be changed from **internal** to **external**, accordingly. In either case, a token will be deposited into place *update_goal/plan/kb*, allowing the abstract transition *update* to fire. As a consequence, the *Goal*, *Plan* and *Knowledge-base* modules are updated if needed, and the agent's mental state may change.

To ensure that all decisions are made upon the latest mental state of the agent, i.e., the latest values in the *Goal*, *Plan*, and *Knowledge-base* modules, and similarly to ensure that the sensor always captures the latest mental state of the agent, we introduce a synchronization unit *syn*, modeled as a place marked with an ordinary token (black token). The token in place *syn* will be removed when the abstract transition *make_decision* or *sensor* fires, thus delaying further firing of these two abstract transitions until completion of actions that update the values in the *Goal*, *Plan* and *Knowledge-base* modules. This mechanism is intended to guarantee the mutual exclusive execution of decision-making, capturing the latest mental state and events, and updating the mental state. Note that we have used the label <*e*> on each of the arcs connecting with the place *syn* to indicate that only ordinary tokens may be removed from or deposited into the place *syn*.

[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.

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 utility methods. There is existing work on agent inheritance with respect to knowledge, goals and plans [Kinny and Georgeff 1997][Crnogorac *et al.* 1997]. However, we argue 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 as a default reasoning mechanism, however this type of modeling is beyond the scope of this chapter. 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 chapter we only consider how to initialize a subclass agent's mental state while an agent subclass is instantiated; meanwhile, we focus on 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

[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 dissertation.

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 the reference of ***super***. In the case of single inheritance, ***super*** refers to a unique superclass G-net, however with multiple inheritance, the reference of ***super*** must be resolved by searching the class hierarchy diagram.

When a message/method is not defined in an agent subclass model, the dispatching mechanism will deposit the message token into a corresponding *ASP(**super**)*. Consequently, the message token will be forwarded to the *GSP* of a subagent, and it will be again dispatched. This process can be repeated until the root subagent is reached. In this case, if the message is still not defined at the root, an exception occurs. In this dissertation, 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 chapter, 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.

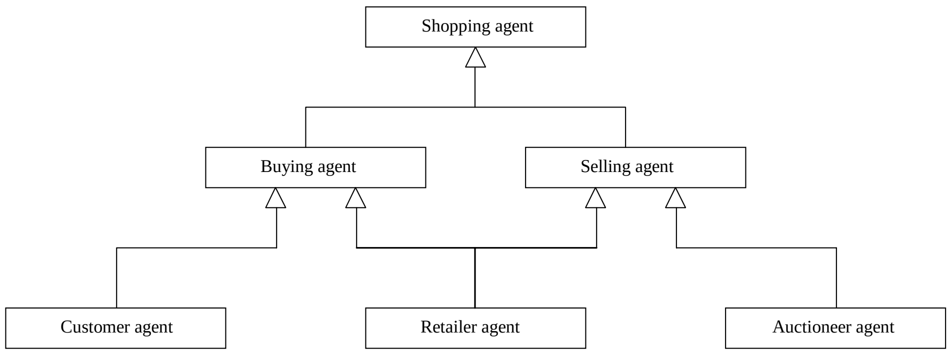


Figure 11. The class hierarchy diagram of agents in an electronic marketplace

4.3.2 <u>Modeling Agents in an Electronic Marketplace</u>

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 describes 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).

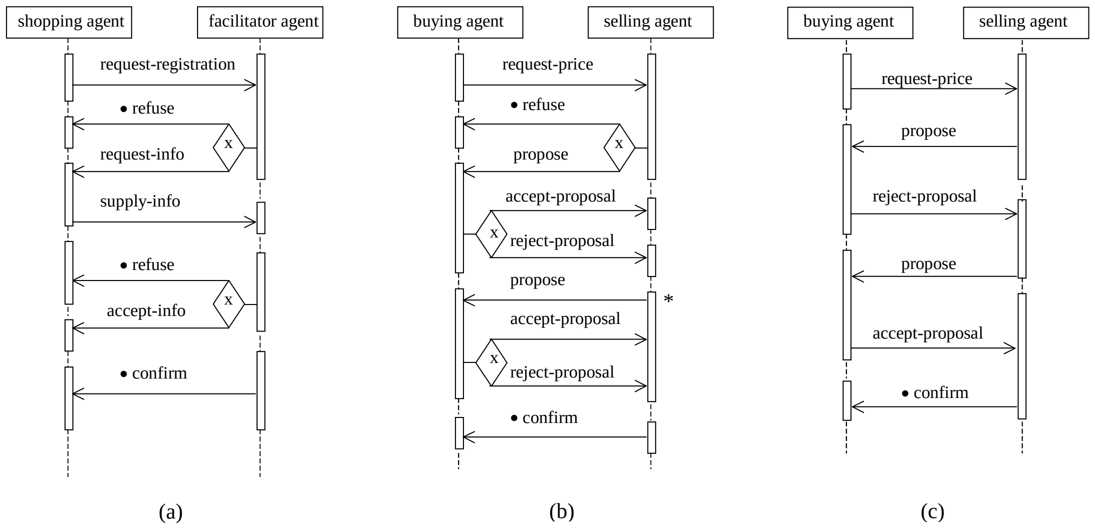


Figure 12. Contract net protocols (a) A template for the registration protocol (b) A template for the price-negotiation protocol (c) An example of the price-negotiation protocol

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. The description of the price-negotiation protocol between a buying agent and a selling agent in Figure 12 (b) and (c) can refer to Section 3.3.

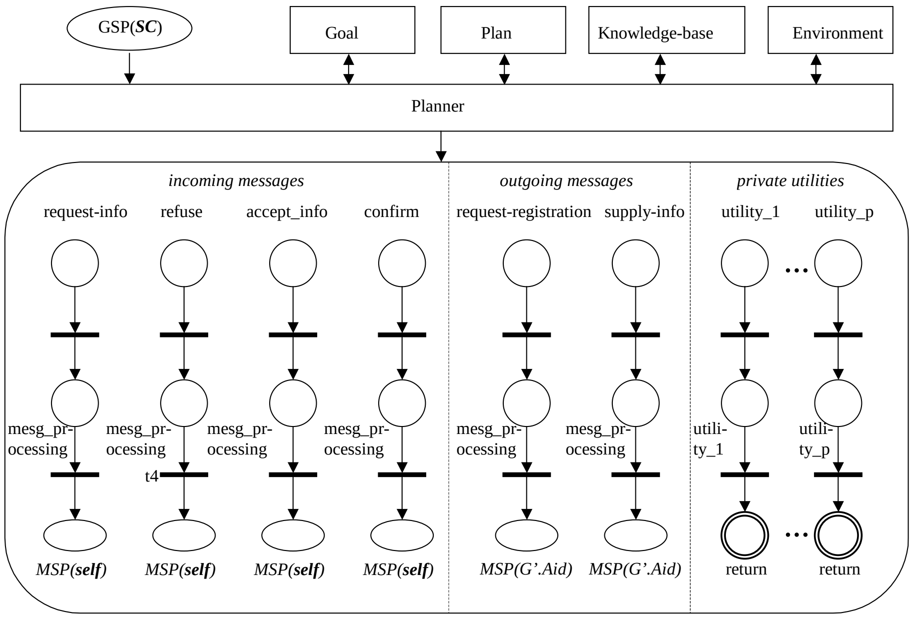


Figure 13. An agent-oriented G-net model for shopping agent class (SC)

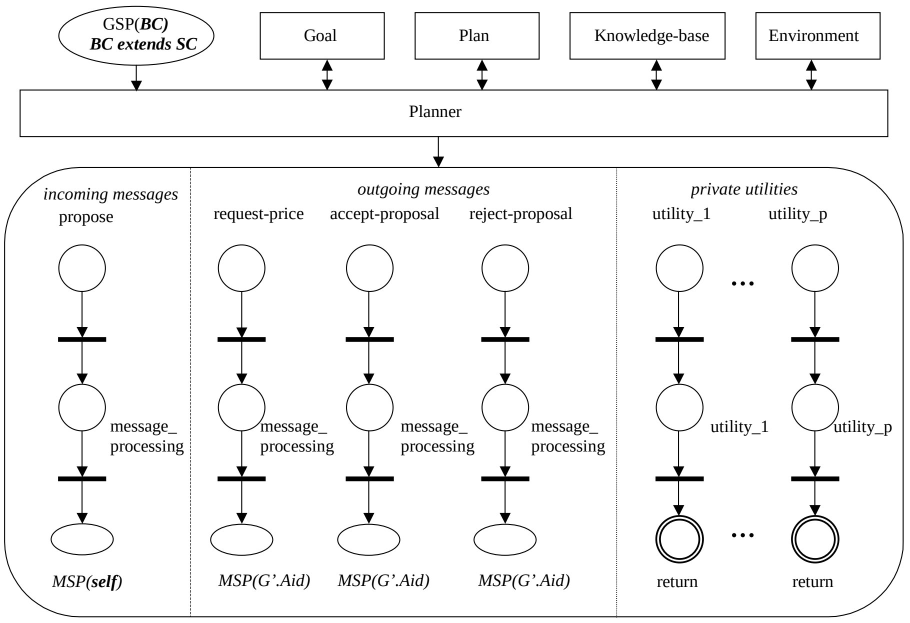


Figure 14. An agent-oriented G-net model for buying agent class (BC)

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 more 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.

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*, and *Knowledge-base* have been removed. Therefore, the transition *bypass* is enabled. When the transition *bypass* fires, the *mTkn* token will be directly deposited into the place *dispatch_outgoing_message*, and now the *mTkn* token can be correctly dispatched into the *MPU* for *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 utility methods 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 utility methods and different forms of inheritance can be found in Section 2.5 [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 algorithm 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.

Table IV

ALGORITHM FOR RESOLVING THE ***Super*** REFERENCE

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

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 in Table IV.

If a *true* value returns, an *MPU/Method* is discovered, and the *mTkn* can be directly deposited into the *GSP* of ***super***; otherwise, the *MPU/Method* cannot be found and an exception occurs. As stated before, we do not consider such exceptions in this chapter. 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 dominance over the original one. Obviously, our modified breadth-first-search algorithm correctly enforces this rule of dominance.

4.5 Summary

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 and buying 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 buying agents and selling agents, we illustrated how to create agent models and how to reuse functional units defined in an agent superclass.

In Chapter 5, we will show how an existing Petri net tool can be used to detect design errors, and how model checking techniques can support the verification of some key behavioral properties of our agent-oriented G-net model.

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 of special mechanisms, 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 Section 3.4, we have shown some analysis technique to prove some properties of our agent model by calculating minimal-support T-invariants. In this chapter, we use a Petri tool, called INA (Integrated Net Analyzer) [Roch and Starke 1999], to automatically 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 Place/Transition (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.

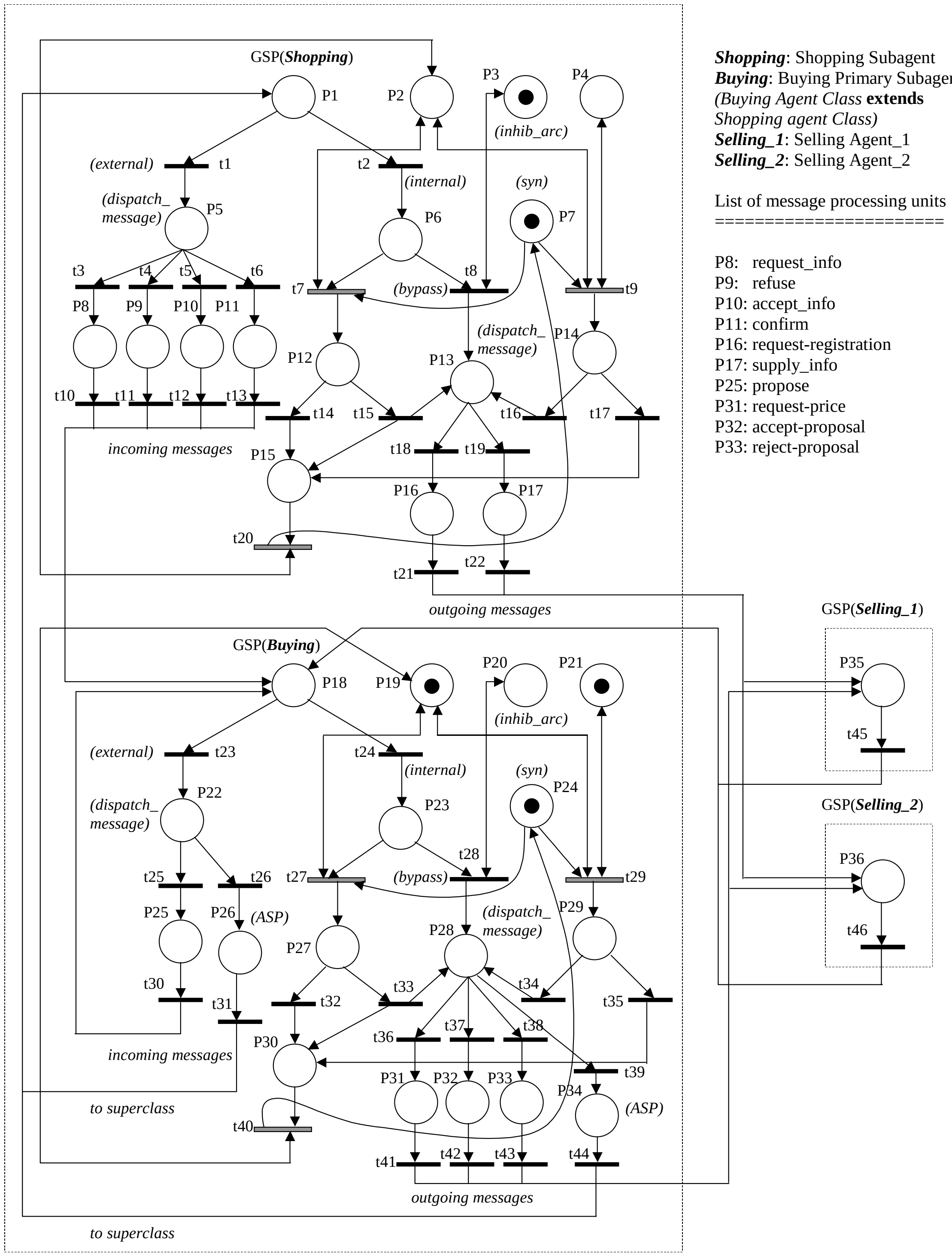
5.2 A Simplified Petri Net Model for a Buying Agent and Two Selling Agents

The interaction between a buying agent and two selling agents can be modeled as a net as in Figure 15. Table V and Table VI provide a legend that identifies the meaning associated with each place and transition in Figure 15. To derive this net model, we use a *GSP* place to represent each selling agent.

This is feasible 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* and *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 *utility method* 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].
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 chapter (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.

Note that the simplified net model preserves most of the behavioral properties of the agent-oriented G-net model, e.g., the property of concurrency. In Section 5.3, we use an existing Petri net tool – INA, to detect a deadlock error in this net model. The presence of the deadlock in Figure 15 is due to a design error in our initial design of the agent-oriented G-net model.



Shopping: Shopping Subagent
Buying: Buying Primary Subagent
(Buying Agent Class **extends** *Shopping agent Class)*
Selling_1: Selling Agent_1
Selling_2: Selling Agent_2

List of message processing units
======================

P8:	request_info
P9:	refuse
P10:	accept_info
P11:	confirm
P16:	request-registration
P17:	supply_info
P25:	propose
P31:	request-price
P32:	accept-proposal
P33:	reject-proposal

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

Table V

LEGEND FOR FIGURE 15 (DESCRIPTION OF PLACES)

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

Table VI

LEGEND FOR FIGURE 15 (DESCRIPTION OF TRANSITIONS)

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

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

The analysis shows that our net model is not live, and the dead reachable states indicate a deadlock. By tracing the firing sequence for those dead reachable states, we find that when there is a token in place *P29*, both the transitions *t34* and *t35* are enabled. At this time, if the transition *t35* fires, a token will be deposited into place *P30*. After firing transition *t40*, the token removed from place *P24*, by firing transition *t29*, will return to place *P24*, and this makes it possible to fire either transition *t27* or *t29* in a future state. However if the transition *t34* fires, instead of firing transition *t35*, there will be no tokens returned to place *P24*. So, transition *t27* and *t29* will be disabled forever, and a deadlock situation occurs.

To correct this error, we need to modify the design of the *Planner* module in Figure 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.

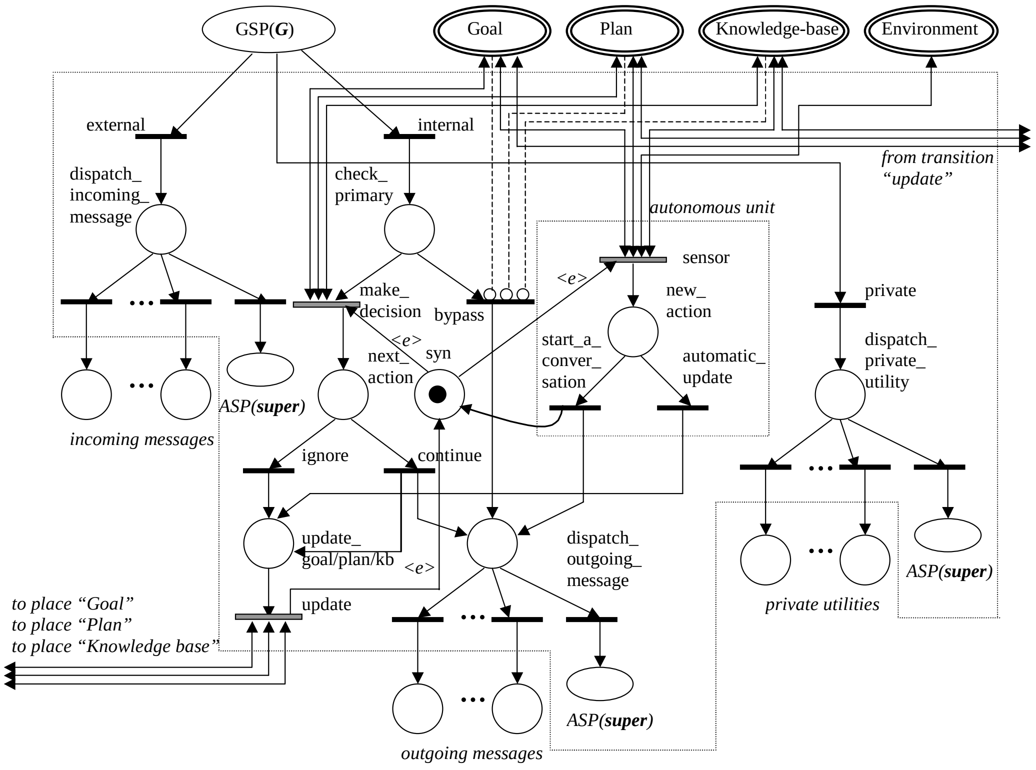


Figure 16. A template for the *Planner* module (revised design)

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 5.4, 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 (Computation Tree Logic) formulae [Roch and Starke 1999][Clarke *et al.* 1986]. The syntax of CTL formulae is defined as follows:

```
<FORMULA> ::= 'T' | 'P'<NUMBER> | '-'<FORMULA> |
            '('<FORMULA>'&'<FORMULA>')' | '('<FORMULA>'V'<FORMULA>')' |
            'E''X'<FORMULA> | 'E''F'<FORMULA> | 'E''G'<FORMULA> |
            'A''X'<FORMULA> | 'A''F'<FORMULA> | 'A''G'<FORMULA> |
            'E''['<FORMULA>'U'<FORMULA>']' | 'A''['<FORMULA>'U'<FORMULA>']'
            'E''['<FORMULA>'B'<FORMULA>']' | 'A''['<FORMULA>'B'<FORMULA>']'
<NUMBER> ::= <NONEMPTY SEQUENCE OF DIGITS FOLLOWED BY A BLANK>
```

`T` stands for the truth value “true”, predicates are referred to with `P<number>`, in the atomic mode `P<number>` denotes the criterion whether the place with the internal number `<number>` is marked or not, `-` stands for the logical “not”, `&` and `V` stand for the logical connectors “and” and “or”, respectively, and EX, EF, EG, AX, AF, AG, EU, EB, AU, and AB stand for the corresponding temporal-logical quantors as shown in Table VII.

Table VII

DESCRIPTION OF TEMPORAL-LOGICAL QUANTORS

Quantor	Description
EX *f*	There exists an immediate post-state in which *f* is satisfied.
A [*f1* **U** *f2*]	Along all paths, f_1 is satisfied until a state is reached which satisfies f_2.
E [*f1* **U** *f2*]	There exists a path, along which f_1 is satisfied until a state is reached which satisfies f_2.
A [*f1* **B** *f2*]	On all paths, f_1 is satisfied at least once, before a state is reached, which satisfies $f_{2.}$
E [*f1* **B** *f2*]	There exists a path along which f_1 is satisfied at least once, before a state is reached which satisfies f_2.
AX [*f*]	*f* is satisfied in all post-states.
AF [*f*]	*f* is satisfied on every path in the future.
EF [*f*]	There exists a path which leads to a state in which *f* is satisfied.
AG [*f*]	*f* is satisfied in every state on all paths.
EG [*f*]	There exists a path where *f* is satisfied in every state.

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 15, 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 15, 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 15, 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 15, 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 15, 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 chapter, 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 Summary

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*. Furthermore, model checking techniques were used to prove some key behavioral properties for our agent model, such as concurrency, mutual exclusion, and correctness of the inheritance mechanism. In addition to proving key behavioral properties of our agent model, our formal method approach is also of value in creating a clear understanding of the structure of an agent, which can increase confidence in the correctness of a particular multi-agent system design. Also, in producing a more detailed design, where the abstract transitions in the planner module are refined, we may again use Petri net tools to capture further design errors.

In Chapter 6, we discuss another research direction of software agents, i.e., mobile agents. We adapt the agent-oriented G-net model proposed in Chapter 4 to support some basic mobility concepts, and present a design model of intelligent mobile agents in the framework for agent-oriented software.

6. EXTENDING AGENT-ORIENTED G-NETS TO SUPPORT MOBILITY

6.1 Introduction

There are two main streams of research on software agents, namely the multi-agent systems (MAS) and mobile agents (MA). Research on multi-agent systems (MAS) is rooted in distributed artificial intelligence (DAI), and dates back to the fifties. In a multi-agent system, agents are autonomous, reactive, proactive and sociable. Agents in a MAS are usually distributed but static, and they are typical organized to execute a given task or achieve their own goals by collaborating and cooperating in an intelligent manner [Kinny and Georgeff 1997] [Jennings *et al.* 1998]. On the other hand, research on mobile agents usually emphasizes agent mobility and agent coordination, and mobile agents are usually assumed to only have very limited or even no intelligence [Roman *et al.* 1997][Mascolo 1999][Cabri *et al.* 2001]. The development scheme in the later case for mobile agents is sometimes called weak agent approach, which contrasts with the strong agent approach that involves artifical intelligence techniques [Silva *et al.* 2001].

For mobile agents, the concern is with software agents that can migrate over computer networks. The concept of location has been one of the key features to characterize mobility in most theoretical models of mobile agents, such as the distributed join-calculus [Fournet *et al.* 1996], which is an extension of the π-calculus that introduces the explicit notions of named localities and distribution failure. Additional typical formalisms for agent mobility modeling are summarized as follows. Mobile UNITY [Roman *et al.* 1997] provides a programming notation that captures the notion of mobility and transient interactions among mobile nodes. Inspired by Mobile UNITY, the concept of connectors [Wermelinger and Fiadeiro 1998] is explicitly identified to describe different kinds of transient interactions, and facilitate the separation of coordination from computation in mobile computing. The connectors are written in COMMUNITY, a UNITY-like program design language whose semantics is given in a categorical framework. MobiS [Mascolo 1999], as an extended version of PoliS, is a specification language based on multiple tuple spaces. It can be used to specify agent coordination and architectures containing mobile components. More recently, LIME [Murphy *et al.* 2001], also based on tuple spaces, has been proposed as a middleware that supports the development of applications that exhibit both physical and logical mobility. A formal architecture model for logical agent mobility has been proposed by using Pr/T nets [Xu *et al.* 2003].

Although the above efforts succeeded in formal modeling mobile agents in terms of their mobility, they are not built upon a framework that explicitly supports the intelligence feature of agents. In other words, they belong to the domain of weak agent approaches. Such models are typically reactive rather than pro-active, i.e., they simply act in response to their environment, but they are not able to exhibit goal-directed behaviors. Other efforts, such as the MARS project [Cabri *et al.* 2001], have attempted to introduce context-dependent coordination into agent models; however in these cases, the communication mechanisms between mobile agents are usually not explicitly suggested. There are also some research efforts concerned with mobile agent communication mechanisms, but without formal definitions [Baumann *et al.* 1997][Finin *et al.* 1998].

From the above review, we find that current work on mobile agents mostly emphasizes some particular features of the mobile agents, e.g., agent mobility. With the continuing improvement of agent technology, and the rapid growth of software system complexity, especially for Internet applications, there is a pressing need for a more general model of mobile agents, in which agents are not only mobile and cooperative, but also intelligent. There are a few previous efforts that discuss intelligent mobile agents [Ku *et al.* 1997][Finin *et al.* 1998], however they lack a formal framework for intelligent mobile agent design. In this chapter, we propose an intelligent mobile agent model by adapting basic mobility concepts into our previously discussed framework for agent-oriented software (Chapter 4). This framework, i.e., the agent-oriented G-net model, has been designed to model intelligent software agents for multi-agent systems, and it supports asynchronous message passing as well as design reuse of functional units. Therefore, our proposed formal model for intelligent mobile agents inherits these features. In addition, since our proposed model is based on the agent-oriented G-net formalism [Xu and Shatz 2003], it can be translated into more "standard" forms of a Petri net for design analysis, including model checking.

The rest of this chapter is organized as follows. Section 6.2 describes the mobile agent background. Section 6.3 proposes the architecture for a mobile agent system, and illustrates how to design the principle agent system components: the intelligent mobile agents (IMA) and the intelligent facilitator agents (IFA). Section 6.4 uses an electronic marketplace example to show how to incrementally design application-specific intelligent mobile agents using the discussed architecture. Finally, in Section 6.5, we summarize our work.

6.2 Mobile Agent Background

Traditionally, distributed applications have relied on the client-server paradigm in which client and server processes communicate either through message-passing or *Remote Procedure Call (RPC)*. This communication model is usually synchronous, i.e., the client suspends itself after sending a request to the server, waiting for the results to come back [Karnik and Tripathi 1998]. Obviously, a prerequisite for an *RPC* to work correctly is that the called procedure is available on the corresponding remote node. This requirement, however limits the usability of the *RPC* concept in large open distributed systems [Rothermel and Schwehm 1999]. In many cases, it is desirable to send a procedure to a remote node and execute it there. This level of flexibility is introduced by the concept of *Remote Evaluation (REV)* [Stamos and Gifford 1990a][Stamos and Gifford 1990b] in which the client, instead of invoking a remote procedure, sends its own procedure code with parameters to a server, and requests the server to execute it and return the results. Common Gateway Interface (CGI) and Java Servlets are typical examples of this technique. The concept of *REV* can be generalized in the sense that not only the client may send the code to a server but also vice versa. With the scheme of *Code on Demand (COD)*, a client initiates the transfer of the program code, and programs stored on server machines are downloaded to the client on demand. The currently most popular technologies supporting this type of mobility are ActiveX Controls and Java Applets.

The mobile agent paradigm has evolved from these antecedents. Figure 17 illustrates how it differs from *RPC* and *REV*/*COD*. In *RPC*, only parameters (data) are transmitted from the client to the server, and the results (data) are returned after the execution of the procedures stored on the server machines. In *REV*, both procedures (code) and parameters (data) are sent from the client to the server, and the results (data) are returned after the procedures coming from the client are executed on the server machine. In *COD*, only procedures (code) are downloaded from the server, and they are executed on the client machine. Obviously, both *REV* and *COD* support "code mobility" rather than "agent mobility", as both schemes transfer their procedures before their activation. In contrast, mobile agents are defined as objects that have behavior, state and location. Mobile agents are autonomous because once they are invoked they will autonomously decide which locations they will visit and what instructions they will perform. A mobile agent is a program (encapsulating code, data and state) sent by a client to a server. Unlike a procedure call, it does not have to return its results to the client. It could migrate to other servers, transmit information back to its original, or migrate back to the client if needed. It thus has more autonomy than a simple procedure call.

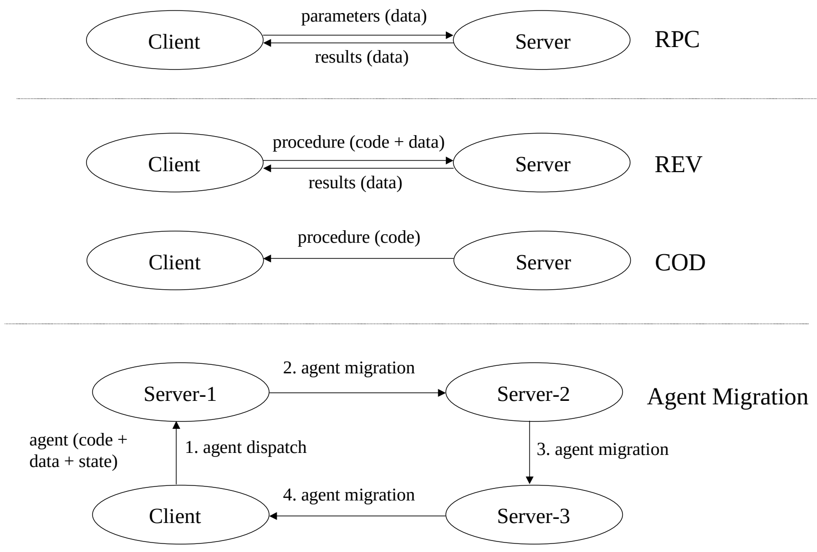


Figure 17. Evolution of the mobile agent paradigm

Previous work on building mobile agent systems can be summarized as follows. Telescript [White 1995], developed by General Magic, was the first commercial product to provide a technology for distributed application development based on mobile agents. AgentTcl is a mobile agent system developed at the Dartmouth College [Gray 1995], which is an extension of the Tool Command Language (Tcl) to write mobile agents. TACOMA (Tromso And Cornell Moving Agents) system runs on several UNIX platforms, and applications of mobile agents may be written in Tcl/Tk, C, Scheme, Perl or Python [Johansen *et al.* 1995]. Mole is one of the first Java-implementations of mobile agent systems and was developed at the University of Stuttgart [Straßer *et al.* 1997]. It uses Java as the agent programming language as well as the implementation language. Mole is thus a pure Java application and can be started at every computer platform for which a Java Development Kit (JDK) is available. Java aglet has been developed at the IBM Tokyo Research Laboratory, which is a lightweight mobile agent for the Java programming environment [Lange *et al.* 1997]. An aglet is a mobile Java object that can be launched by a visual agent manager called Tahiti. For more prominent mobile agent systems, refer to paper [Rothermel and Schwehm 1999][Silva *et al.* 2001]. Note that the above examples of mobile agent systems are neither built upon formal agent models, nor illustrate any agent intelligence. These missing features make the above efforts differ from our research, in which we aim to build intelligent mobile agent systems based on a formal agent model.

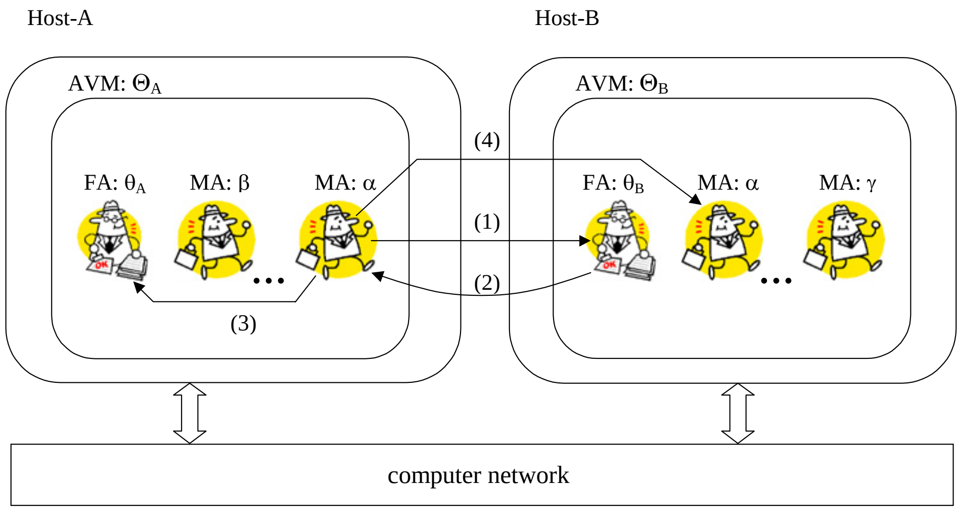
6.3 Modeling the Agent World for Mobile Agents

Today's users demand ubiquitous network access independent of their physical location. This style of computation, often referred to as *mobile computing*, is enabled by rapid advances in wireless communication technology [Murphy *et al.* 2001]. The networking scenarios enabled by mobile computing range roughly between two extremes. At one end, the availability of a fixed network is assumed, and its facilities are exploited by the mobile infrastructure. We call this form of mobility *logical mobility*. At the other end, the fixed network is absent and all network facilities (e.g., routing) must be implemented by relying only on the available mobile hosts, namely *ad hoc* networks. This form of mobility is called *physical mobility*. Mobile agent technology is a new networking technology that deals with both forms of mobility. It offers a new computing paradigm in which a program, in the form of an intelligent software agent, can suspend its execution on a host computer, transfer itself to another agent-enabled host on the network, and resume execution on the new host. Here, as we will see in the next section, we define a host as either a static host or a mobile host, which is situated in an *ad hoc* network.

6.3.1 Agent World Architecture

First, we introduce the concepts of agent virtual machine (AVM) and agent world (AW), which serve to define a framework for a mobile agent system. Figure 18 shows a generic mobile agent system, and an example of agent migration. In the figure, Host-A and Host-B are two machines connected by a network. To make mobile agent platform independent, a mobile agent runs on an agent virtual machine (AVM), which provides a protected agent execution environment. Each host may have a number of AVMs, however, to make it simple, we only illustrate one AVM for each host in Figure 18. Each AVM is responsible for hosting and executing any agents created on that AVM or those arrive over the network, and for providing API for agent programmers.

We now provide a few key definitions for the mobile agent system. Note that our definitions for mobile agents apply for both logical and physical mobility.



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

Figure 18. Agent world architecture and an example of agent migration

Definition 6.1 *Agent World (AW)*

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

Definition 6.2 *Static Host (SH) and Mobile Host (MH)*

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

Definition 6.3 *Agent Virtual Machine (AVM)*

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

Definition 6.4 *Static Agent (SA) and Mobile Agent (MA)*

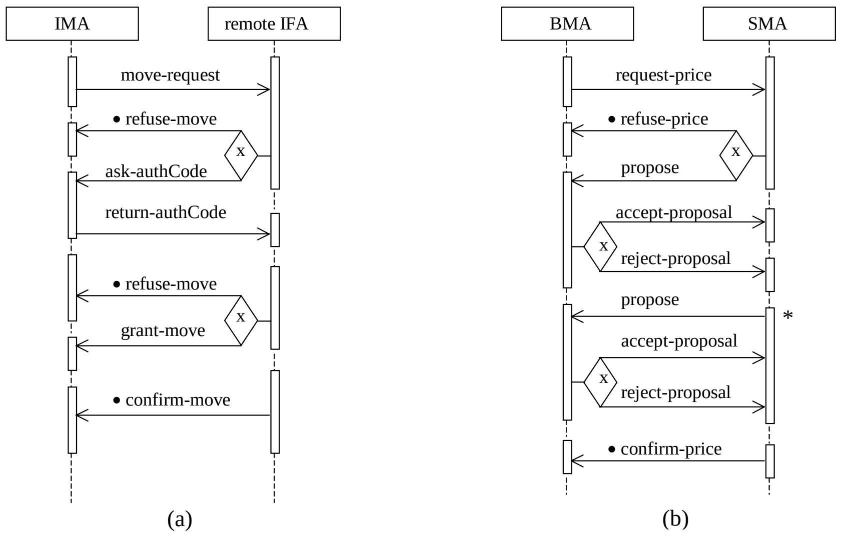
An *agent A* is 3-tuple (HOMEIP, CURIP, AO), where HOMEIP is the IP address of the host, on which agent A is created. CURIP is the IP address of the host where agent A currently runs on. AO is the agent object with the general structure as we described in Section 4.2. If at ant time, CURIP = HOMEIP, we refer to agent *A* as a *static agent (SA)*; otherwise, we refer to agent *A* as a *mobile agent (MA)*.

Since in this chapter we view mobile agents and facilitator agents (an example of static agent) as intelligent software agents, for the rest of this chapter a mobile/facilitator agent always refers to an intelligent mobile agent (IMA) or an intelligent facilitator agent (IFA). As shown in Figure 18, when a mobile agent α on AVM Θ_A wants to migrate to another AVM Θ_B, it needs to contact with the remote facilitator agent θ_B first, which resides on AVM Θ_B (step 1). In fact, the mobile agent α needs to know the address of the remote facilitator agent θ_B before the communication can begin. This could be done by querying this information from its local facilitator agent θ_A, which resides on AVM Θ_A. If the local facilitator agent θ_A knows the address of the remote facilitator agent θ_B, it will provide this information to the mobile agent α; otherwise, it will contact with the well-known static host Π (we do not show it in Figure 18) for this information and forward the results to the mobile agent α thereafter. For simplicity, this procedure is omitted in Figure 18. Based on security and resource criteria, the remote facilitator agent θ_B decides if the migration request is granted. If the migration request is granted (step 2), the mobile agent α notifies its local facilitator agent θ_A about its leaving (step 3), and it finally moves to the remote AVM Θ_B (step 4). In the following section, we will see that, in our approach, step 1 and step 2 are modeled by asynchronous message passing; while step 3 and step 4 are modeled by method invocations.

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

6.3.2 <u>Intelligent Mobile Agent and Intelligent Facilitator Agent</u>

To illustrate the processes for design of intelligent mobile agents (IMA) and intelligent facilitator agents (IFA) by extending our proposed agent-oriented G-net model (Chapter 4), we use the following examples. Since we view a facilitator agent as an IFA, in addition to provide public services to a mobile agent or some other IFA, an IFA also has the capability of making decisions. This feature is vitally important for an IFA to cater for the needs of service allocation in a dynamic network environment, such as resource management and security verifications. Figure 19 (a) depicts a template of a contract net protocol [Flores and Kremer 2001] expressed as an agent UML (AUML) sequence diagram [Odell 2001] for a migration-request protocol between a *mobile agent (MA)* and a remote *facilitator agent (FA)*. Figure 19 (b) depicts a template of a contract net protocol expressed as an AUML sequence diagram for a price-negotiation protocol between a *buying mobile agent (BMA)* and a *selling mobile agent (SMA)*.



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

Figure 19. Contract net protocols (a) a temple for the migration-request protocol (b) a template for the price-negotiation protocol

Consider Figure 19 (a). When a conversation based on a contract net protocol begins, the *intelligent mobile agent (IMA)* sends a request for migration to a remote *intelligent facilitator agent (IFA)* on a different AVM. The remote *IFA* can then choose to respond to the *IMA* by refusing its migration or asking the *IMA*'s authorization code, which is used to verify that the *IMA* is on a trustable AVM. Here the "x" in the decision diamond indicates an exclusive-or decision. If the remote *IFA* refuses the migration based on resource limitation or some other reasons, the protocol ends; otherwise, the remote *IFA* waits for the *IMA*’s authorization code to be supplied. If the *IMA*’s authorization code is correctly provided, the remote *IFA* may grant the *IMA* for migration if it is trustable, or refuse the migration otherwise. Again, if the remote *IFA* refuses *IMA*'s migration, the protocol ends; otherwise, a confirmation message will be provided afterwards. Similarly, the price-negotiation protocol between a *buying mobile agent (BMA)* and a *selling mobile agent (SMA)*, which are subclasses of *IMA*, can be illustrated in Figure 19 (b).

Based on the communicative acts (e.g., *move-request*, *refuse-move*, etc.) needed for the contract net protocol in Figure 19 (a), we may adopt the agent design template shown in Figure 5, and design the intelligent mobile agent class (IMA) as in Figure 20. The *Goal*, *Plan*, *Knowledge-base* and *Environment* modules remain as abstract units and can be refined in further design stages. The *Planner* module may reuse the template shown in Figure 16. The design of the remote facilitator agent, i.e., intelligent facilitator agent class (IFA), is similar; however, since an *IFA* works as a server, and it may provide public services to other agents, we must extend the *IS* in Figure 5 with a *public service* section. The resulting design for *IFA* is illustrated in Figure 21.

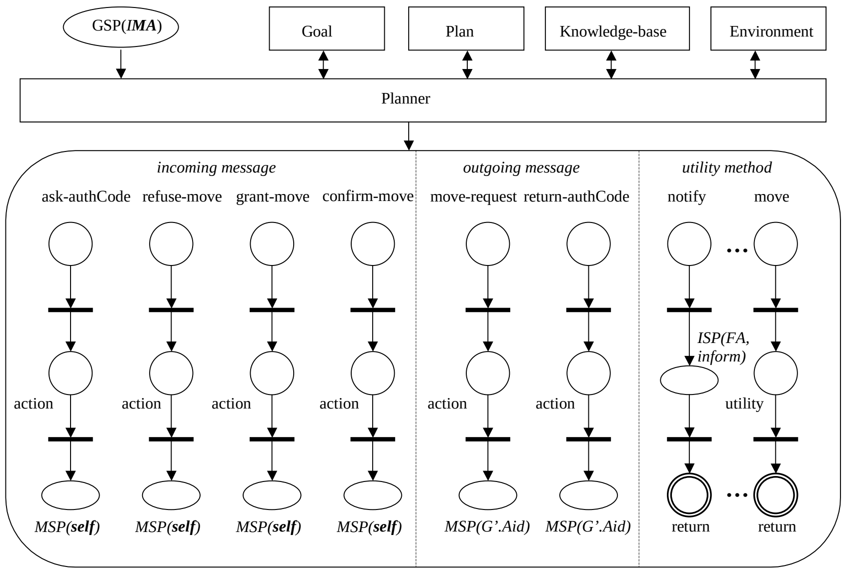


Figure 20. An agent-oriented G-net model for intelligent mobile agent class (IMA)

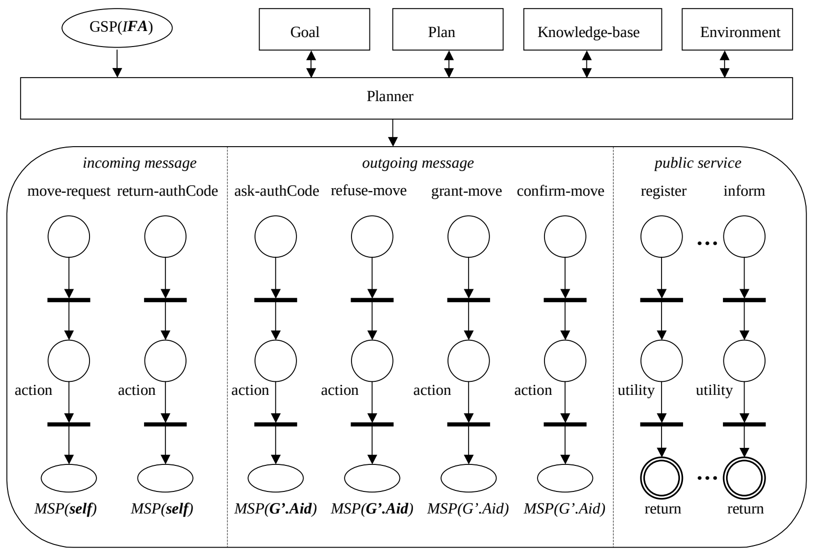


Figure 21. An agent-oriented G-net model for intelligent facilitator agent class (IFA)

To show how our agent models work correctly in an agent conversation, we now discuss an example. Consider a mobile agent object *MAO*, which receives a message of *ask-authCode* from a remote facilitator agent object *FAO*. A *mTkn* token with a tag of **external** will be deposited in the *GSP* of the primary subagent of *MAO*, i.e., the *GSP* of the corresponding intelligent mobile agent class (*IMA*). The transition *external* in *MA*'s *Planner* module may fire, and the *mTkn* will be moved to the place *dispatch_incoming_message*. Since there is an *MPU* for *ask-authCode* defined in the internal structure of *MA*, the *mTkn* will be dispatched to the entry place of that *MPU*. After the message is processed, *MSP(**self**)* changes the tag of the *mTkn* from **external** to **internal**, and sends the processed *mTkn* token back into the *GSP* of *IMA*. Upon the arrival of this message token, the transition *internal* in the *Planner* module of *MA* may fire, and the *mTkn* token will be moved to the place *check_primary*. Since *IMA* corresponds to the primary subagent of *MAO*, there are tokens in the special places *Goal*, *Plan*, *Knowledge-base* and *Environment*. Therefore the abstract transition *make_decision* may fire, and any necessary actions are executed in place *next_action*. Then the current conversation is either ignored or continued based on the decision made in the abstract transition *make_decision*. If the current conversation is ignored, the goals, plans and knowledge-base are updated as needed; otherwise, in addition to the updating of goals, plans and knowledge-base, a newly constructed *mTkn* with a tag of **internal** is deposited into place *dispatch_outgoing_message*. The new *mTkn* token has the message name *return-authCode*, following the protocol defined in Figure 19 (a). Again, there is an *MPU* for *return-authCode* defined in *IMA*, so the new *mTkn* token will be dispatched into the entry place of that *MPU*. After the message is processed, the *MSP(G'.Aid)* mechanism changes the tag of the *mTkn* token from **internal** to **external**, and transfers the *mTkn* token to the *GSP* of the receiver agent, in this case, the remote facilitator agent object *FAO*.

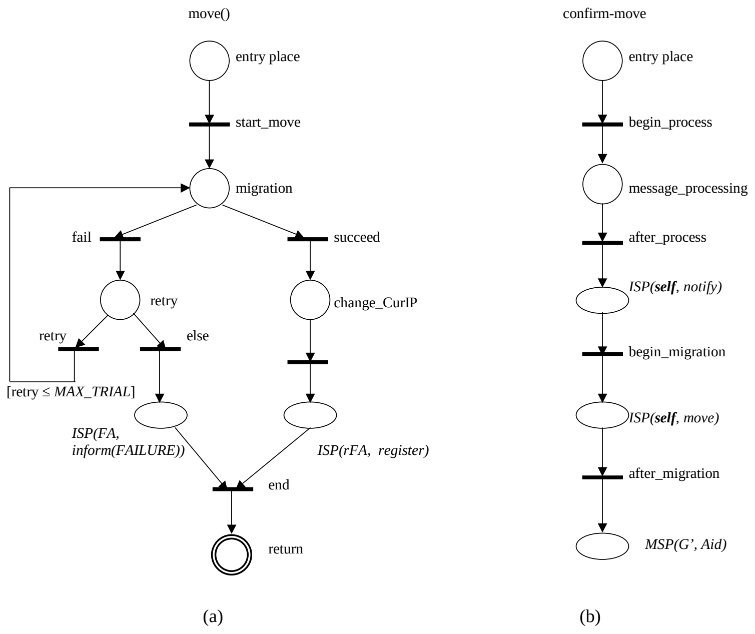


Figure 22. Refinement of functional units (a) Refinement of method *move()*
(b) Refinement of MPU *confirm-move*

To further illustrate how to refine the MPU/method in a mobile agent's internal structure, we use the examples of the MPU *confirm-move* defined in the incoming message section and the method *move* defined in the utility method section. The refinement of another method *notify()* is straightforward; as shown in Figure 20, the *notify()* method makes a method invocation *inform()* to its local facilitator agent. This is done to notify the facilitator agent that the calling agent is leaving. The refinement of method *move()* and MPU *confirm-move* are shown in Figure 22 (a) and Figure 22 (b), respectively. In Figure 22 (a), when there is a token deposited in the entry place, the transition *start_move* fires, and deposit a token into place *migration*. The migration might be successful or failed, due to the network condition. If the migration fails, the transition *fail* fires, and deposits a token into place *retry*. The mobile agent will then count the number of retrials. If it has retried less than *MAX_TRIAL* times, the mobile agent will try to migrate again; otherwise, the transition *else* fires, and a method call *inform(FAILURE)* will be made to its local *facilitator agent (FA)* to notify the local *FA* that its migration is failed. This is modeled by the *ISP(rFA, inform(FAILURE))* mechanism. After that, the method call *move()* returns. If the migration succeeds, the transition *succeed* fires, and the mobile agent's current IP address *CURIP* will be changed to the new one. Then a method call *ISP(rFA, register)* is made to the remote *facilitator agent (FA)*, which is actually the mobile agent's local *FA* now. After registering with the *FA*, the method call *move()* returns.

In Figure 22 (b), the refinement of MPU *confirm-move* is straightforward. When there is a token deposited into the entry place of the MPU *confirm-move*, the transition *begin_process* fires. After processing the message token, it makes a method call *ISP(**self**, notify)* to the agent itself, which further makes a method call to the mobile agent's local facilitator agent -- to inform the facilitator agent that the mobile agent is leaving. After that, the migration starts by invoking the method *move()*. Finally, after finishing the migration, either failed or succeeded, it transfers the message token to the agent itself, and ends the conversation.

6.4 Design of Intelligent Mobile Agents in an Electronic Marketplace

Consider a mobile agent family in an electronic marketplace domain, which is a global stock market tracking and trading system. Figure 23 shows the agents in a UML class hierarchy notation. An *intelligent mobile agent class (IMA)* is defined as a superclass that is capable of communicating with an *intelligent facilitator agent class (IFA)*, and migrating among *AVM*s. The functionality of an *intelligent mobile agent class (IMA)* can be inherited by an agent subclass, such as a *buying mobile agent class (BMA)* or a *selling mobile agent class (SMA)*. Both the *BMA* and *SMA* may reuse the functionality of *IMA* for communication with *IFA* and migration among *AVM*s. Furthermore, a *broker mobile agent class* is designed as a subclass of both the *BMA* and *SMA*, and a *stock-buyer/stock-seller mobile agent class* may be defined as a subclass of a *BMA*/*SMA*.

Based on the communicative acts (e.g., *request-price*, *refuse-price*, etc.) needed for the contract net protocol between the *buying mobile agent (BMA)* and the *selling mobile agent (SMA)*, we may design the *BMA* as shown in Figure 24. The *SMA* can be designed in the same way.

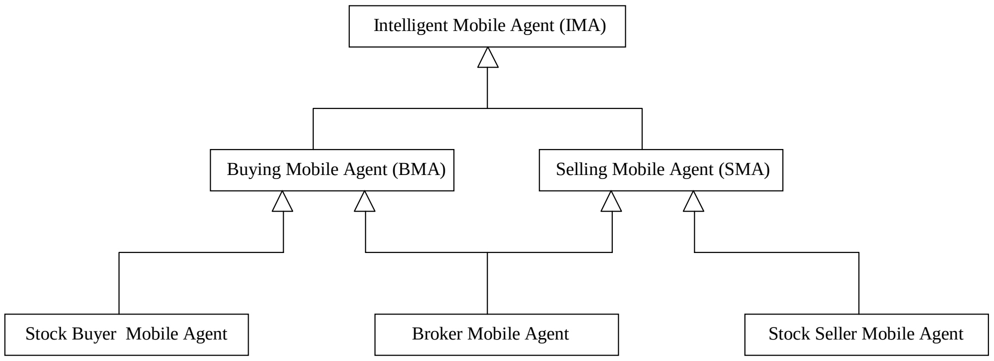

Intelligent Mobile Agent (IMA)
Buying Mobile Agent (BMA)
Selling Mobile Agent (SMA)
Stock Buyer Mobile Agent
Broker Mobile Agent
Stock Seller Mobile Agent

Figure 23. The class hierarchy diagram of mobile agents in an electronic marketplace

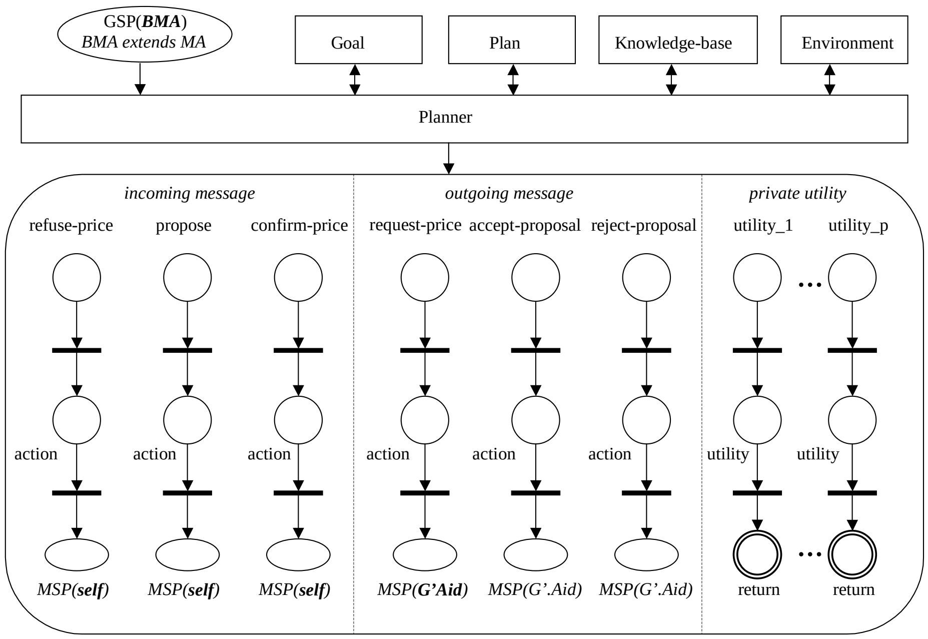


Figure 24. An agent-oriented G-net model for buying mobile agent class (BMA)

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

Now we discuss an example to show how the reuse of *MPU/methods* works. Consider a buying mobile agent object *BMO*, which receives a message of *ask-authCode* from a remote facilitator agent object *FAO*. A *mTkn* token will be deposited in the *GSP* of the primary subagent of *BMO*, i.e., the *GSP* of the corresponding buying mobile agent class (*BMA*). The transition *external* in *BMA*'s *Planner* module may fire, and the *mTkn* will be moved to the place *dispatch_incoming_message*. Since there is no *MPU* for *ask-authCode* defined in the internal structure of *BMA*, the *mTkn* will be moved to the *ASP(**super**)* place. Since ***super*** here refers to a unique superclass – the mobile agent class (*MA*) – the *mTkn* will be transferred to the *GSP* of *MA*. Now the *mTkn* can be correctly dispatched to the *MPU* for *ask-authCode*. After the message is processed, *MSP(**self**)* changes the tag of the *mTkn* from **external** to **internal**, and sends the processed *mTkn* token back into the *GSP* of *BMA*. Note that *MSP(**self**)* always sends a *mTkn* back to the *GSP* of the primary subagent. Upon the arrival of this message token, the transition *internal* in the *Planner* module of *BMA* may fire, and the *mTkn* token will be moved to the place *check_primary*. Since *BMA* corresponds to the primary subagent of *BMO*, there are tokens in the special places *Goal*, *Plan*, *Knowledge-base* and *Environment*. Therefore the abstract transition *make_decision* may fire, and any necessary actions are executed in place *next_action*. Then the current conversation is either ignored or continued based on the decision made in the abstract transition *make_decision*. If the current conversation is ignored, the goals, plans and knowledge-base are updated as needed; otherwise, in addition to the updating of goals, plans and knowledge-base, a newly constructed *mTkn* with a tag of **internal** is deposited into place *dispatch_outgoing_message*. The new *mTkn* token has the message name *return-authCode*, following the protocol defined in Figure 19 (a). Again, there is no *MPU* for *return-authCode* defined in *BMA*, so the new *mTkn* token will be dispatched into the *GSP* of *MA*. Upon the arrival of the *mTkn* in the *GSP* of *MA*, the transition *internal* in the *Planner* module of *MA* may fire. However at this time, *MA* does not correspond to the primary subagent of *BMO*, so all the tokens in the special places of *Goal*, *Plan*, and *Knowledge-base* have been removed. Therefore, the transition *bypass* is enabled. When the transition *bypass* fires, the *mTkn* token will be directly deposited into the place *dispatch_outgoing_message*, and now the *mTkn* token can be correctly dispatched into the *MPU* for *return-authCode* defined in *MA*. After the message is processed, the *MSP(G'.Aid)* mechanism changes the tag of the *mTkn* token from **internal** to **external**, and transfers the *mTkn* token to the *GSP* of the receiver agent, in this case, the remote facilitator agent *FAO*.

For the reuse of public services and utility methods defined in a superclass, the situation is the same as in the case of object-oriented design. In addition, there are three different forms of inheritance that are commonly used, namely augment inheritance, restrictive inheritance and refinement inheritance. The usage of these three forms of inheritance in agent-oriented design is also similar to that in object-oriented design. Examples concerning reuse of public services and utility methods and different forms of inheritance can be found in earlier work [Xu and Shatz 2000].

6.5 Summary

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

We presented the design models of intelligent mobile agents in a framework for agent-oriented software. Unlike previous work, which only models a particular feature of mobile agents, our mobile agent models can be served as a general agent model that has the capabilities of mobility, corporative behavior, and intelligence. With the example of electronic marketplace, we show that application-specific mobile agents can be design incrementally as subclasses of the mobile agent base class, i.e, the IMA class.

Furthermore, our intelligent mobile agent models are based on the agent-oriented G-net formalism, which can be translated into a standard form of Petri net (Predicate/Transition net, Pr/T net) [Murata 1989][Deng *et al.* 1993]. Because the Petri net formalism is theoretically mature and supported by robust tools, our approach supports formal analysis, such as model checking. In addition, since we embed agent movement, and any other possible actions, in the context of agent conversations, we believe that our approach leaves adequate room for security modeling.

In Chapter 7, we will return to the topic of multi-agent systems (MAS). We will discuss about an agent development kit (ADK) that is based on the agent-oriented G-net model (Chapter 4), which provides a framework and a full class library for development of multi-agent software systems.

7. AN AGENT DEVELOPMENT KIT BASED ON AGNET-ORIENTED G-NET MODEL

7.1 Introduction

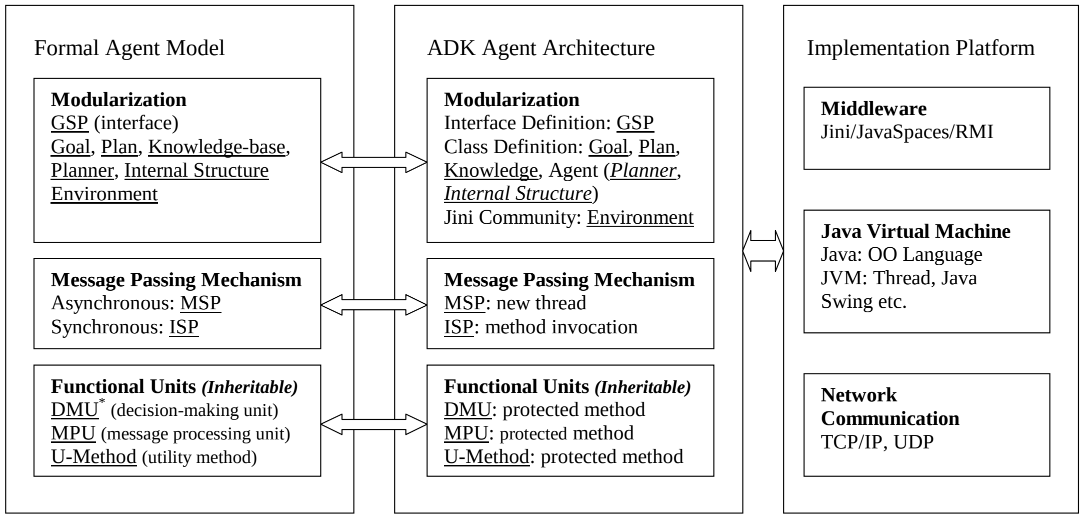
The development of agent-based systems offers a new and exciting paradigm for production of sophisticated programs in dynamic and open environments, particularly in distributed domains such as web-based systems and electronic commerce. An intelligent agent is defined as an agent that at least has the following characteristics: autonomy, reactivity, proactiveness, and sociability. Agent autonomy is akin to human free-will and enables an agent to choose its own actions, while agent proactiveness requires an agent to behave in a goal-directed fashion. Agent proactiveness is usually considered in relation to planning, and is strengthened with agent autonomy. We call an autonomous and proactive agent a *goal-driven* agent. A reactive agent is defined as an agent that has the ability to perceive and to response to a changing environment. We call a reactive agent an *event-driven* agent, and an event could be any environment change that may influence an agent's execution. The sociability of an agent refers to the ability of an agent to converse with other agents. The conversations, normally conducted by sending and receiving messages, provide opportunities for agents to coordinate their activities and cooperate with each other, if needed. An agent is different from an object in that agents usually do not use method invocations to communicate with each other. In contrast, agents distinguish different types of messages and use complex protocols, such as contract net protocols [Smith 1980][Flores and Kremer 2001], to negotiate. In addition, agents analyze these messages and can decide whether to execute the requested action [Wooldridge *et al.* 2000]. To meet this requirement, the design of agents needs to support asynchronous message passing. We call an agent that supports asynchronous message passing a *message-triggered* agent.

Though there have been significant commercial and industrial research and development efforts underway for some time, developments based on formal agent frameworks are rare. In this chapter, we present a development approach, including design and implementation, for intelligent agents in multi-agent systems (MAS). The approach is based on our proposed formal agent model, i.e., the agent-oriented G-net model, discussed in Chapter 4. To bridge the gap between formal agent modeling and agent implementation, we integrate our formal model into the design phase of the agent development life cycle. Unlike most current research on formal modeling of agent systems or agent behavior [Wooldridge and Ciancarini 2001], our agent model specifically serves as a high-level design for agent implementation, rather than just as a specification for agent behaviors. In other words, the formal model guides a software engineer by prescribing "how," rather than "what," to develop in terms of intelligent agents. Our formal agent model supports design modularization and inheritance. To show the feasibility of our approach, we highlight a system that provides a full class-library for the domain of intelligent agents for multi-agent systems. We call the development system ADK (Agent Development Kit).

The rest of this chapter is organized as follows. In Section 7.2, we discuss the role of ADK in serving as a bridge between the formal agent model and the agent implementation platform. In Section 7.3, we describe the architectural design and detailed design of intelligent agents, and discuss the role of inheritance in agent development. We also summarize the procedures to design and implement intelligent agents for multi-agent systems. In Section 7.4, we use an air ticket trading example to illustrate the derivation of an application using the ADK approach. The generality of the example supports the notion that our model-based approach is feasible and effective. In Section 7.5, we summarize our work.

7.2 From Formal Agent Model to Agent Implementation

We now turn to the fundamental contributions in this chapter – the principles and practice of a proposed Agent Development Kit (ADK). ADK is intended to provide the necessary facilities for agent implementation based on the formal agent model described previously. Thus, the development of ADK is not ad hoc, but results from a model-based development process. The agent-oriented G-net model, as an operational model, provides the high-level design for intelligent agents. Specifically, the key components or mechanisms defined in the agent-oriented model serve as building blocks of our agent development kit.


Formal Agent Model
Modularization
GSP (interface)
Goal, Plan, Knowledge-base, Planner, Internal Structure Environment
Message Passing Mechanism
Asynchronous: MSP
Synchronous: ISP
Functional Units (Inheritable)
DMU* (decision-making unit)
MPU (message processing unit)
U-Method (utility method)
ADK Agent Architecture
Modularization
Interface Definition: GSP
Class Definition: Goal, Plan, Knowledge, Agent (Planner, Internal Structure)
Jini Community: Environment
Message Passing Mechanism
MSP: new thread
ISP: method invocation
Functional Units (Inheritable)
DMU: protected method
MPU: protected method
U-Method: protected method
Implementation Platform
Middleware
Jini/JavaSpaces/RMI
Java Virtual Machine
Java: OO Language
JVM: Thread, Java Swing etc.
Network Communication
TCP/IP, UDP

** DMUs are not inheritable in agent-oriented G-net model*

Figure 25. The role of ADK between formal agent model and implementation platform

As Figure 25 shows, the role of ADK agent architecture is to serve as a bridge between the formal agent model and the agent implementation platform. Between the formal agent model and the ADK agent design architecture, there is a clear mapping of agent components and mechanisms. For instance, the *GSP* place defined in the formal agent model can be mapped to the *GSP interface definition* in the ADK agent architecture. Thus we claim that the formal agent model can be interpreted as an agent design model, and the model reveals to the software engineers not only the agent properties and behaviors, but more importantly, the agent architecture.

The key components and mechanisms defined in the formal agent model and their mappings to the components and implementation strategies in the ADK agent architecture are listed as follows: First, the modularization of the agent design provides the formal agent architecture that makes an agent autonomous, reactive, proactive and sociable. The *GSP* place in the formal model is defined as the only interface for agent communication, and this is carried forward in the ADK agent design architecture. The *Goal*, *Plan*, and *Knowledge-base* modules are based on the BDI agent model [Kinny *et al.* 1996] that is a conceptual model for intelligent agents. These modules are mapped to the class definitions of *Goal*, *Plan* and

Knowledge in the ADK agent architecture. The *Planner* module is used for decision-making, message dispatching and event capturing. And the *Internal Structure* is a container for methods and *MPU*s, where methods are defined for method invocation, and *MPU*s support asynchronous message passing. These two modules are defined as two sections in the definition of the *Agent* class. Finally, the *Environment* module in the formal agent model will be implemented as the Jini community in ADK.

Second, the message passing mechanisms are defined in two cases: synchronous message passing and asynchronous message passing. Synchronous message passing is usually used for method invocation, and it is realized through the *ISP* mechanism; while asynchronous message passing is vital for agent communication, and it is achieved by the *MSP* mechanism [Xu and Shatz 2001a]. Recall that in the case of asynchronous message passing, when a *MSP* is called, the agent does not need to wait for the result to come back, and it may proceed to execute other functionality. Straightforwardly, the *ISP* mechanism maps to method invocation in ADK, and *MSP* maps to a new thread on platforms such as JVM.

Third, the formal agent model defines the functional units as inheritable components. As methods are defined as inherited units in object-oriented design, all *U-Methods* (Utility Methods) and *MPU*s (Message Processing Units) could be inherited from an agent superclass to an agent subclass. In ADK, the functional units, including *DMUs* (Decision-Making Units), which are defined in a superclass, are implemented as protected methods that can be reused by their subclasses.

As shown on the right hand side of Figure 25, the implementation platform provides the standard computer technologies, such as the Jini middleware [Edwards 1999][Arnold *et al.* 1999] and the Java Virtual Machine (JVM), for agent implementation. We choose Java as our programming language because applications developed on JVM are platform independent, and they are suitable for web-based applications such as electronic commerce. In addition, we use the Jini middleware to simplify our development process for agent communication. In this case, we do not need to take care of the low-level communication protocols, such as the TCP/IP and UDP protocols, which can be automatically handled by the Jini middleware, and therefore, we can concentrate on high-level communication protocols, such as price-negotiation protocol. In summary, ADK represents the design and implementation of intelligent agents for multi-agent systems, and it refines the formal agent model and derives the detailed design as will be discussed in Section 7.3.

7.3 Design of Intelligent Agents

7.3.1 Middleware Support for Agent Communication

As we mentioned before, the Jini middleware can be used to simplify the development process for agent communication. The Jini architecture is intended to resolve the problem of network administration by providing an interface where different components of the network can join or leave the network at any time [Edwards 1999][Arnold *et al.* 1999]. Such a collection of services is called a Jini community, and the services within the Jini community represent service providers or service consumers. The heart of the Jini system is a trio of protocols called *discovery*, *join*, and *lookup*. *Discovery* occurs when a service is looking for a lookup service with which to register. *Join* occurs when a service has located a lookup service and wishes to join it. And *lookup* occurs when a client or user needs to locate and invoke a service described by its interface type and possibly, other attributes.

Table VIII

SCHEMA FOR AN AGENT INTERFACE

```
public interface GSP extends Remote {
    public void asynMessagePassing(Message message) throws RemoteException;
}

public class MiddlewareSupport implements GSP {
    // agent interface
    public void asynMessagePassing(Message message) {
        System.err.println("This method should be overridden by an agent "
                           + "class!");
    }

    // find lookup services and join the Jini community
    public void setup(String[] groupsToJoin) {…}
    …
}
```

In designing the ADK, we use Jini as a middleware for agents to find each other and to communicate with each other. Each agent is designed as both a service provider and a service consumer. Since agents only interact with each other through asynchronous message passing, the service provided by an agent through Jini is designed as an interface to let other agents send asynchronous messages to that agent, and the agent who sends out the messages becomes the service consumer. This approach is consistent with the agent-oriented G-net model, in which the *GSP (Generic Switch Place)* is defined as the only interface among agents [Xu and Shatz 2003]. Thus, we design the schema for an agent interface as in Table VIII.

The class *MiddlewareSupport* implements the *GSP* interface, where an abstract method *asynMessagePassing()* is defined. However, in class *MiddlewareSupport*, the implementation of this method is again deferred to subclasses of the *MiddlewareSupport* class because we want that the class *MiddlewareSupport* only defines the functionality to deal with the Jini community, such as discovering lookup service on the network, registering with the Jini community, and searching for other agents in the Jini community. Here the method *setup()* is defined to let the *GSP* find a lookup service and joins the Jini community. As we will see in Section 7.3.2, the *Agent* class, which is defined as a subclass of the *MiddlewareSupport* class, actually implements the method *asynMessagePassing()*, and inherits all the functionality defined in class *MiddlewareSupport*.

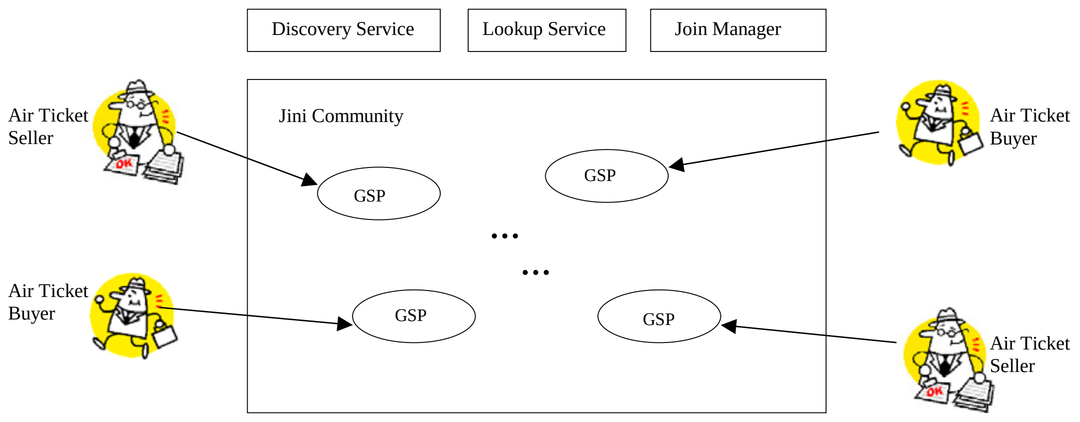


Figure 26. The Jini community with agents of *AirTicketSeller* and *AirTicketBuyer*

As an example, consider the design of an electronic marketplace in which seller agents and buyer agents may find each other and communicate with each other asynchronously through the Jini community. The design is illustrated in Figure 26, where both air ticket seller agents and air ticket buyer agents register their *GSP* interfaces with the Jini community, and they may find each other by the agent attribute, for instance, an agent name called “Seller”.

7.3.2 A Pattern for Intelligent Agents

Figure 27 shows the architectural design for intelligent agents. By comparing this figure to the agent-oriented G-net model in Figure 5 (Chapter 3), one can observe how the agent model drives the agent design. One obvious variation is that the *GSP* place of an agent model becomes a part of the environment module, which is the Jini community, in the agent design architecture. In the design architecture, each agent is composed of its *GSP* component, which serves as the agent’s interface element, and its action component (consisting of the following major elements: *Goal*, *Plan*, *Knowledge-base*, *Planner*, *Internal Structure*). Note that Figure 27 explicitly shows only the action component for one agent, agent *B*. The environment module contains the interface element for agent *B*, as well as some other interface elements (e.g., for agent *A*). The directed arcs shown in Figure 27 represent only a sample of the logical connections between the various elements – for example we explicitly see the connection from agent *B*’s interface to its planner module, and from agent *B*’s outgoing message processing unit to agent *A*’s interface (under the assumption that agent *B* does send messages to agent *A*).

Currently, we use a simplified version of the environment module in ADK, in which case the only external events of concern are those related to agents entering and/or leaving the Jini community. In future design versions, we can extend the environment module to include other events, such as network topology changes and user interventions. Similarly, data changes in *Goal*, *Plan* and *Knowledge-base* modules may act as internal events and trigger the sensor in the *Planner* module. To simplify matters, in ADK the sensor in the *Planner* module is implemented to only capture external events.

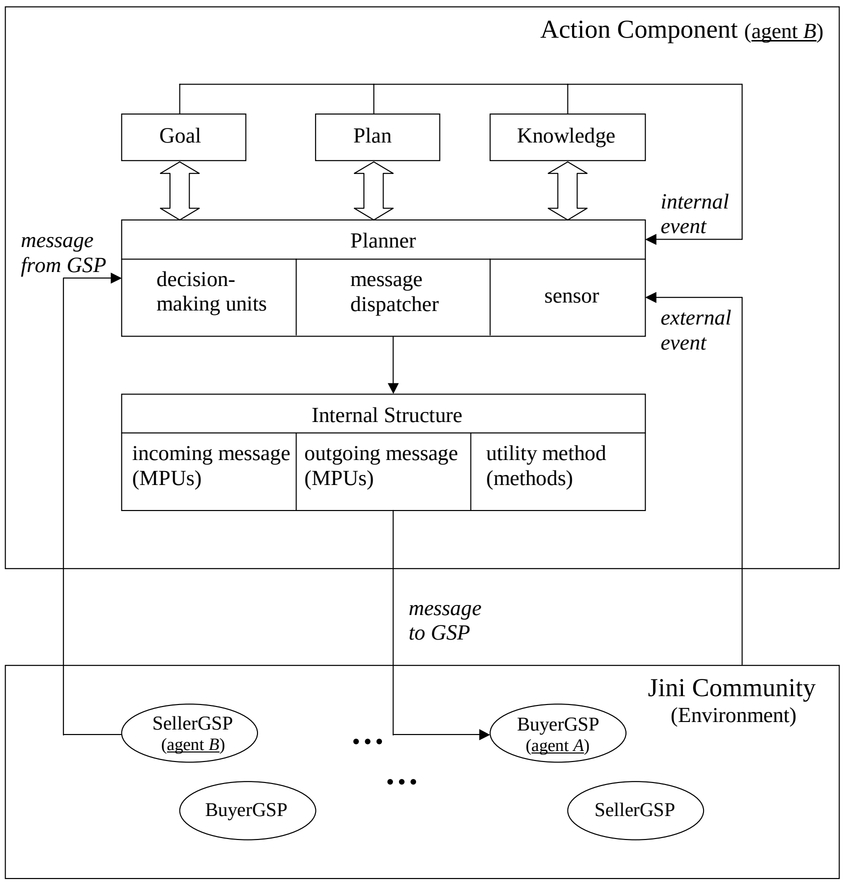


Figure 27. The architectural design of intelligent agents

Referring again to Figure 27, we can observe that when agent *A* wants to converse with agent *B*, it sends a message to the *GSP* of agent *B* in the Jini community (but, this connection is not explicitly shown in Figure 27 since agent *A*'s action component is not shown). Then the message will be sent to the *Planner* module of agent *B*. After the message is dispatched into a *MPU* in the incoming message section, the message will be processed, e.g., decoded, and sent back to the *Planner* module. Now the message goes to the decision-making units, where decisions may be made to ignore the message, or to continue with the conversation. If the conversation is to be continued, a new outgoing message is generated, and dispatched into a *MPU* defined in the outgoing message section. The outgoing message will be processed and certain actions may be executed before the message is sent to the *GSP* of agent *A*.

In addition, the *MPU*s and the *U-Methods* (defined in the incoming/outgoing message section and utility method section, respectively) can be inherited by agent subclasses, and can only be accessed or called by the agent itself. Unlike the agent-oriented G-net model, methods defined in the planner module can also be inherited optionally if a subclass agent chooses to reuse or refine the reasoning mechanisms defined in its superclass. This treatment is practical if we need to derive a subclass agent with similar behavior to its superclass – for instance, to derive a domestic air ticket seller agent class from a general air ticket seller agent class.

The goal of the above architectural design is to derive an architectural rendering of a system, which serves as a framework from which more detailed design activities are conducted. Based on the architectural design illustrated in Figure 27, we now proceed to describe the detailed design of intelligent agents for multi-agent systems. This design is expressed in the form of a pattern or class template.

Since the agent-oriented G-net model supports inheritance, we will follow this design schema and present first the pattern for the *Agent* class, which is a superclass for application-specific agents. The design schema for application-specific agents will be introduced in Sections 7.3.3 and 7.3.4. In an object-oriented system, design patterns can be used with either inheritance or composition. Using inheritance, an existing design pattern becomes a template for a new subclass, and the attributes and operations that exist in the pattern become part of the subclass [Pressman 2001]. Similarly, in an agent-oriented system, a pattern of an agent superclass can serve as a template for an agent subclass, and a specific agent subclass, such as an air ticket seller agent class, can be derived from an agent superclass by augmenting the template to meet system requirements.

The *Agent* class defined in ADK provides such a pattern for agent implementation. The pattern is shown in Table IX in a form of Java pseudocode. As shown in Table IX, the *Agent* class is defined as a subclass of *MiddlewareSupport* (defined in Section 7.3.1) to reuse the functionality of discovering a lookup service, registering with the Jini community, and searching for other agents. More importantly, an agent object may communicate with other agent objects asynchronously through the *GSP* interface. This functionality makes an agent sociable. To simulate the asynchronous message passing, we have used the thread technique to generate a new thread called *messageProcessThread*. Upon receiving an incoming message, the *messageProcessThread* of the message receiver (the callee) dispatches the message to a *MPU* and returns immediately. This ends up the *messageProcessThread* quickly, and therefore, the message sender (the caller) does not need to wait for the message to be processed and may proceed to execute other tasks.

Table IX

A PATTERN FOR INTELLIGENT AGENTS

```
public class Agent extends MiddlewareSupport {
    private static final String PRODUCT = "Agent";
    private static final String VERSION = "ADK 1.0";
    …

    /**************************
     * Agent Interface -- GSP *
     **************************/
    public void asynMessagePassing(Message message) {
        Thread messageProcessThread = new Thread(new Runnable() {
            public void run() {
                dispatchMessage(message);         // -- message-triggered
            }
        });
        messageProcessThread.start();
     }

     /************************************************
      * Class Variables for Knowledge, Goal and Plan *
     /************************************************/
     Goal myGoals; // a list of committed goals
     Plan myPlans; // a set of plans
     Knowledge myKnowledge; // a knowledge-base
     …

     /***********
      * Planner *
      ***********/
     private class Sensor extends Listener {
         …
         public void notify(RemoteEvent ev) {
             if (!(ev instanceof ServiceEvent)) return;
             updateServices();
             invokePlan(ev);                      // -- event-driven
         }
     }
     protected void dispatchMessage(Message message) {…}
     protected Message makeDecision(Message message) {…}
     protected void updateMentalState() {…)
     …

     /**********************
      * Internal Structure *
      **********************/
     // incoming message section – a set of message processing units
     protected void MPU_In_Hello(Message message) {…}
     …
     // outgoing message section – a set of message processing units
     protected void MPU_Out_Hello(Message outgoingMessage) {…}
     …
     // utility method section – a set of private utility methods
     initAgent(String[] args) {…}
     protected void autonomousRun() {…}
     protected void other_Method_1() {…}
     …

     public static void main(String[] args) {
         initAgent(args);
         autonomousRun();                         // -- goal-driven
     }
}
```

Corresponding to the three modules (*Goal*, *Plan* and *Knowledge*) in the architectural design of intelligent agents (Figure 27), the *Agent* class defines a list of committed goals *myGoals*, a set of plans *myPlans*, each of which is associated with a goal or a subgoal, and a knowledge-base *myKnowledge*. The *Goal*, *Plan* and *Knowledge* class define the basic properties and behaviors for an intelligent agent, and may be refined or redefined if an application-specific agent requires further functionality. Refer to Figure 30 for the definitions of the *Goal*, *Plan* and *Knowledge* class. For brevity, other class variables, such as *theGoalSet* – a set of goals from which the goal list *myGoals* is generated – are omitted in Table IX.

The reactivity of an agent can be designed through the Jini's notification facility. In the Jini community, whenever a new event occurs, an agent should be automatically notified by the system. For instance, when a seller agent joins or leaves the Jini community, the buyer agents need to be notified; thus, the buyer agents can always keep an up-to-date list of the seller agents that are currently in the community (by keeping a list of interested agents locally, it can also decrease the network traffic). In Table IX, we can see that the *Sensor* class is defined as a private inner class in the *Agent* class, and is derived as a subclass from the *Listener* class, which is defined by the Jini. Thus, an application class, such as a seller agent class or a buyer agent class, which will be defined as a subclass of the *Agent* class, can be notified by the Jini community whenever an event occurs, as long as the corresponding agent object has instantiated a *Sensor* object and has registered it with the Jini community.

Based on the architectural design of intelligent agents in Figure 27, the *Planner* module in the *Agent* pattern defines a method called *dispatchMessage()*, which is used to dispatch messages to the appropriate *MPU* defined in the incoming/outgoing message section. Examples of methods defined as decision-making units in the *Planner* module are the methods *makeDecision()* and *updateMentalState()*. In method *makeDecision()*, decisions are made to ignore an incoming message, to start a new conversation, or to continue with the current conversation. In method *updateMentalState()*, the mental state of the agent, i.e., the goal, plan, and knowledge-base are updated whenever a decision is made or a new event occurs. The *Internal Structure* module includes three sections, i.e., the incoming message section, outgoing message section, and utility method section. Each section defines a set of *MPU*s or methods, which are depicted as *MPU_In_x()*, *MPU_Out_y()* or *Method_k()* in Table IX. We only implemented the *MPU_In_Hello()* and *MPU_Out_Hello()* in the *Agent* class, which allows instances of the *Agent* class or any of its subclasses to greet with each other. Further protocol related *MPUs* shall be defined in agent subclasses. The autonomy and proactiveness of an agent are related with the *Goal*, *Plan*, *Knowledge-base*, *Planner* and *Internal Structure* modules of an agent. To connect them together, we define the control as the method *autonomousRun()*, which includes a list of committed goals to be achieved based on the agent's mental state. Each goal is defined as a goal tree that is traversed in depth-first order, and selected plans associated with each goal or subgoal are invoked accordingly. The method *autonomousRun()* is invoked in the method *main()*, as shown in Table IX, and is executed after the agent is initialized with the method *initAgent()*.

7.3.3 Inheritance in Agent-Oriented Development

Inheritance in agent-oriented programming has been studied in terms of reusing mental states such as goal, plan and knowledge [Crnogorac *et al.* 1997]. We argue that since an agent maintains a dynamic list of goals and plans, and acquires most of its knowledge during its lifetime, to inherit mental states is not appropriate. Furthermore, agents are autonomous with different goal-directed behavior. For instance, in the class hierarchy of Figure 28, an air ticket seller agent and a book seller agent shall have different goals and plans, and more practically, they may have different negotiation strategies and reasoning mechanisms. Thus, the class hierarchy in Figure 28 shall only imply the reuse of superclass' functional mechanisms, for instance, the communication mechanism. Since inheritance happens at the class level, new knowledge acquired, new plans made, and new goals generated in an agent object (e.g., an air ticket seller), cannot be inherited by a subclass agent object (e.g., a domestic air ticket seller). In contrast, most of the functional mechanisms, for instance the function of comparing prices or selecting the ticket with shortest travel time, can be reused. Optionally, the domestic air ticket seller may also reuse the negotiation strategies adopted by an air ticket seller, but practically it may have its own specific strategies. As a result, we need to allow a subclass agent to inherit any reasoning mechanisms defined in its superclass agent, but also allow such a subclass agent to redefine or refine these mechanisms.

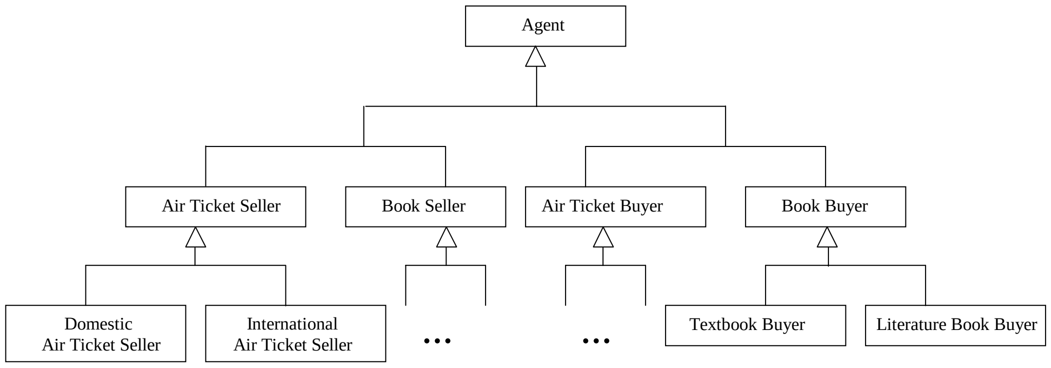


Figure 28. The class hierarchy diagram of agents in an electronic marketplace

Figure 29 shows the inheritance relationship between the classes defined in ADK and classes derived from the *Agent* class. In this figure, all the classes above the dashed line are provided as an agent framework or a class library – these classes define the ADK environment, which supports developing intelligent agents for multi-agent systems. The classes below the dashed line are derived classes that represent specific intelligent agents in a multi-agent system. This figure shows that both the air ticket seller agent and the air ticket buyer agent may reuse the functional mechanisms and reasoning mechanisms defined in the superclass *Agent*. Especially, air ticket seller agents and air ticket buyer agents may communicate with each other through Jini. We do not need to deal with this issue again in the design of these two classes, since all needed functionality for communication through Jini has been implemented in the *Agent* class and can be reused by its subclasses. The event-driven feature is also inherited by the air ticket seller agents and the air ticket buyer agents. In other words, a designer of subclasses of the *Agent* class does not need to be concerned this feature, since subclasses automatically have this feature inherited from their superclass, i.e., the *Agent* class. In addition, the air ticket seller agent and air ticket buyer agent may reuse the default reasoning mechanisms defined in the *Agent* class. The default reasoning mechanism is defined as a search through a goal tree that achieves each subgoal, with associated plans, in a depth-first search order.

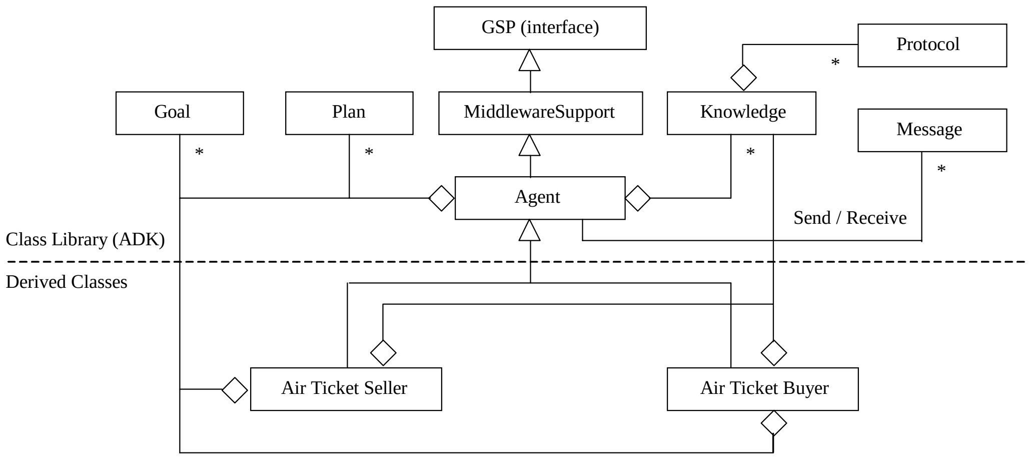


Figure 29. Classes defined in ADK and derived classes of the *Agent* class

Though most of the features defined in the *Agent* class can be reused, each subclass of the *Agent* class shall associate with the *Goal*, *Plan* and *Knowledge* class directly. This implies that any goal, plan or knowledge defined in a superclass cannot be inherited by its subclasses. This design is consistent with the high-level design of agent-oriented G-net models, in which the *Goal*, *Plan* and *Knowledge-base* modules of the superclass are disabled when the inheritance mechanism is invoked. The *Goal*, *Plan,* and *Knowledge* classes define the basic (default) structure for their corresponding modules. Subclasses of the *Agent* class may either reuse these structures or define their own. Obviously, if these classes are redefined, the reasoning mechanisms shall also be redefined in the subclass agents.

This approach derives the template (pattern) for application-specific agent design, which is defined as a subclass of the *Agent* class in ADK. The template is shown in Table X. In this template, we use the definitions of the *Goal*, *Plan*, and *Knowledge* classes that are defined in ADK, but it is worth noting that designers can define their own classes for these modules. Alternatively, they may refine these classes (defined in ADK) by subclassing them and inheriting their default structures.

Table X

DESIGN OF APPLICATION-SPECFIC AGENTS

```
public class ApplicationSpecificAgent extends Agent {

    /************************************************
     * Class Variables for Knowledge, Goal and Plan *
    /************************************************/
    Goal myGoals; // committed goals, redefinition of Goal class is optional
    Plan myPlans; // plans, redefinition of Plan class is optional
    Knowledge myKnowledge; // knowledge-base, redefinition of Knowledge
                           // class is optional
    …

    /***********
     * Planner *
     ***********/
    protected void dispatchMessage(Message message) {…}  // refinement
                                                         // or redefinition
    protected Message makeDecision(Message message) {…}  // refinement
                                                         // or redefinition
    protected void updateMentalState() {…)               // refinement
                                                         // or redefinition
    …

    /**********************
     * Internal Structure *
     **********************/
    // incoming message section – a set of message processing units
    protected void MPU_In_1(Message message) {…}         // new definition
    …

    // outgoing message section – a set of message processing units
    protected void MPU_Out_1(Message outgoingMessage) {…}// new definition
    …

    // utility method section – a set of private utility methods
    protected void initAgent(String[] args) {…}          // refinement
                                                         // or redefinition
    protected void autonomousRun() {…}                   // refinement
                                                         // or redefinition
    protected void other_Inherited_Method_1() {…}        // refinement
                                                         // or redefinition
    …
    protected void other_New_Method_1() {…}              // new definition
    …

    public static void main(String[] args) {
        initAgent(args);
        autonomousRun();
    }
}
```

In the *Planner* section of the *ApplicationSpecificAgent* class, all the decision-making units (e.g., *makeDecision* and *updateMentalState*) inherited from those defined in the *Agent* class can be refined or redefined. In an extreme case, this section can be left blank, if the default reasoning mechanisms defined in the *Agent* class are reused. In the *Internal Structure* section, sets of *MPU*s are defined corresponding to a set of protocols. For instance, from a price-negotiation protocol, we can derive a set of *MPU*s, such as *request-price*, *propose*, *accept-proposal*. A description of how to derive *MPU*s from interaction protocols has been developed in Chapter 3 [Xu and Shatz 2003], but this level of detail is outside the scope of this dissertation.

In the *utility method* section, methods (i.e., *U-Methods*) are defined as “protected” so that they can be further inherited by their subclasses. In addition to refining or redefining the two outstanding methods, *initAgents()* and *autonomousRun()*, we can refine or redefine any inherited methods defined in the utility method section of the *Agent* class. Furthermore, new application-specific functions shall be added here.

One advantage of our model-based approach is its support for the principle of “separation of concerns,” in particular the separation of agent mental states and agent communication capabilities. Therefore, it is possible for us to choose some existing implementation schema of intelligent agents (agent with or without communication capabilities) to design and implement intelligent agents for multi-agent systems. For instance, we can choose the *Task Representation Language (TRL)* to support knowledge representation and agent reasoning [Ioerger *et al.* 2000], or we can use Petri nets to model the mental state of agents for multi-agent simulation [Yen *et al.* 2001]. Alternatively, we can, and do, use a more commonly used intelligent agent model – the *Belief-Desire-Intention (BDI)* model [Kinny *et al.* 1996]. A *BDI* architecture includes and uses an explicit representation for an agent's beliefs, desires and intentions. The BDI implementations, such as The Procedural Reasoning System (PRS), the University of Michigan PRS, and JAM, all define a new programming language and implement an interpreter for it [Vidal *et al.* 2001]. The advantage of this approach is that the interpreter can stop the program at any time, save state, and execute some other plan, or intention, if it needs to. In this chapter, we use a simplified implementation of the BDI agent model based on previous work, and show relationships between agent mental states and communication related modules.

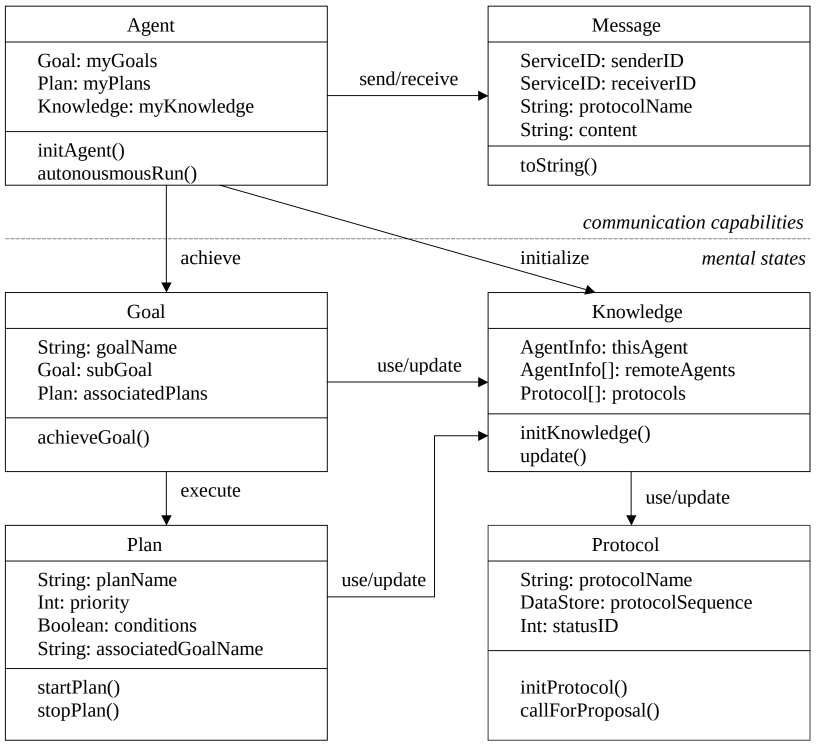


Figure 30. Relationship between classes defined for communication capabilities and mental states

The relationships between the key classes defined for communication capabilities and agent mental states are illustrated in Figure 30. As shown in this figure, two key classes for communication capabilities are the *Agent* class and the *Message* class, and an *Agent* object may send or receive *Message* objects through its *GSP* interface. Meanwhile, the three key classes for an intelligent BDI agent are the *Goal*, *Plan* and *Knowledge* class. A *Goal* object is defined as a goal tree, and a goal or a subgoal associates with a set of plans. When a goal or a subgoal is to be achieved, the most appropriate plan, for instance the plan with the highest priority, is selected and executed. As a result of the execution of a plan, a *Knowledge* object may be updated. Both a *Goal* object or a *Plan* object may use the *Knowledge* object for its own purpose, e.g., to select the appropriate plan to achieve a goal or a subgoal. Protocol instances are defined inside the *Knowledge* class. Therefore, the *Knowledge* class may use/update protocols.

The *Agent* class defines a list of committed goals *myGoals*, a set of plans *myPlans* that associate with a goal or a subgoal, and a knowledge-base *myKnowledge*. The list of committed goals and the set of plans may be updated at run time. For instance, when a goal is achieved, it may be deleted from the goal list, and new goals may be added into the goal list if needed. In addition, the *myKnowledge* object is initialized by the *Agent* object, and may be updated at run time by a *Goal* or *Plan* object. The intelligent agent is so-called *goal-driven*, because in the method *automousRun()*, goals defined in the goal list are achieved one by one through a loop. When all the goals are achieved, the *Agent* object waits for new committed goals to be added into the goal list.

7.3.4 An Agent Development Process

The purpose of the proposed agent design architecture and agent design patterns is to ease the programmer's effort to develop applications of intelligent agents for multi-agent systems. As we mentioned before, a specific agent, such as an air ticket seller agent, could be defined as an agent subclass of the *Agent* class. Since the *Agent* class shown in Table IX provides the basic functionality of intelligent agents as well as the agent implementation framework, what we need to do for developing a application-specific intelligent agent is to inherit the functional units and the behaviors of the *Agent* superclass and fill out certain sections in the pattern for application-specific agent, as shown in Table X. In addition, we need to redefine or define subclasses of the *Goal*, *Plan*, and *Knowledge* classes that are defined in ADK to meet certain behavioral requirements of agent intelligence.

As a summary, we now briefly describe the generic procedure to develop a specific intelligent agent for multi-agent systems. In Section 7.4, we cast the procedure into more specific terms by way of an example. The 6-step procedure is defined as follows:

1. Define a set of goals Φ as the class variable *theGoalSet*, where each goal is defined as a goal tree Γ. A goal tree could consist of just a root, which means a goal may or may not have a number of subgoals.

2. Define a goal list Ω as the class variable *myGoals* (Table X) and initialize the goal list Ω with any committed goal $g_c \in \Phi$. The goal list Ω is dynamic, which means achieved goals may be deleted from Ω and newly committed goals could be added into Ω at run time.
3. Define a set of plans P as the class variable *myPlans* (Table X). Each plan $p \in P$ has a priority and a set of conditions, and is associated with a particular goal or subgoal. The plan $p_{hp} \in P$, which has the highest priority and whose conditions are evaluated to *true*, will be executed to achieve the associated goal or a subgoal.
4. Refine the *Knowledge* class, including the *Protocol* class, if the application-specific agent requires additional types of knowledge beyond the basic properties and behaviors predefined in Figure 30, and initialize the knowledge-base *myKnowledge* (Table X) for that agent.
5. An interaction protocol ρ serves as a template for agent conversation. Based on ρ, we define a set of *MPU*s Ψ, where each MPU corresponds to a method *MPU_In_i()* or *MPU_Out_j()* as shown in Table X. Refer to [Xu and Shatz 2001a][Xu and Shatz 2001b] for a detailed description for transforming from ρ to ψ.
6. Refine the decision-making units defined in the *ApplicationSpecificAgent* class, if needed. Examples of decision-making units include functions like *makeDecision()* and *updateMentalState()*.

The decision-making units serve as the reasoning engine for the agent. The major functionality of the decision-making units includes the following tasks:

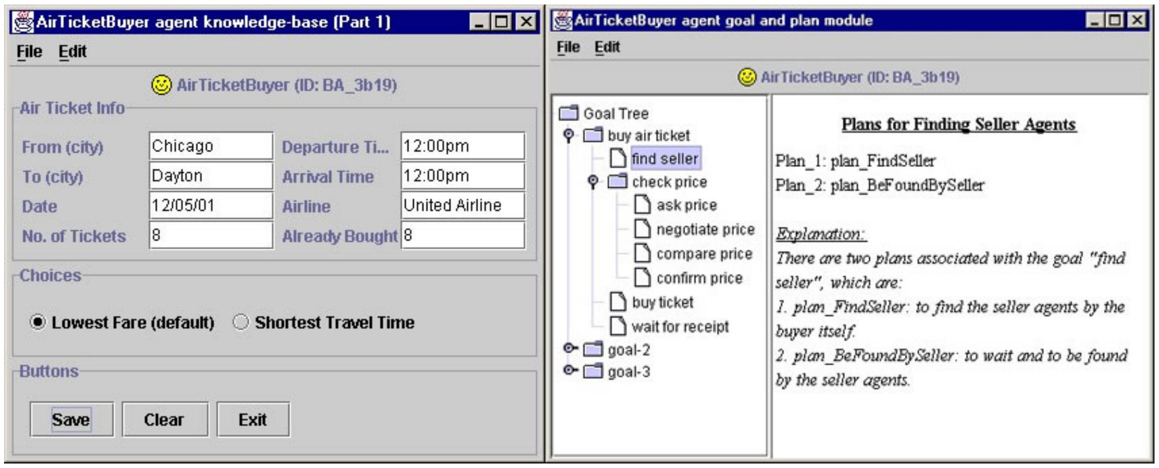
- For each goal or subgoal, choose the most appropriate plan to execute.
- Create outgoing messages and send them out through *MPU*s.
- Upon receiving incoming messages, decide to ignore or continue with the conversations.
- Decide when to update the agent's mental state.
- Upon capturing new events, update the goal list and invoke certain plans.

It should be mentioned that the above procedures could be automated, or partially automated by providing a development environment, to ease the programmers' work. This is also one of the major motivations of our ADK project. An *Agent Development Environment (ADE),* which encompasses the ADK, is envisioned as a future, and more ambitious research direction.

7.4 A Case Study: Air-Ticket Trading

We can now discuss an example that shows how to develop intelligent agents upon the ADK platform. Suppose we wish to design and implement a multi-agent system for air ticket trading. The multi-agent system will include two types of agents, air ticket seller agents and air ticket buyer agents. According to the procedures described previously, a set of goals will be identified for both the air ticket sellers and the air ticket buyers. For instance, the goal list for a simplified air ticket buyer may include the goal "*buy air ticket*," and the goal "*buy air ticket*" may have subgoals of "*find seller*," "*check price*," "*buy ticket*," and "*wait for receipt*," as shown on the right hand side of Figure 31. The air ticket seller has a similar goal list for the purpose of selling air tickets. For each goal or subgoal, we define a set of plans. For instance, for the subgoal "*find seller*", we have two plans, which are *plan_FindSeller* and *plan_BeFoundBySeller*. The plan *plan_FindSeller* can be executed to search for air ticket sellers in the Jini community, while the plan *plan_BeFoundBySeller* is executed to wait to be found by air ticket sellers. Which plan will be executed to achieve the subgoal "*find seller*" is determined by actual situations. For instance, the buyer may want to wait and be contacted by air ticket sellers initially. However, if the subgoal cannot be achieved in a period of time, the buyer can change its mind to search for air ticket sellers by itself.

The protocols used for the above two plans are fairly simple. For the plan *plan_FindSeller*, the buyer asks the sellers in the Jini community if they sell air tickets, then the sellers may reply with "Yes" or "No", or simply ignore the conversation. If a seller replies with "Yes," the buyer may ask further questions to check if the air ticket seller has enough certain types of air tickets. For instance, the buyer may ask if the seller has tickets from "Dayton" to "Chicago." If the seller has the type of air tickets that the buyer wants, the subgoal may be achieved or partially achieved (if the seller has the type of tickets but not enough). Then, in the next step, the seller continues to achieve the subgoal "*check price*."


AirTicketBuyer agent knowledge-base (Part 1)
File
Edit
Save
Clear
Exit
AirTicketBuyer agent goal and plan module
File
Edit

AirTicketBuyer (ID: BA_3b19)

Air Ticket Info

From (city)	Chicago	Departure Ti...	12:00pm
To (city)	Dayton	Arrival Time	12:00pm
Date	12/05/01	Airline	United Airline
No. of Tickets	8	Already Bought	8

Choices

[✓ Lowest Fare (default)] Shortest Travel Time

Buttons

AirTicketBuyer (ID: BA_3b19)

- Goal Tree
  - buy air ticket
    - find seller
    - check price
      - ask price
      - negotiate price
      - compare price
      - confirm price
    - buy ticket
    - wait for receipt
  - goal-2
  - goal-3

Plans for Finding Seller Agents

Plan_1: plan_FindSeller
Plan_2: plan_BeFoundBySeller

Explanation:
There are two plans associated with the goal "find seller", which are:
1. plan_FindSeller: to find the seller agents by the buyer itself.
2. plan_BeFoundBySeller: to wait and to be found by the seller agents.

Figure 31. User Interface of the *Knowledge-base*, *Goal* and *Plan* module

The following pseudo-code gives some examples of how to "fill out" certain sections of the implementation pattern provided by the *ApplicationSpecificAgent* class. Now we list a few *MPU*s that correspond to the above two plans:

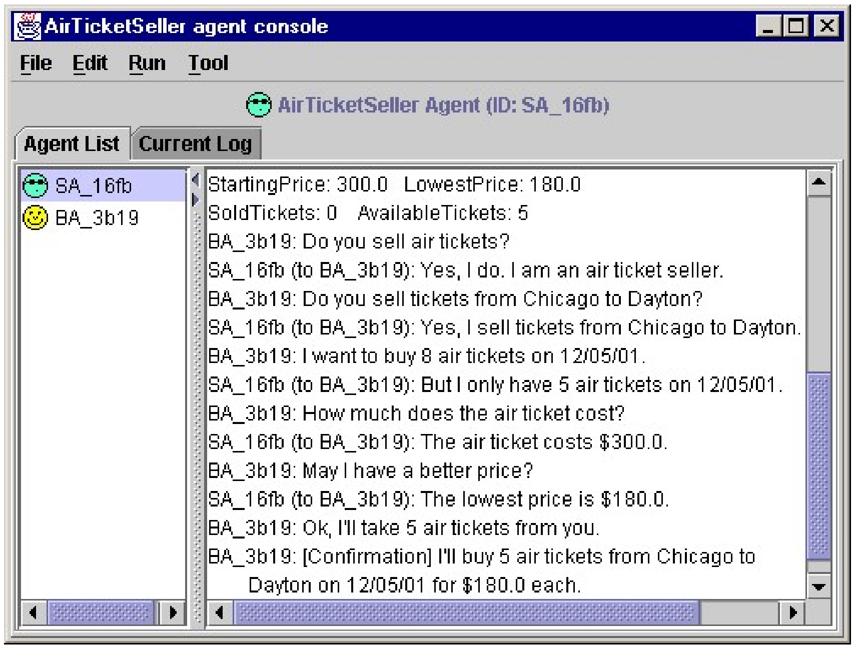
```
// incoming message section
//
// plan_FindSeller
protected void MPU_In_SellerYesOrNo(Message message) {}
…

// plan_BeFoundBySeller
protected void MPU_In_BeFoundBySeller(Message message) {}
…

// outgoing message section
//
// plan_FindSeller
protected void MPU_Out_FindSeller(Message outgoingMessage) {}
…

// plan_BeFoundBySeller
protected void MPU_Out_BuyerYesOrNo(Message outgoingMessage) {}
…
```

The *Knowledge-base* of a seller or buyer agent includes two parts, which provides information about the agent itself and information about other agents. For instance, the *Knowledge-base* of the buyer agent should include ticket information for the type of tickets that the buyer agent wants to buy (as shown on the left hand side of Figure 31), and ticket information for the type of tickets that other seller agents may hold. Other information, such as the protocols and agent states, may also be stored in the *Knowledge-base* of that agent. We do not show these types of knowledge in our illustrated figures. Finally, for the decision-making units for this air ticket trading application, we simply reuse those that are predefined in ADK.

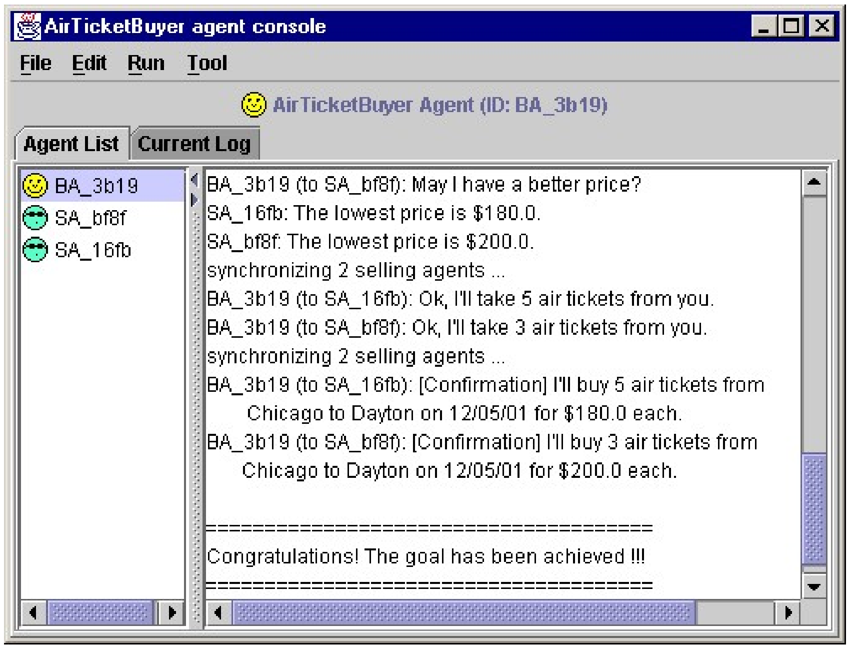

AirTicketSeller agent console
File Edit Run Tool
AirTicketSeller Agent (ID: SA_16fb)
Agent List
Current Log
SA_16fb
BA_3b19

```
StartingPrice: 300.0   LowestPrice: 180.0
SoldTickets: 0    AvailableTickets: 5
BA_3b19: Do you sell air tickets?
SA_16fb (to BA_3b19): Yes, I do. I am an air ticket seller.
BA_3b19: Do you sell tickets from Chicago to Dayton?
SA_16fb (to BA_3b19): Yes, I sell tickets from Chicago to Dayton.
BA_3b19: I want to buy 8 air tickets on 12/05/01.
SA_16fb (to BA_3b19): But I only have 5 air tickets on 12/05/01.
BA_3b19: How much does the air ticket cost?
SA_16fb (to BA_3b19): The air ticket costs $300.0.
BA_3b19: May I have a better price?
SA_16fb (to BA_3b19): The lowest price is $180.0.
BA_3b19: Ok, I'll take 5 air tickets from you.
BA_3b19: [Confirmation] I'll buy 5 air tickets from Chicago to
      Dayton on 12/05/01 for $180.0 each.
```

Figure 32. User interface of the seller agent *SA_16fb*

The user interface of a seller agent is designed as a console window as shown in Figure 32. In the agent console window, the content for the agent communication is displayed. Meanwhile, a list of agents, including the agent itself and those agents with which that agent communicates, is displayed on the left hand side of the window. The user interface will also provide a set of tools, such as to lookup existing services, to test message sending/receiving, and to edit agent properties. Figure 32 shows an example of air ticket trading process. In Figure 32, a buyer agent, with an agent ID of *BA_3b19*, first asks if the seller agent *SA_16fb* sells air tickets. After the seller agent *SA_16fb* confirms with “Yes”, the buyer agent *BA_3b19* continues to ask if the seller agent *SA_16fb* has the type of air tickets it wants. After the seller agent *SA_16fb* confirms with “Yes” again (although it does not have enough tickets), the buyer agent *BA_3b19* begins to bargain price with the seller. Finally, the conversation between agent *SA_16fb* and agent *BA_3b19* ends up with a confirmation message that the buyer agent *BA_3b19* buys all the 5 tickets from the seller agent *SA_16fb* with the price of $180.0 for each ticket. It is worth noting that although we have used natural language in this example, agents do not talk with each other in natural language. The sentences in natural language have been generated based on the values carried with messages and the semantic of their corresponding interaction protocols.

In this example, the agent ID for the seller agent or the buyer agent is defined by a prefix of *SA* (seller agent) or *BA* (buyer agent) with the last four digits of the service ID of that agent, where the service ID is a 32 digits hexadecimal number provided by the Jini community when the agent is registered [Edwards 1999][Arnold *et al.* 1999].


AirTicketBuyer agent console
File
Edit
Run
Tool
AirTicketBuyer Agent (ID: BA_3b19)
Agent List
Current Log
BA_3b19
SA_bf8f
SA_16fb

```
BA_3b19 (to SA_bf8f): May I have a better price?
SA_16fb: The lowest price is $180.0.
SA_bf8f: The lowest price is $200.0.
synchronizing 2 selling agents ...
BA_3b19 (to SA_16fb): Ok, I'll take 5 air tickets from you.
BA_3b19 (to SA_bf8f): Ok, I'll take 3 air tickets from you.
synchronizing 2 selling agents ...
BA_3b19 (to SA_16fb): [Confirmation] I'll buy 5 air tickets from
      Chicago to Dayton on 12/05/01 for $180.0 each.
BA_3b19 (to SA_bf8f): [Confirmation] I'll buy 3 air tickets from
      Chicago to Dayton on 12/05/01 for $200.0 each.

======================================
Congratulations! The goal has been achieved !!!
======================================
```

Figure 33. User interface of the buyer agent *BA_3b19*

In Figure 33, we show the user interface for the air ticket buyer agent. In this figure, we can see that the buyer agent *BA_3b19* concurrently communicates with two seller agents: *SA_bf8f* and *SA_16fb*, and buys 5 tickets from the seller *SA_16fb* and 3 tickets from the seller *SA_bf8f* with the *lowest fare* criteria.

7.5 Summary

Although a number of agent-oriented systems have been built in the past few years, there is very little work on bridging the gap between theory, systems, and application. The contribution of this chapter is to use the agent-oriented G-net model, which is a formal agent model, as a high-level design for agent development, thus we bring formal methods directly into the design phase of the agent development life cycle. Also the role of inheritance in agent development has been carefully discussed. Based on the architectural design and the detailed design of a generic intelligent agent, we developed the ADK as a class library that supports designing and implementing applications of intelligent agents for multi-agent systems. An air ticket trading example was presented to illustrate the derivation of a multi-agent application using the ADK approach. The generality of the example supports the notion that our model-based approach is feasible and effective. As a potential solution for automated software development, we summarized the procedure to generate a model-based design for application-specific agents in multi-agent systems. Therefore, an *Agent Development Environment (ADE)* to support the development process can be the vision of our future project for automating the implementation process to reduce the programming-level tasks.

8. CONCLUSIONS AND FUTURE WORK

Formal methods can be used to precisely specify the behavior of a computer system and its components, and facilitate the development of a correct implementation using automated reasoning techniques. Examples of formal methods are Z, temporal logic, and Petri nets. Among them, Petri nets are most suitable to describe and study information systems that are characterized as being concurrent, asynchronous, distributed, parallel and non-deterministic. Petri nets are a graphical and mathematical tool that allows one to build a model of a desired system, and analyze the model formally to study the behavior of the system. Many researchers have suggested object-based formal methods using high-level Petri nets. Formalisms such as LOOPN++, CO-OPN/2, and G-nets were proposed to extend the Petri net formalism into object Petri net. Among them, G-nets have been proposed with the motivation of integrating Petri net theory with the software engineering approach for system design. A notable benefit of using G-nets is its modular and object-based approach for the specification and prototyping of complex software system. The modular features of G-nets provide support for incremental design and successive modification, however the G-net formalism is not fully object-oriented due to a lack of support for inheritance. In Chapter 2, we defined an approach to extending the G-net model to support class modeling and inheritance modeling. Unlike LOOPN++ and CO-OPN/2, we use net-based extensions to capture inheritance properties, and explicitly models inheritance at the net level to maintain an underlying Petri net model that can be exploited during design simulation or analysis.

The development of agent-based systems offers a new and exciting paradigm for production of sophisticated programs in dynamic and open environments, particularly in distributed domains such as web-based systems and electronic commerce. An intelligent agent is defined as an agent that at least has the following characteristics: autonomy, reactivity, proactiveness, and sociability. Unlike most current research on formal modeling of agent systems or agent behavior, the agent model we proposed in Chapter 4, called *agent-oriented G-net* model, serves as a high-level design for agent implementation instead of a specification for agent behavior. In other words, the agent-oriented G-net model gives a software engineer by prescribing “how”, rather than “what”, to develop in terms of intelligent agents. Our formal agent model supports design modularization, asynchronous message passing and inheritance. Specifically, these major aspects of our model can be described as follows:

Modularization: The *GSP* (Generic Switch Place) serves as the only interface among agents. Agents communicate with each other by sending messages to the *GSP* of other agents. The *Goal*, *Plan* and *Knowledge-base* modules are based on the concept of BDI (belief, desire and intention) intelligent agent model. The *Planner* module represents the heart of an agent that may decide to ignore an incoming message, to start a new conversation, or to continue with the current conversation. The *Internal Structure* of an agent-oriented G-net contains message processing units *(MPU)*, and utility methods *(U-Method)* that can be invoked only by the agent itself.

Asynchronous Message Passing: Agent communications are typically based on asynchronous message passing. Since asynchronous message passing is more fundamental than synchronous message passing, we have introduced a new mechanism, called *Message Switch Place (MSP)*, to directly support asynchronous message passing.

Inheritance Modeling: In the agent-oriented G-net model, we defined the necessary facilities for inheriting functional units, such as *MPUs* and *U-Methods*. We also defined mechanisms to avoid inheriting agents' mental state, and thus, a subclass agent can be independent of its superclass agent.

Since our design model is based on the agent-oriented G-net formalism, our approach supports formal analysis and verification. In Chapter 5, we have used an existing Petri net tool to detect a design error in the original design of agents, and used model checking techniques to verify some key behavior properties of our agent model. In contrast to the common usage of formal methods in agent modeling, we use our agent-oriented G-net model as an agent design model rather than simply as a specification for agent behaviors. Based on this formal design model, we derived the agent design architecture in Chapter 7, and further implemented the *ADK* (Agent Development Kit) that provides a framework and a full class library for agent development. The *ADK* supports asynchronous message passing among agents and hides low-level communication details through middleware, e.g., Sun Jini. Application-specific agent classes can be defined as subclasses of the *Agent* class that is provided in *ADK*, and specific functionalities can be filled in through a template of the application-specific class. The procedure for building application-specific agents may be automated within a development environment *ADE* (Agent Development Environment) that envisions our future research work.

As our future research plans, we are particularly interested in the following two research areas – *Agent-Based Peer-to-Peer Computing* and *Ubiquitous Computing*:

Agent-Based Peer-to-Peer Computing: Peer-to-peer (P2P) networks are emerging as a new distributed computing paradigm for their potential to harness the computing power of the hosts composing the network and make their under-utilized resources available to others [Milojicic et al. 2002]. In a P2P system, peer and web services in the role of resources become shared and combined to enable new capabilities greater than the sum of the parts. This means that services can be developed and treated as pools of methods that can be composed dynamically. The decentralized nature of P2P computing makes it also ideal for economic environments that foster knowledge sharing and collaboration as well as cooperative and non-cooperative behaviors in sharing resources.

Meanwhile, in a multi-agent system, the interaction pattern between agents is peer-to-peer, and agents are usually characterized as cooperative and communication oriented. This fact makes a perfect match to combine agent-based computing and peer-to-peer networking. Agents can be used to embody the description of the task environments, the decision-support capabilities, the collective behavior, and the interaction protocols of each peer. My plan is to focus on the issue of how peer-to-peer computing may allow computing networks to dynamically work together by using intelligent agents. Specifically, agents reside on peer computers and communicate various kinds of information back and forth. Agents may also initiate tasks on behalf of other peer systems. For instance, intelligent agents can be used to prioritize tasks on a network, change traffic flow, search for files locally or determine anomalous behavior and stop it before it affects the network.

Ubiquitous Computing: The next generation global computer network is becoming a ubiquitous medium for communication, collaboration, and personal information management, allowing access to personalized and collaborative computing services anywhere through a variety of desktop and mobile computing devices. Such a vision can only be realized through further evolution of Internet and mobile computing technologies that support scalable resource sharing and group communication for a large number of mobile and nomadic users [Weiser 1993]. Deployment of a large-scale mobile and nomadic computing system requires a uniform treatment of these distributed systems issues, as opposed to the ad-hoc treatment that these issues receive today. Thus a uniform architecture for ubiquitous computing is necessary.

I am interested in proposing a high-level and architectural design for wide-area mobile networks using mobile agent paradigm. The mobile agent paradigm is an extension of distributed objects, exhibiting features such as active threading, run-time code mobility with autonomous navigation, and knowledge-based inter-agent communication. These characteristics favor a uniform implementation of essential mobile computing services such as multicast communication, intelligent fault-tolerant routing, proxy server/client handling, pessimistic and optimistic data replication management, and multi-level security models. Such services will enable the rapid construction and secure, scalable deployment of wide-area mobile and nomadic computing applications.

CITED LITERATURE

[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 (eds.), *Application and Theory of Petri Nets 1997*, volume 1248 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 62-81.

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

[Arnold *et al.* 1999] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath, *The Jini Specification*, Addison-Wesley, 1999.

[Ashri and Luck 2000] R. Ashri and M. Luck, “Paradigma: Agent Implementation through Jini,” In *Proceedings of the Eleventh International Workshop on Database and Expert Systems Applications*, A. M. Tjoa and R. R. Wagner and A. Al-Zobaidie (eds.), IEEE Computer Society, 2000, pp. 453-457.

[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,” In *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 (eds.), *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,” In *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency*, 16th International Conference on Application and Theory of Petri nets, Turin, Italy, June 1995.

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

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

[Bellifemine *et al.* 1999] F. Bellifemine, A. Poggi, G. Rimassa, "JADE - A FIPA-compliant Agent Framework," In *Proceedings of. 4th International Conference on the Practical Application of Intelligent Agent and Multi Agent Technology (PAAM99)*, London, U.K., 1999, pp. 97-108.

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

[Biberstein *et al.* 1997] O. Biberstein, D. Buchs and N. Guelfi, "CO-OPN/2: A Concurrent Object-Oriented Formalism," In *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, CA, 1994.

[Brazier *et al.* 1997] F.M.T. Brazier, B. Dunin Keplicz, N. Jennings, and J. Treur, "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.), In *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 (ed.), *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*, DFKI Document D-96-06, 1996.

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

[Chavez and Maes 1996] Anthony Chavez, Pattie Maes, "Kasbah: An Agent Marketplace for Buying and Selling Goods," In *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*, Vol. 8, No. 2, 1986, pp. 244-263.

[Crnogorac *et al.* 1997] L. Crnogorac, A. S. Rao, K. Ramamohanarao, “Analysis of Inheritance Mechanisms in Agent-Oriented Programming,” In *Proceedings of the 15th International Joint Conference Artificial Intelligence (IJCAI’97)*, 1997, pp. 647-652.

[Davies and Woodcock 1996] Jim Davies and Jim Woodcock, *Using Z: Specification, Refinement and Proof*, Prentice Hall International Series in Computer Science, 1996.

[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,” In *Proceedings of the 14th International Conference 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.

[D’Inverno *et al.* 1997] M. d’Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge, “Formalisms for Multi-Agent Systems,” *The Knowledge Engineering Review*, Vol. 12, No. 3, 1997.

[D’Inverno and Luck 2001] M. d’Inverno and M. Luck, “Formal Agent Development: Framework to System,” *Formal Approaches to Agent-Based Systems: First International Workshop*, FAABS 2000, Rash, J.L., Rouff, C.A., Truszkowski, W., Gordon, D., Hinchey, M.G. (eds.), Lecture Notes in Artificial Intelligence, Vol. 1871, Berlin, Springer-Verlag, 2001, pp. 133-147.

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

[Edwards 1999] W. K. Edwards, *Core Jini*, The Sun Microsystems Press, Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[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,” In *Proceedings of 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.

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

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

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

[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*, Vol. 1, No. 6, 1997, pp. 37-65.

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

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

[Fournet *et al.* 1996] C. Fournet, G. Gonthier, J. Lévy, L. Maranget, and D. Rémy, “A Calculus of Mobile Agents,” In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR’96)*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 1119, August 1996, pp. 406-421.

[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*, Kluwer Academic Publishers: Boston, MA, 1992, pp. 81-108.

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

[Gray 1995] Robert S. Gray, “Agent Tcl: A Transportable Agent System,” In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95),* Baltimore, Maryland, December 1995.

[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?” *Technical Report,* IBM Research Division, T. J. Watson Research Center, March 1995.

[Howden *et al.* 2001] N. Howden, R. Rönnquist, A. Hodgson and A. Lucas, “JACK Intelligent Agents – Summary of an Agent Infrastructure,” In *Proceedings of the 5th International Conference on Autonomous Agents*, 2001.

[Huber 1999] M. Huber, “JAM: a BDI-theoretic Mobile Agent Architecture,” In *Proceedings of International Conference on Autonomous Agents*, 1999, pp. 236-243.

[Iglesias *et al.* 1998] Carlos Argel Iglesias, Mercedes Garrijo, José Centeno-González, “A Survey of Agent-Oriented Methodologies,” In *Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Language (ATAL-98)*, 1998, pp. 317-330.

[Ioerger *et al.* 2000] T. R. Ioerger, R. A. Volz, and J. Yen, “Modeling Cooperative, Reactive Behaviors on the Battlefield Using Intelligent Agents,” In *Proceedings of the Ninth Conference on Computer Generated Forces (9th CGF)*, 2000, pp. 13-23.

[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,” *International Journal of Autonomous Agents and Multi-Agent Systems*, Vol. 1, No. 1, 1998, pp. 7-38.

[Jennings 2000] N. R. Jennings, “On Agent-Based Software Engineering,” *Artificial Intelligence*, Vol. 117, 2000, pp. 277-296.

[Johansen *et al.* 1995] Dag Johansen, Robbert van Renesse, and Fred B. Schneider, “Operating system support for mobile agents,” In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, USA, May 1995.

[Karnik and Tripathi 1998] Neeran Karnik and Anand Tripathi, “Design Issues in Mobile Agent Programming Systems,” *IEEE Concurrency*, July-September 1998, pp. 52-61.

[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,” In *Proceedings of the 4th International 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,” In W. Van de Velde and J. W. Perram (eds.), *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modeling Autonomous Agents in a Multi-Agent*

World, Lecture Notes in Artificial Intelligence, Vol. 1038, Springer-Verlag: Berlin, Germany, 1996, pp. 56-71.

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

[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", In *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," In *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," In *Proceedings of Petri Net Conference 97*, Touloure, France, 1997.

[Lange *et al.* 1997] D. B. Lange, M. Oshima, G. Karjoth, and K. Kosaka, "Aglets: Programming Mobile Agents in Java," In *Proceedings of Worldwide Computing and Its Applications (WWCA'97),* Lecture Notes in Computer Science, Vol. 1274, 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*, Vol. 20, No. 1, 1993, pp. 69-86.

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

[Luck *et al.* 1997] M. Luck, N. Griffiths and M. d'Inverno, "From Agent Theory to Agent Construction: A Case Study," In J. P. Muller, M.Wooldridge and N. R. Jennings (eds.), *Intelligent Agents III*, Lecture Notes in Artificial Intelligence, Vol. 1193, Springer-Verlag: Heidelberg, Germany, 1997.

[Manna and Pnueli 1992] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.

[Mascolo 1999] C. Mascolo, "MobiS: A Specification Language for Mobile Systems," In *Proceedings of the Third International Conference on Coordination Models and Languages,* Amsterdam, The Netherlands, April 1999, P. Ciancarini and A. Wolf (eds.), Lecture Notes in Computer Science, Vol. 1594, Springer-Verlag, pp. 37-52.

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

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

[Milojicic *et al.* 2002] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja1, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, “Peer-to-Peer Computing,” *Technical Report HPL-2002-57*, HP Lab, 2002.

[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,* Vol. 77, No. 4, April 1989, pp. 541-580.

[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*, Vol. 57-58, 1991, pp. 361-384.

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

[Nwana *et al.* 1999] H. Nwana, D. Ndumu, L. Lee, and J. Collins, “ZEUS: A Toolkit for Building Distributed Multi-Agent Systems,” *Applied Artificial Intelligence Journal*, Vol. 13, No. 1, 1999, pp. 129-186.

[Odell 2001] J. Odell, H. Van Dyke Parunak, and B. Bauer, “Representing Agent Interaction Protocols in UML,” *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge (eds.), Springer-Verlag, Berlin, 2001, pp. 121–140.

[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*, Vol. 39, No. 1, 1997, pp. 39-59.

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

[Poslad *et al.* 2000] S. Poslad, P. Buckle, R. Hadingham, “The FIPA-OS Agent Platform: Open Source for Open Standards,” In *Proceedings of 5th International Conference on the Practical Application of Intelligent Agent and Multi Agent Technology (PAAM2000),* Manchester, UK, April 2000.

[Pressman 2001] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th Edition, McGraw-Hill, 2001.

[Rao and Georgeff 1993] A. S. Rao and M. P. Georgeff, “A Model-Theoretic Approach to the Verification of Situated Reasoning Systems,” In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93),* Chambery, France, 1993, pp. 318-324.

[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.

[Rodriguez-Aguilar *et al.* 1999] J. A. Rodriguez-Aguilar, F. J. Martin, P. Garcia, P. Noriega and C. Sierra, “Towards a Formal Specification of Complex Social Structures in Multi-agent Systems,” In J. Padget, editor, *Collaboration between Human and Artificial Societies,* Lecture Notes in Artificial Intelligence, Vol. 1624, Springer-Verlag, 1999, pp. 284-300.

[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),* Vol. 14, No. 2-3, 2000, pp. 95-113.

[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”, In *Proceedings of ECOOP’96,* Lecture Notes in Computer Science, Vol. 1219, Springer-Verlag, 1996, pp. 248-274.

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

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

[Saldhana *et al.* 2001] J. Saldhana, S. M. Shatz, and Z. Hu, “Formalization of Object Behavior and Interactions from UML Models,” *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, Vol. 11, No. 6, December 2001, pp. 643-673.

[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*, Vol. 60, No. 1, March 1993, pp. 51-92.

[Siegel *et al.* 2001] J. Siegel, and the OMG Staff Strategy Group, "Developing in OMG's Model Driven Architecture (MDA)," *OMG White Paper*, Object Management Group, November 2001.

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

[Smith 1980] R. G. Smith, "The contract net protocol: high-level communication and control in a distributed problem solver," *IEEE Transactions on Computer*, Vol. C-29, December 1980, pp. 1104-1113.

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

[Stamos and Gifford 1990a] James W. Stamos and David K. Gifford, "Remote Evaluation," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 4, 1990, pp. 537-565.

[Stamos and Gifford 1990b] James W. Stamos and David K. Gifford, "Implementing Remote Evaluation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 7, 1990, pp.710-722.

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

[Straßer *et al.* 1997] M. Straßer, J. Baumann, and F. Hohl, "Mole – A Java based Mobile Agent System," In M. Mühlhäuser (ed.), *Special Issues in Object Oriented Programming*, dpunkt Verlag, 1997, pp. 301-308.

[Tay and Ananda 1990] B. H. Tay and A. L. Ananda, "A Survey of Remote Procedure Calls," *Operating Systems Review*, Vol. 24, No. 3, July 1990, pp. 68-79.

[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.

[Vasconcelos *et al.* 2002] W. Vasconcelos, J. Sabater, C. Sierra and J. Querol, "Skeleton-Based Agent Development for Electronic Institutions," In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Italy, July 2002.

[Vidal *et al.* 2001] J. M. Vidal, P. A. Buhler, and M. N. Huhns, “Inside an Agent,” *IEEE Internet Computing*, Vol. 5, No. 1, January-February 2001.

[Weiser 1993] M. Weiser, “Some computer science issues in ubiquitous computing,” *Communications of the ACM (CACM),* Vol. 36, No. 7, July 1993, pp. 74-83.

[Wermelinger and Fiadeiro 1998] M. Wermelinger, J. L. Fiadeiro, “Connectors for Mobile Programs,” *IEEE Transactions on Software Engineering*, Vol. 24, No. 5, May 1998, pp. 331-341.

[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*, Vol. 3, No. 3, 2000, pp. 285-312.

[Wooldridge and Ciancarini 2001] M. Wooldridge and P.Ciancarini, “Agent-Oriented Software Engineering: The State of the Art,” In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, Lecture Notes in Artificial Intelligence, Vol. 1957, Springer-Verlag, January 2001.

[Wooldridge 2002] M. Wooldridge, *An Introduction to Multiagent Systems*, John Wiley and Sons, Ltd., 2002.

[White 1995] J. E. White, “Telescript Technology: An Introduction to the Language”, *White Paper*, General Magic, Inc., Sunnyvale, CA, 1995.

[Xie 2000] X. Xie, *Design Support for State-Based Distributed Object Software*, Ph.D. Thesis, EECS Department, 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,” In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC 2000),* 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,” In *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, March 2001, Dallas, Texas, USA, pp. 11-18.

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

[Xu and Shatz 2003] H. Xu and S. M. Shatz, "A Framework for Model-Based Design of Agent-Oriented Software," *IEEE Transactions on Software Engineering (IEEE TSE)*, Vol. 29, No. 1, January 2003, pp. 15-30.

[Xu *et al.* 2002] D. Xu, R. A. Volz, T. R. Ioerger, and J. Yen, "Modeling and Verifying Multi-Agent Behaviors Using Predicate/Transition Nets," In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, Italy, July 2002, pp. 193-200.

[Xu *et al.* 2003] D. Xu, J. Yin, Y. Deng, and J. Ding, "A Formal Architectural Model for Logical Agent Mobility," *IEEE Transactions on Software Engineering*, Vol. 29, No.1, January 2003, pp. 31-45.

[Yan 2002] Y. Pan, *Refinement of an Agent-Based Model to support Decision Making and Standard Agent Communication Languages*, Masters Thesis, University of Illinois at Chicago, November 2002.

[Yen *et al.* 2001] J. Yen, J. Yin, T.R. Ioerger, M. Miller, D. Xu, and R.A. Volz, "CAST: Collaborative Agents for Simulating Teamwork," In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, Seattle, WA, August 2001, pp. 1135-1142.

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

VITA

NAME	Haiping Xu
EDUCATION	B.S., Electrical Engineering, Zhejiang University, Hangzhou, China, 1989
	M.S., Electrical Engineering, Zhejiang University, Hangzhou, China, 1992
	M.S., Computer Science, Wright State University, Dayton, Ohio, 1998
	Ph.D., Computer Science, University of Illinois, Chicago, Illinois, 2003
WORK EXPERIENCE	Concurrent Software Systems Laboratory, University of Illinois at Chicago, Chicago, Illinois, 1998 – 2003
	Parallel Computing Laboratory, Wright State University, Dayton, Ohio, 1996 – 1998
	Intelligent Systems Laboratory, Nanyang Technological University, Singapore, 1996
	Hewlett-Packard Company, Beijing, China, 1995 – 1996
	Shen-Yan Systems Technology, Inc., Beijing, China, 1992 – 1995
TEACHING EXPERIENCE	Department of Computer Science University of Illinois at Chicago, Chicago, Illinois, 1999 – 2003
HONORS	University Fellowship, University of Illinois, Chicago, Illinois, 2001
	Dayton Area Graduate Studies Institute (DAGSI) Scholarship, Ohio, 1998
	Excellent Graduate Student of Zhejiang University, China, 1992
	Guang-Hua National Merit Scholarship, China, 1990
	Graduate Scholarship of Zhejiang University, China, 1990
	Zhejiang University Special Class for Gifted Young, China, 1985
PROFESSIONAL MEMBERSHIP	Member of IEEE, IEEE Computer Society, and IEEE SMC Society Member of Association for Computing Machinery (ACM)
REFEREE	IEEE Transactions on Multimedia (IEEE TMM)
	IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)
	International Journal of Software Engineering & Knowledge Engineering (IJSEKE)
	International Conference on Distributed Computing Systems (ICDCS)
	International Conference on Application and Theory of Petri Nets (ICATPN)
	International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE)

PUBLICATIONS OF THE AUTHOR

[1] H. Xu and S. M. Shatz, “ADK: An Agent Development Kit Based on a Formal Model for Multi-Agent Systems,” Submitted to *Automated Software Engineering*, February 2003.

[2] H. Xu and S. M. Shatz, “A Framework for Model-Based Design of Agent-Oriented Software,” *IEEE Transactions on Software Engineering (IEEE TSE)*, Vol. 29, No. 1, January 2003, pp. 15-30.

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

[4] H. Xu and S. M. Shatz, “An Agent-Based Petri Net Model with Application to Seller/Buyer Design in Electronic Commerce,” In *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, March 2001, Dallas, Texas, USA, pp. 11-18.

[5] H. Xu and S. M. Shatz, “Extending G-Nets to Support Inheritance Modeling in Concurrent Object-Oriented Design,” In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC 2000)*, October 8-11, 2000, Nashville, Tennessee, USA, pp. 3128-3133.

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

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

[8] K. Warendorf, H. Xu, and A. Verhoeven, “Case-Based Instructional Planning for Learning in a Context,” In *Proceedings of the Joint 1997 Pacific Asian Conference on Expert Systems / Singapore International Conference on Intelligent Systems (PACES/SPICIS 97)*, February 24-27, 1997, Singapore, pp. 354-360.

[9] H. Xu, *The Basic Study and Partial Implementation of Intelligent Chinese Tutoring System*, Masters Thesis, Zhejiang University, Hangzhou, China, January 1992.

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

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

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

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

[14] H. Xu, “Software Design of a Microcomputer-Based Nuclear Scaler,” *Process Automation Instrumentation*, 1991, Vol. 12, No. 10, pp. 13-16.

[15] X. Ruan, H. Xu, and Z. Chen, “Intelligent CAI and Chinese Tutoring System,” *Communications of Computation and Information*, June 1991, No. 7, pp. 6-14.

A MODEL-BASED APPROACH FOR DEVELOPMENT OF MULTI-AGENT SOFTWARE SYSTEMS

Haiping Xu, Ph.D.
Department of Computer Science
University of Illinois at Chicago
Chicago, Illinois (2003)

The advent of multi-agent systems has brought opportunities for the development of complex software that will serve as the infrastructure for advanced distributed applications. During the past decade, there have been many agent architectures proposed for implementing agent-based systems, and also some efforts to formally specify agent behaviors. However, research on narrowing the gap between agent formal models and agent implementation is rare. In this thesis, we present a model-based approach to designing and implementing multi-agent software systems. Instead of using formal methods only for the purpose of specifying agent behavior, we bring formal methods into the design phase of the agent development life cycle. Our approach is based on the G-net formalism, which is a type of high-level Petri net defined to support modeling of a system as a set of independent and loosely-coupled modules.

We first introduce how to extend G-nets to support class modeling and inheritance modeling for concurrent object-oriented design. Then, by viewing an agent as an extension of an object with mental states, we derive an agent-oriented G-net model from our extended G-nets that support class modeling. The agent-oriented G-net model serves as a high-level design for intelligent agents in multi-agent systems. To illustrate our formal modeling technique for agent-oriented software, an example of an agent family in electronic commerce is provided. We show how an existing Petri net tool can be used to detect design errors, and how model checking techniques can support the verification of some key behavioral properties of our agent models. In addition, we adapt the agent-oriented G-net model to support basic mobility concepts, and present

design models of intelligent mobile agents. Finally, based on the high-level design, we derive the agent architecture and the detailed design needed for agent implementation. To demonstrate the feasibility of our approach, we describe a toolkit called ADK (Agent Development Kit) that supports rapid development of application-specific agents for multi-agent systems.