3-31-2004

A methodology for formally modeling and analyzing software architecture of mobile agent systems

Junhua Ding
Florida International University

DOI: 10.25148/etd.FI14062292

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A METHODOLOGY FOR FORMALLY MODELING AND ANALYZING SOFTWARE ARCHITECTURE OF MOBILE AGENT SYSTEMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Junhua Ding

2004

To: Dean R. Bruce Dunlap
College of Arts and Sciences

This dissertation, written by Junhua Ding, and entitled A Methodology for Formally Modeling and Analyzing Software Architecture of Mobile Agent Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Raimund K. Ege

Jie Mi

Shu-Ching Chen

Yi Deng

Xudong He, Major Professor

Date of Defense: March 31, 2004

The dissertation of Junhua Ding is approved.

Dean R. Bruce Dunlap
College of Arts and Sciences

Dean Douglas Wartzok
University Graduate School

Florida International University, 2004

ACKNOWLEDGMENTS

I wish to thank the members of my committee for their support, suggestions and patience. Their gentle but firm direction has been most appreciated. I particularly thank my major professor Dr. Xudong He for encouraging and instructing me to conduct this research and in guiding me toward a qualitative methodology. I must thank Dr. Yi Deng, Dr. Shu-Ching Chen, Dr. Jie Mi and Dr. Raimund K. Ege who gave this dissertation a very careful and intelligent reading.

I owe special thanks to Dr. Dianxiang Xu, who contributed substantially to the ideas in this dissertation. He helped me to solidify ideas in countless discussions. Thanks to Zhengfan Dai and Shu Gao for their helpful works in the draft review and system implementation. Thanks to the group in CADSE, who gave me constructive comments and continuing support. I would like to thank Will Franco, David Frazier and the group in university graduate school who read and commented on drafts of this dissertation. To Chun Huang, the one indispensable person in the whole process – I cannot thank you enough.

This research was partially supported by NSF grant HRD-0317692 and NASA grant NAG2-1440.

ABSTRACT OF DISSERTATION

A METHODOLOGY FOR FORMALLY MODELING AND

ANALYZING SOFTWARE ARCHITECTURE OF MOBILE AGENT SYSTEMS

by

Junhua Ding

Florida International University, 2004

Miami, Florida

Professor Xudong He, Major Professor

A methodology for formally modeling and analyzing software architecture of mobile agent systems provides a solid basis to develop high quality mobile agent systems, and the methodology is helpful to study other distributed and concurrent systems as well. However, it is a challenge to provide the methodology because of the agent mobility in mobile agent systems.

The methodology was defined from two essential parts of software architecture: a formalism to define the architectural models and an analysis method to formally verify system properties. The formalism is two-layer Predicate/Transition (PrT) nets extended with dynamic channels, and the analysis method is a hierarchical approach to verify models on different levels. The two-layer modeling formalism smoothly transforms physical models of mobile agent systems into their architectural models. Dynamic channels facilitate the synchronous communication between nets, and they naturally capture the dynamic configuration and agent mobility of mobile agent systems. Component properties are verified based on transformed individual components, system properties are checked in a simplified system model, and interaction properties are analyzed on models composing from involved nets. Based on the formalism and analysis method, this researcher formally modeled and analyzed the software architecture of mobile agent systems, and designed an architectural model of a medical information processing system based on mobile

agents. The model checking tool SPIN was used to verify properties such as reachability, concurrency and safety of the medical information processing system.

From successful modeling and analyzing the software architecture of mobile agent systems, the conclusion is that PrT nets extended with channels are a powerful tool to model mobile agent systems, and the hierarchical analysis method provides a rigorous foundation for the modeling tool. The hierarchical analysis method not only reduces the complexity of the analysis, but also expands the application scope of model checking techniques. The results of formally modeling and analyzing the software architecture of the medical information processing system show that model checking is an effective and an efficient way to verify software architecture. Moreover, this system shows a high level of flexibility, efficiency and low cost of mobile agent technologies.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

Introduction

1 Background

Formally modeling and analyzing software architecture has profound impact on the development of high quality software systems. A rigorous approach toward architectural level system design can help to detect and eliminate design errors as early as possible in the development cycle, to avoid costly fixes at the implementation stage, and thus to reduce overall development cost and to improve the quality of the systems. A formal and rigorous way to model and analyze software architecture is required to achieve the above advantages. Software architecture is the overall structure and organization of software systems. With the increase in the size and complexity of software systems, the design problem goes beyond using better algorithms and data structures. Designing and specifying the overall system structure and organization becomes more important [GS93]. In order to define software architecture, an architecture description language (ADL) is required to define system architectural models, and an analysis technique is needed to verify system properties. There are many ADLs, but research on software architecture development and analysis techniques is still not enough [Sha01]. This dissertation provides a formalism to model mobile computing systems especially mobile agent systems, and proposes a systematic analysis method to analyze software architecture of mobile agent systems.

A mobile agent is an executing program representing its users and is capable to migrate from one node to another in a network, and, thus, is able to execute at different locations during its life span [GCK01]. A mobile agent system consists of mobile agents, and agent support systems that support agent executions. Mobile agent systems are useful to conserve bandwidth, reduce total completion time and latency, support dynamic load balancing, support offline operation in mobile computing environments, and support dynamic deployment [XYD03]. It has potential applications in fields like on-demand software systems, interactive training systems, and data mining systems. Formally modeling and analyzing software architectures of mobile agent systems not only help further understand mobile agent systems, but also facilitate developing high quality applications based on mobile agent techniques.

Petri nets are a popular formal approach with graphical and mathematical notations, noted for its many advantages on the behavioral specification and analysis of distributed concurrent systems, and it is a promising tool for studying mobile agent systems that are characterized as being concurrent, asynchronous, distributed, parallel, and non-deterministic. Thus, Petri nets can serve as a powerful medium of communication between practitioners and theoreticians: practitioners can learn from theoreticians on how to make their models more methodical, and theoreticians can learn from practitioners on how to make their models more realistic [Mur89]. Predicate/Transition (PrT) nets are a high level formalism of Petri nets that are especially suitable for agent system modeling due to its similarity to a logic agent system, and efficient reachability analysis [XYD03][BFF95]. PrT net models are much more compact and abstract than low-level Petri net models. It brings us a convenient way to model complex distributed systems and enables us to focus on important and interesting system properties. Temporal Logic is a formalism for describing state changes or sequences of transition firings in a reactive system. Linear temporal logic (LTL) is a common formalism to specify properties of reactive systems. It has sufficient expressive power for most purposes, but with relatively simple syntax and semantics. LTL is interpreted over infinite executions that make it appropriate to specifying properties of the executions of Kripke structure. We will model the behavior of mobile agent systems using PrT nets extended with dynamic channels, and specify system properties using LTL.

Theorem proving, testing, model checking and simulation are most popular approaches to analyze software architectures. Theorem proving demands user interactions during proving and the tedious and difficult works make it unsuitable to verify complex systems. Testing software architectures needs complex support environments to support the model executions , but it cannot guarantee the system correctness. Simulation suffers from the same problems as testing. On the other hand, model checking is a powerful technique for analyzing software architectures and the verification process is completely automatic. Model checking techniques have been successfully used in mission critical system development [CHO02] [PMH02], and it has become an important verification method in hardware development. In our previous work, we successfully used a model checking tool called SMV [CGP99] to find an error in a flexible manufacturing system (FMS) model [HDD02] [Din00]. However, model checking techniques suffer from the state-space explosion problem, because some systems are composed of many parallel processes and in general, the size of the state-space grows exponentially with the number of processes [GL94]. According to the approaches to address this issue, model checking can be classified as symbolic verification and explicit verification. Symbolic verification such as SMV uses symbolic representations for sets of states and transition relations, and can check very large state space (10^{100} or more) systems. Explicit verification model checking techniques such as SPIN use partial order to reduce state-space, and they are more powerful in software verification than symbolic model checking techniques in this field [EP02]. In this dissertation, we propose a hierarchical analysis method to address the state-space explosion issue when analyzing the software architecture of mobile agent systems. This method is particularly effective when it is integrated with model checking tool SPIN.

However, existing works on modeling mobile systems using Petri nets (high-level or low-level Petri nets) [FB98] [KMH03] [KMR01] [KR01] [MK96] [MW97] [XD00] [XS03] [XYD03] cannot naturally capture the mobility and dynamic reconfiguration property of mobile systems.

Most works focus on modeling and analyzing specific property such as mobility, cooperation or security, but modeling the system architecture is not enough. In addition, to best of my knowledge, there is not a systematic analysis method to formally analyze the Petri net software architecture of mobile agent systems.

2 Scope of the Dissertation

In this dissertation, I propose a methodology for formally modeling and analyzing software architecture of mobile agent systems. The goal is to define a formalism to model the software architecture of mobile agent systems, and then to propose a systematic method to analyze the architectural model. The formalism has the expressive power to naturally model the architecture of mobile agent systems, and easily capture the properties especially the mobility and dynamic configuration of mobile agent systems. The systematic analysis method provides a formally and effective approach to analyze the software architecture of mobile agent systems, and it provides a solid foundation for the architecture. In order to achieve this goal, the following works are necessary. First, we extend PrT nets with dynamic channels (We also call the nets as CPrT nets), which build the communication links at run time. Second, we model and analyze a software architecture of mobile agent systems using CPrT nets with two-layer framework. Third, we provide a hierarchical analysis technique for analyzing the software architecture of mobile agent systems. Finally, we use the extended PrT nets to model a medical information processing system based on mobile agents, and analyze the model using model checking tool SPIN based on the hierarchical analysis technique.

2.1 Modeling Mobile Agent Systems

The primary identifying characteristic of mobile agents is their ability to autonomously migrate from one computer to others in a network. Thus, support agent mobility is a fundamental requirement of mobile agent systems. Even agent migration can be naturally simulated by transition firing of Petri nets. The migration of mobile agents leads to the dynamic configuration of the software architecture of mobile agent systems. It might be complex and even difficult to represent agent migration and dynamic connection between agents and their environments. Tokens in PrT nets are passive, whereas agents are active. It is a challenge to naturally model and analyze the dynamic property. In order to model the dynamic connection or reconfiguration of mobile agent systems, we extended PrT nets with dynamic channels, which dynamically connect agents with their environments according to their contexts at run time. Although CPrT nets do not improve the expressive power of PrT nets, it provides a more flexible and powerful means to model the dynamic property of mobile systems with more compact and more easily understand models. In addition, to bridge the gap between tokens and agents, a two-layer approach (EOS) [Val98] is chosen to model mobile agent systems. In the framework for modeling mobile agent systems, we chose CPrT nets to define system behavior models, and LTL to define system properties. In this method, agents are modeled as object nets that are wrapped as tokens in system nets that representing the running environments of the agents, and communications between object nets and system nets are through channels. We extend EOS from three aspects: 1. We use CPrT nets instead of low-level place transition nets. This makes our model more compact and easier to understand. 2. We use dynamic channels instead of textural labels on transitions to facilitate the interaction and communication between object models and system models. The channel method is more flexible and easier to model the synchronous communication of mobile systems since communication links between transitions are built at run time and data exchanges are through channel parameters, but labels on transitions in EOS are only for synchronization between transitions. More importantly, we chose dynamic channels instead of static textural labels for transitions, so that it brings a dynamic property to the static structures of PrT nets. The values of channels are decided by their contexts at run time, which is different from existing channels for Petri nets. 3. In EOS, it has to use process markings to deal with the object fork-joint situation. Each object has to remember its state path, and then the jointed state is calculated based on the least upper bound of the jointed object processes. However, in our method, each object only needs to keep its current state since objects are independent from each other.

2.2 Analyzing Mobile Agent Systems

In order to support the formal analysis of the software architecture of mobile agent systems, we propose a hierarchical analysis method, which verify system properties at different levels depending on properties. For some properties, it is not necessary to check the whole model but only certain individual components; some properties are verified involving several different models on different levels, and the system properties are verified based on a simplified system model forming from individual models in the system level. This is reasonable since agents or the agent systems are relatively independent programs, and even the interaction between an agent and its system might be reduced to an agent model with its interface, which represents the environment if our interested properties are on the agent. We classify three levels of analysis: the system level analysis, the component level analysis, and the interaction level analysis. System level analysis is used to verify system properties, and the model consists of all high-level models. Component-level analysis is used to analyze individual component properties, which are on these components. Interaction-level analysis is used to check properties that involve models in two different levels, such as the interaction between agents and their systems. Since each component is a part of the system model, we have to transform each component model as an independent model, and the system-level model is simplified based on verification results of their components. If we check an agent internal behaviors or tasks, we only need to check the agent model. These behaviors include agent receiving or sending messages, updating itineraries, scheduling tasks etc. Similar to agent environments, we only check the supporting system model for internal behaviors. These behaviors include receiving or transferring an agent, restarting agents, terminating agents etc. In our analysis approach, the checking for agent mobility is applied to system-level without considering unfolding agent tokens since agents are inactive during migration. However, the analysis on cooperation property has to be applied on the model consisting of both agents and their supporting systems. The mobility involves different agent support systems so that it has to be checked on the model that is composed from connecting agent support system models but keeping agents as wrapped tokens.

The high-level Petri nets models such as PrT models are much more compact and abstract than low-level Petri nets models or SPIN model. However, it also complicates analysis. In order to validate models using SPIN verifier, we must translate the net models into SPIN acceptable models -- Promela programs. It is straightforward to translate low-level Petri nets models into Promela programs. From intuition, we can translate a high-level Petri nets model into a low-level model, and then translate the low-level Petri nets model into a Promela program. However, even though it is possible to translate high level Petri net models into low level Petri net model in theory, it is not practical to do so since there is not a good general way for the translation except unfolding high level models. The unfolding method is a tedious and sometimes impossible task if some predicate types are infinite [GP98]. Therefore, we limit predicate type into finite, and each element on the arc label is an enumerable type. The basic idea of translating PrT nets into Promela is to translate predicates in nets into variables in Promela, and translate each transition from nets into an atomic sequence. Global variables and channel variables are used to synchronize different processes, which represent different nets.

2.3 An Application of Mobile Agent Systems

Although mobile agent technologies are an active research topic in last decade, there are only few killer applications of this technology. It is important to find a practical and convincing application of mobile agent technologies to prompt further researches. In this dissertation, we describe a practical application system -- a medical information processing system based on mobile agents. The system is difficult to be implemented using traditional solutions, but it can be nicely implemented using mobile agent technologies. We design the system infrastructure, and formally model and analyze the software architecture of the system using CPrT nets and hierarchical analysis method.

2.4 Contributions

The principle contributions of this work are described below:

1. *PrT nets extended with dynamic channels for synchronous communication.* The dynamic channels are nicely integrated with the static structures of PrT nets. Dynamic channels provide a powerful and natural mechanism to facilitate the communication between mobile agents and environments. We prove that the behavior equivalence between the extended PrT nets and ordinary PrT nets, and we show that the extended PrT nets provide a more flexible approach to model synchronous communication between nets, especially the communication between nets on different levels and mobile objects. Comparing to existing works on Petri nets, PrT nets extended with dynamic channels is an original work with some advantages.

2. *A Hierarchical analysis method for analyzing software architectures.* We propose a method to analyze software architecture of mobile agent systems using component level analysis, composition level analysis and system level analysis according to different properties. The hierarchical analysis method provides a solid foundation for the modeling method using CPrT nets. This method reduces the analysis complexity, and expands the application scope of model checking techniques.

3. *An architectural model of mobile agent systems.* We define an architectural model of mobile agent systems, which follow the MASIF specifications [OMG98]. The model is more compact and easy-to-understand comparing to existing Petri nets models. In addition, we formally analyze the mobility and dynamic configuration property of mobile agent systems based the two-layer CPrT nets model. The model and analysis method are helpful to further study mobile agent systems, and it is useful to develop high quality applications based on mobile agents.

4. *A medical information processing system based on mobile agents*. We design an application for medical information processing system based on mobile agents. We define the software architecture of the application using two-layer CPrT nets, and analyze the software architecture using model checking tool SPIN based on hierarchical analysis method. The medical information processing system not only demonstrates the capacity of the methodology for formally modeling and analyzing software architecture of mobile agent systems, but also provides a convincing example for mobile agent technologies.

CHAPTER II

Modeling and Analyzing Mobile Agent Systems

1 Mobile Agent Systems

Mobile agent systems have become one of the most active research areas on distributed systems since the early 1990s. A mobile agent system consists of a finite set of agent support systems and a group of mobile agents. A mobile agent is an autonomous executable program that represents its users to migrate and compute from hosts to hosts in networks. It has its own task and executes the task in destination hosts and continuing its running on other hosts according to its schedule or execution results. The unique characteristic of mobile agents is the active mobility, which is different from passive mobile programs such as the applets. In order to support the execution of mobile agents, each host needs an agent support system or agent system to support particular types of mobile agents. An agent system is a server program that resides at a host where mobile agents might visit. Each agent system can create, execute, transfer and terminate agents. Moreover, it offers one or more services to mobile agents that enter it [Obj97] [Gay96]. There are many different mobile agent systems for different research or application purposes. These systems differ widely in architecture and implementation, such as implementation languages, communication mechanisms, authentication methods, and whether they support strong mobility,. These differences are impeding interoperability, and rapid proliferation of mobile agent technology. In order to solve these problems, there are two main standardization efforts on mobile agent systems. The first one is the Foundation for Intelligent Physical Agents (FIPA) [FIP00], which defines standard interfaces for all different types of mobile agent systems. Another is Mobile Agent System Interoperability Facilities (MASIF) defined by Object Management Group

(OMG) [OMG98], which defines basic functions or facilities to construct a mobile agent system, and the common interfaces for interoperability between mobile agent systems.

FIPA is a non-profit organization to promote the development of specifications of generic intelligent agent technology and improve the interoperability of agent-based applications. FIPA standards specify the interfaces of different components in the environments with which an agent can interact. The specification includes four parts: agent management, agent communication, agent software integration, and reference applications. The agent management defines basic system management, mobility support and security management. The agent communication describes the interaction between human and agents, an ontology service and an agent communication language (ACL). The software integration and reference applications provide application cases to improve the implementation and application of this specification.

MASIF specifies two interfaces: MAFAgentSystem for agent transfer, management, and MAFFinder for the naming and locating of agents. In order to support the interoperability between different mobile agent systems, it standardizes the following four areas: agent management, agent transfer, agent and agent system names, and agent system type and location syntax. Agent management defines standard operations to mange agents, such as creating, starting, and terminating an agent. Agent transfer defines methods to migrate and receive agents and different transferring types. Agent and agent system names standardize syntax and semantics of agent and agent system names to allow agent systems and agents to identify each other, as well as clients to identify agents and agent systems. MASIF supports agent tracking, which locates agents on different agent systems registered at MAFFinders through these agent and agent system names. The agent system type is defined so that each agent system can easily decide whether it supports a particular type of agents. The location syntax is standardized so that the agent systems can locate each other. The MASIF only provides the features required for transporting standardized information between agent systems. It does not address how each agent system deals with this information internally since it is the implementation issue.

As a general paradigm for implementing distributed systems, mobile agent systems have been demonstrated beneficial in several areas of applications such as workflow management, distributed information retrieve, and automated software installation. Even though all of these applications could be developed using traditional techniques, the mobile agent technique provides a single infrastructure so that many distributed applications can be implemented easily, efficiently and robustly. The paper [GCK01] listed six strengths of mobile agents. 1. Conservation of bandwidth. Since a mobile agent migrates to the destination computers or servers to operate locally, it is not necessary to send intermediate results to clients but just the results. However, the agent's code may be larger than the total of intermediate results in some cases. The system should estimate the potential bandwidth usage, and then decide whether to use mobile agent technique or traditional client/server solution. 2. Reduction in total completion time. If a client requests a service which needs many operations in the server, and the interaction of intermediate results between the client and server is required, mobile agent runs in the server locally will reduce the total completion time. 3. Reduction in latency. This works for the application that must react quickly to some events by sending out new status or control information. In such case, if the reactive component is implemented as a mobile agent, it can move closer to the producer of the event producers and migrate with the producers, so the event information can be captured and sent back to clients much faster. 4. Support disconnected operation and mobile computing. A mobile agent is a relatively independent program, and as soon as it arrives at destination computer, it can start its process without interaction with the clients until it needs to send back the results. 5. Support dynamic load balancing. Load balancing aims to improve performance by partitioning a task into components and distributing them across multiple processors. Since mobile agents can move across the platforms with application-specific code, they naturally support dynamic redistribution of computing components. 6. Support dynamic deployment. A mobile agent can move itself to a remote sever and install itself to provide services there.

Even though each one of them can be realized efficiently using traditional techniques, mobile agent technology has all six strengths. The future direction for mobile agents will be expanded to mobile codes. Mobile codes include not only the active mobile agents, but also passive migration codes such as applets, and component-based mobile agents. The component-based mobile agent technique will not move monolithic mobile agents to destinations; instead, it only moves some core codes and a script program that is used to assemble agents in the destination computers or servers, which have code bases for reconstructing agents. In addition, there will be more and more middleware implemented using mobile agents to improve system performance and reliability through using dynamic load balancing, dynamic deployment and disconnected operation [GCK01].

1.1 Basic Concepts

1.1.1 Mobile Agent

An agent is an object that acts autonomously on behalf of a person or organization. Each agent has its own thread of execution so that it can perform task of its own volition. A mobile agent can move from hosts to hosts according its plan. Each mobile agent has its own task, but it also has some common functions. A mobile agent includes at least the following parts: 1. An agent identity. Since each mobile agent migrates in the networks and communicates with other agents or systems, an agent requires a unique identity value to identify a particular agent instance. 2. The agent owner. Since each agent represents its users, the agent owner is an important factor to decide its authority. 3. The agent itinerary. Since mobile agent migrates in networks, it needs an itinerary to decide its visiting paths. The itinerary could be assigned to agents before they move out, or agents could query a special directory service at the host to dynamically update its itinerary. 4. Code. Each agent will fulfill some tasks, and the code is the program that will run in the destination host to provide some services. 5. Agent authority. Since mobile agents migrate to different places, each agent needs an authority attribute, and the authority identifies the person or organization for which the agent acts. An authority must be authenticated. 6. The agent execution state. An agent's execution state is its runtime state, including program counter and fame stacks, which is encapsulated with agents and maybe resumed in destination hosts. 7. The requirements for resources. The destination servers can provide reasonable resources for agent's execution according its resource requirements. 8. History information. Agents log their visiting information to help users to process the results or dialog some failures.

1.1.2 Agent System

An agent system is a platform that can create, interpret, execute, transfer and terminate agents. Each agent system has an authority that identifies the person or organization for which the agent system acts. Each agent system has a name and it is uniquely identified by its name and address. A host might contain one or more agent systems to support different types of agents.

The functions of an agent system include [OMG98]: 1. Creating an agent. An agent system creates an agent according to requirements, assigns unique identity and authority for the agent, and might associate an itinerary or moving algorithms for the agent. 2. Transferring and running an agent. It includes initiating an agent transfer, receiving an agent, and transferring classes. When an agent system initiatizes an agent transfer, it firstly suspends the agent if the agent is running, then encapsuates the agent's state, serializes the instance of the agent classes and states, encodes the serialized agent, authenticates client and finally transfers the agent. When an agent system receives an agent which it can interpret, it accepts the agent and firstly authenciates the client, then decodes the agent, deserializes the agent classes and states, instantiates the agent, restores the agent states, and finally resumes the agent's execution. Class transfer is the ability to transfer class information from one agent system to another. This ability is a requirement in agent systems that support object-oriented agents. Classes can be transferred automatically or transferrd by request or on-demand since classes might not be transferred as a whole when the agent was transferred. 3. Finding a mobile agent. When an agent wants to communicate with other agents, it must be able to find the destination agent system to establish communication with the party. The ability to locate a particular mobile agent is also important for agent management since the migration of agents should be under control. Because mobile agents can travel in networks at any time, an agent name must be unique across all agent systems. 4. Ensuring a secure environment for an agent operations. Since a mobile agent is a computer program that can travel among agent systems, a mobile agent is often compared to a virus. It is imperative for agent systems to identify and screen incoming agents. An agent system must protect resources including its operating system, file systems, disks, CPUs, memory, other agents, and access to local programs. To ensure the safety of system resources, an agent system must identify and verify the authority that is associated with the agent. The ability to identify the authority of an agent enables access control and agent authentication within an agent system. And activity confidentiality is also one of the issues for mobile agent security. 5. Terminating an agent. An agent owner can terminate its agents' execution for any reason, and the agent system has the ability to move any guest agent out for performance or security reasons.

1.1.3 Communication and Cooperation

In order to support cooperation between agents or agents and agent systems, a mechanism for communication between them is required. There are two cooperation styles, one is message passing and another is method invoking. Message passing is a kind of coarse-grained cooperation, and agent behaviors are black boxes to other agents or agent systems. Agents only provide interfaces for receiving or sending messages. This is the mainstream of cooperation style for mobile agent systems. Method invoking is a kind of fine-grained cooperation similar to RPC mechanism, so that one task could be completed through invoking several methods from different agents. This style may break the encapsulation of agents, and it leads to security problem and system complexity. Since mobile agents are relatively independent programs for particular tasks, supporting RPC style method invoking is not to its advantage. If a task is to be completed using several methods from different agents, the better way is to build a new agent based on these methods.

The communication could be synchronous or asynchronous. Synchronous communication means the sender has to wait response from communicating party to continue its execution, and asynchronous communication means the sender continues its execution immediately after sending out messages. The communication in mobile agent systems can be implemented as direct communication or indirect communication. Direct communication relies on message passing. A special case is a rendezvous model where two agents can communicate only when they reside within the same place, which overcomes the requirement for locating other agents on the network. If two agents need to communicate, they must move to the same host or mobile agent system. Indirect communication implies that agent interact via blackboard located in each hosting environment, which are used as information spaces to store and receive message locally. In addition, it needs a special directory service at some hosts to locate the receiver agents. So direct communication only happen between agents that reside on the same host, and indirect communication happens between agents who reside in different hosts and the communication is via some agent systems across the network.

1.1.4 Mobility

Mobility is the unique characteristic of mobile agent systems. Supporting agent mobility is a fundamental requirement of the agent infrastructure. An agent can request its current support system to transport it to some remote destinations. The agent system must then deactivate the agent, capture its state, and transmit it to the agent system at the remote destination host. The destination agent system then restores the agent state and reactives it, thus completing the migration. According to how to recover the agent states coming from its source host, there are two types of mobility: weak mobility and strong mobility. Weak mobility permits the migration of both code and part of the execution state. After migration, the execution starts from the beginning or from a specific point. By strong mobility, both the code and the whole execution state are moved in order to restart the execution exactly from the point where it was stopped before migration.

From logical agent mobility point of view, agent mobility can be classifed as two types: 1. Remote agent creation. A client program interacts with the destination agent system with necessary information to create an agent in the remote host and to resume the agent execution. 2. Agent transfer. If an agent needs to transfer to other agent systems, its current agent system creates a travel request. As part of the travel request, the agent provides naming and addressing information that identifies the destination host. If the source agent system reaches the destination agent system, the destination agent system must either fullfill the travel request, or return a failure indication to the agent. If the source agent system cannnot reach the destination agent system, then a failure indication must be returned to the source agent system. When the destination agent system agrees to the transfer, the source agent's state, authority, security credentials, and, if necessary, its codes are tranferred to the destination agent system. The destination agent system then reactivates the source agent, and then execution is resumed. There are three implementation possibilites for agent mobility. The first one is that the agent carries all codes as it migrates. This allows the agent to run on any host which can execute the code. The second one is that the agent does not carry any code, but will recontruct its program using code from destination hosts. This reduces the traffic, time and improves security, but it lacks flexibility and agent functions are limited by available components in destination hosts. The third one is that the agent contains only reference to its code base, so code will be provided from a code base server upon the agent request, and this is also called code-on-demand.

1.1.5 Security

Security is the most concerned issue to mobile agent systems since the mobile agents may come from unknown hosts and have bad intentions. On the other side, agents may lose their activity confidence or carry altered results because of bad actions from other agents or agent systems. The attack to agent systems includes pilfering of sensitive information, damage to host resources, denial of service to other agents, and annoyance attacks. The attack to agents includes destroying the agent, stealing or modifying data that the agent carries, changing agents' codes or itineraries to have them perform malicious behaviors.

In order to ensure agents and agent system behavior responsibly, there are some requirements for security mechanisms: 1. Protection of privacy and integrity of agents. The system must provide mechanisms for secure communications, and secure transfer of agent code and states as it migrates across networks. Tampering of agents should be detectable. 2. Authentication of entities in the system. The entities participating in a mobile agent application, such as servers and agents, must be unambiguously identified. 3. Authorization and access control. Agent systems must be provided with a mechanism for protecting their resources, by specifying their access control policies and enforcing them. These policies includes such as restricting or granting agent capabilities, setting agent resource consumption limits, and restricting or granting access. So agents or agent systems only have limited capabilities for some operations such as agent creation or migration. The policies may limit agents to consume resource such as CPU, and memory in order to protect resources from abusive behaviors. Agent systems should control access to some resources or destinations, operations that an agent can invoke, and data that an agent can view, alter or provide.

1.2 A Logic Mobile Agent System

In this section, we define a logic mobile agent system as the reference model, which follows the MASIF standard. It includes four levels: communication, distributed system, mobile agent systems and mobile agent. The lowest level is operating systems that support TCP/IP protocol for communication. Then the CORBA level provides basic distributed system functions. Each mobile agent system is installed in each node, which accepts visiting mobile agents. The top most level is the level for mobile agents, which migrates from hosts to hosts for completing some tasks representing their users in the network.

1.2.1 Mobile Agent

Each mobile agent has a unique identity in the network. The identity consists of the name of the agent system that creates the agent, and one unique integer, which is assigned to it by agent systems. The name of each agent system name consists of its address and one unique identity within that host. Therefore, each agent system or mobile agent has one unique identity that identifies itself in the network. Each agent is associated with an authority that comes from its client. Each agent has an itinerary that is an ordered set of locations of agent systems, but that agent could update it on the trip. We do not consider code-on-demand style, so each agent always carries its code with its state during migration. Agents cannot clone themselves since only agent systems can create agents. Nevertheless, agent systems can create agents with the same functions but with different identities.

1.2.2 Agent System

An agent system can create, interpret, execute, transfer and terminate agents. Each agent system has an authority that identifies the person or organization. Each agent system has a unique name consisted of its name and address. Each host only has one agent system, and it may provide a blackboard for communication. Some agent systems may have more powers to act as coordinators for a group of agent systems. The coordinator has a directory service to help locate agents. Each agent system only can create mobile agent in its own host, so it does not support remote agent creation. Agent systems support strong mobility, so agents carry their states to other hosts and resume from the exact point when they move out from the source agent systems. Agent systems can force guest agents to move out, and they can destroy their own agents even though they might be in remote hosts.

1.2.3 Communication

Mobile agent systems support synchronous and asynchronous communication. In this reference model, we only consider message passing, which is of asynchronous communication. If two agents are within the same agent system, they can communicate with each other directly. However, if they reside at different hosts, they may move to the same host for communication, or they must communicate through other agent systems, which provide blackboards to save messages and directory facility to locate other agents. After an agent sends out a message to another agent, it should not move out from this host before it receives response or until timeout if it needs interaction from its communication partner. Before an agent moves to another host, it must register its next destination in a directory service. Each directory keeps all active agent paths, and all directories are synchronized so that each agent only needs to register on its nearest directory. However, if this system is in an open network such as the Internet, this method will have very inefficient. For that matter, we have to limit the communication so that the communication only happens within one region or on the same host.

1.2.4 Mobility

An agent can request its current agent system to transport it to some remote destinations. Agent systems also can force their guest agents move out. The agent system must then deactivate the agent, capture its state, and transmit the state with code to the server at the remote host. The destination agent system then restores the agent state and reactives it, thus completing the migration. The agent system supports strong mobility, which means both the code and the whole execution state are moved in order to restart the execution exactly from the point where it is stopped before migration.

2 Modeling Mobile Agent Systems

Mobile agents bring a wide range of new distributed applications. In order to deeply research earnest issues such as security, mobility and cooperation, it is necessary to introduce formal methods to provide a mathematical framework useful for specifying and verifying these applications [SM98].

There are a variety of formalisms for mobile agent systems, and they have different levels of expressiveness that may be used to formalize mobility, which is the most important property of mobile agent systems. In this chapter, we mainly describe formalisms based on Petri nets since we chose one high-level Petri nets (PrT nets) to model system behaviors in our work. However, in order to improve our understanding on the theories of mobile agents and compare our works with others, we also describe formalisms based on process algebra and other formalisms such as mobile UNITY and PoliS. We are especially interested in the communication mechanisms such as channels used in these formalisms, because modeling the communication between concurrent components is so important but difficult. Moreover, we chose channels, which are similar to the channels in CSP [Hoa85] and in π-Calculus [Mil99], to facilitate the communication between agents and systems in our models.

2.1 Petri Nets

Petri nets are a popular formalism with graphical and mathematical notations, which are effective to specify system behaviors and analyze concurrent and parallel systems. Agent mobility can be naturally simulated by transition firing of Petri nets. Nevertheless, it might be complex and even difficult to represent agent migration and dynamic connection. Tokens in Petri nets, even in self-modifying nets and reconfigurable nets are passive, whereas agents are active. To bridge the gap between tokens and agents, some multiple-level approaches were provided. We introduce two formalisms here, one is based on PrT nets, and another one is based on colored

Petri nets. However, all of them chose a multi-level paradigm from the elementary object system (EOS) [Val98], which allows some nets wrapped as tokens in other nets.

2.1.1 PrT Nets

In our previous work [XYD03], we defined a formal architecture for logical agent mobility using PrT nets. It is a two-layer PrT net model consisting of system nets and agent nets and connector nets, to model the behaviors of the environments, mobile agents, and connectors, respectively. The system nets define environments or platforms, and the agent nets define agents. Communications between systems or agents are defined as connector nets. Furthermore, agent nets are wrapped as tokens in system nets, and these agents only can update their states in a particular place of each system net. The connectors include external connectors and internal connectors. External connectors connect components, and internal connectors connect agents to their environments. The internal connectors are dynamically configured so that a changing number of agents in each component can be connected to their environment. There is at least one external connector for each mobile agent system, and each component has exactly one internal connector. In the following section, we introduce this two-layer PrT net approach from an architectural model – the LAM (Logic Agent Mobility) model.

LAM model: The LAM model specifies a mobile agent system as a set of components and connectors. Different components identify different locations for mobile agents. The connectors specify the interactions among components. Each component is made up of an environmental part and an internal connector, both defined as PrT nets. The environment part of components provides facilities for agent mobility, and the internal connector of components is responsible for the dynamic connection of the environmental part with a changing number of mobile agents. An agent can migrate from one component to another by transition firing at run time because the whole agent net is used as part of a structured token in the PrT nets modeling components and connectors. Therefore, the migration results in the change of agent locations. When an agent is being transferred, no transition in the agent is enabled.

Agent model: Each agent is defined as a PrT net, called agent net. The interface, behavior, and state of an agent are modeled by some input/output predicates for incoming/outgoing messages, the transitions, and the predicates of the agent net, respectively. Particularly, the state of the agent is the marking of the agent net.

System model: Each environment is modeled as a PrT net, called system net, and each component includes a system net and an internal connector net. A system net and its connector net forms a whole net. Each system net has external input/output interfaces connecting to external connectors, which transfer messages or agents. In addition, each system net has internal input/output interfaces that connect to internal connectors, which transfer messages between agents and the system. Since agent nets and their states can be packed up as part of tokens in the system nets, agent transfer is naturally simulated by the transition firing of PrT nets: if a transition is activated, an agent, used as part of a token, moves from an input predicate to an output predicate of the transition. After a certain sequence of transition firing, the agent is moved from one component to another through connectors.

Internal connectors: In order to capture the social ability of agents and to bridge the gap between agents and first-class components, agents need to dynamically connect with their environments. A single internal connector is used to connect an environment with all mobile agents residing in the current component. Such internal connector depends on the internal interface of environments, the running agents, and agent interfaces. The basic structure of an internal connector includes two parts: one part receives messages from the system net, and another part sends messages from agents to the system. The first part receives messages from the internal output interface of the system, and then delivers the messages to the input interface of an agent according to the message destination address. The second part sends messages from agents residing in the system to the internal input interface of the system. If there are several agents in the system, the sending messages are synchronized using a synchronization predicate.

External connector: A group of components is connected via external connectors, and arcs of connector nets are supposed to be properly labeled so that a migrating agent is always transferred to a proper destination. Each external output place of system nets may connect to all other components. The structure of external connector is simple: each system external output interface connects to external input interfaces of all other connected systems through transitions.

2.1.2 Reference Nets

Reference nets are a type of high-level Petri nets derived from colored Petri nets, which are especially well suited for the description and execution of complex, concurrent processes [Kum98]. Reference nets are similar to colored Petri nets except with four conceptual extensions: net instances, nets as token objects, communication via synchronous channels, and several different arc types. In the following section, we introduce the syntax of reference nets, and reference net model of mobile agent systems.

Nets as tokens: Reference nets implement the "nets within nets" paradigm of elementary object nets (EOS). In some nets, the structures of tokens are other nets.

Net instances: Net instances are similar to objects in object oriented programming languages. If tokens in some nets are nets, these tokens are instantiated copies of their template nets. Different instances of the same net can take different states at the same time and are independent from each other in all respects. In reference nets, a *new* operation, which is associated with a transition, creates an instance of a template net when the transition fires. If two tokens represent the same instance of a template net, the two tokens share the same state at any time. This is same as the "*call by reference*" in programming language. This is the reason to call this formalism as reference nets.

Synchronous channels: The idea behind introducing synchronous channels into Petri nets can be found in [CH94], which introduced channels to colored Petri nets. Reference nets implement this idea to synchronize and communicate between different transitions. The synchronous channels in reference nets are not symmetric but directed, which means only one of the two synchronized transitions indicates the net instance in which the counterpart of the channels is located. The information transferred between two transitions through a synchronous channel can be bi-directional and it is possible to transfer information within one net instance. The invoking side of channels is called downlink, and the invoked side is called uplink [Kum98]. To fire a transition that has a downlink, the reference net instance must provide an uplink with the same name and parameter count, and it must be possible to bind the variables suitably so that the channel expressions evaluate to the same values on both sides. The transitions can then fire simultaneously. A transition may have several downlinks, but it only has at most one uplink.

Extended arc types: Reference nets have three special arc types: *reservation arcs*, *test arcs* and *inhibitor arcs*. A reservation arc has an arrow at both ends and is solely for one occurrence of a transition. It is a short hand notation for two opposite arcs with the same inscription connecting a place and a transition. A test arc does not consume any token but is used for testing the existence of a token in a given place (and the same token can be tested simultaneously by more than one arc). An inhibitor arc prevents the occurrence of transitions as long as the connected place is marked. Here is an example of reference nets:

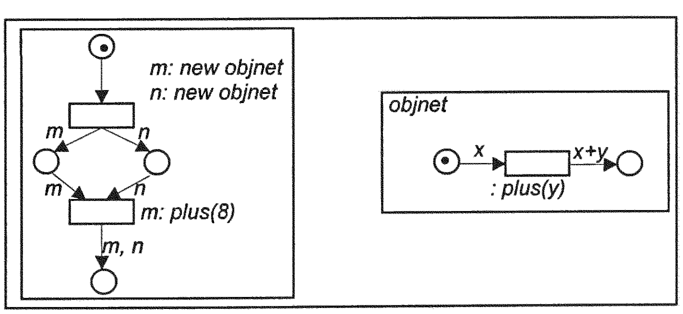

Figure 2.1 An example of reference nets

In the left diagram of Figure 2.1, the operation *new* creates two instances *m, n* from template *objnet*. The synchronous channel *plus* sends value 8 to the instance net *m* for a calculation. The results in the bottom place have two instances *m, n* but with different states.

MULAN [KR01] is a mobile agent system defined using reference nets. We introduce MULAN system below:

System architecture: In MULAN, a multi agent system consists of many agent platforms connected via a network. Therefore, the top model is some places connected with transitions. The places represent locations of agent platforms, and the tokens in these places are agent platforms, which are defined as agent platform nets. The transitions describe communication or mobility channels, which build up the communication infrastructure.

Agent platform: In each agent platform, there is a place to accommodate agents. The communication between agents is through internal or external communication. Two agents communicate through internal communication if these two agents are within the same platform. The internal communication binds two agents: the sender and the receiver, to pass one message over a synchronous channel. The communication between two agents from different platforms is via external communication, which only binds one agent in the platform since another one is in another platform. The transitions *new* and *destroy* are used to create agents and to kill agents. The transitions *receiver agent* and *sender agent* are used to receive agents from other platforms and to send agents to other platforms.

Mobile agents: Agents are modeled in terms of nets. The agent net is modeled as receiving, processing and sending out messages. Agents act reactive or proactive, so an agent net has two transitions to model these two behaviors. It may have a knowledge base to provide intelligence for agents. In an agent net, there is a place to model the protocol for agent communication. Therefore, tokens in a particular place of an agent net are wrapped from protocol nets. In summary, in this modeling system, the tokens could be agent platform nets, agent nets, protocol nets, or other regular types.

Mobility: The mobility of agents is modeled as token migration from one system net to another. Agent tokens in one platform net are sent to another when *send agent* transition fires, and the receiver platform net gets agents when the *receive agent* transition fires. The *send agent* transition and *receive agent* transition are synchronized using a synchronous channel. In the paper [KMR03], Kohler *et al.* defined four types of mobility that are supported by reference nets. These types of mobility are differentiated by the interaction between object nets (agent nets) and system nets (platform nets):

1. Spontaneous move: The object net moves inside the system net, neither object nor system net controls the move.
2. Subjective move: Only the object net triggers the movement.
3. Transportation or Objective move: The system net forces the object net to move.
4. Consensual move: Both the system net and the object agree to move the object net.

2.2 Other Formalisms

There are some other formalisms to model mobile agent systems. Examples are π-Calculus, mobile Petri nets, mobile UNITY, and Polis. Each of these formalisms defines a mathematical framework that can be used to model and analyze code mobility. They vary greatly in their expressiveness, in the mechanisms they provide to specify mobile code based applications, and in their practical usefulness for the validation and the verification of such applications [SM98].

2.2.1 π-Calculus

π-Calculus is a process algebra to model the changing connectivity of interactive systems [Mil99]. "The π-Calculus is a way of describing and analyzing systems consisting of agents which interacts among each other, and whose configuration or neighborhood is continually changing" [Mil93]. The most important concept in π-Calculus is channels, which provide the communication mechanism between processes and define the configuration of systems. The basic entity is channel names with which the complex entities called processes are built. π-Calculus has several versions depending on the content transferred in channels. If processes only can send channel names in channels, this π-Calculus is called monadic π-Calculus. If processes can send tuples of channel names in channels, this π-Calculus is called polyadic π-Calculus. Moreover, if processes can send tuples of processes and channel names in channels, this π-Calculus is called higher-order π-Calculus.

A monadic π-Calculus process is given by the following syntax [SM98]:

$$P ::= \sum_{i \in I} \alpha_i . P_i \quad | \ P_1 | P_2 \ | \ P_1 + P_2 \ | \ \nu x P \ | \ !P$$
$$\alpha ::= x(y) \mid \overline{x}y \qquad (2.2.1)$$

Where I is any finite indexing set, x, y are channel names, | is parallel operator, + is the sum operator, νxP is the restriction operator, which bounds the name x within P, and $!$ is the replication operator. $x(y)$ means name y is received over channel x, and $\overline{x}y$ means the name y is send over channel x. Since π-Calculus allows processes to pass channel names as parameters over channels, if a process moves, its neighborhood changes and the process changes its channels for communication. If x, y and z are channel names, transition $x(y).P \xrightarrow{x<z>} P\{z/y\}$ means channel name z is sent along channel x, then the resulting process $P\{z/y\}$ is able to use z as a channel for future communication. The values of channel names may be assigned to processes at run time.

The polyadic π-Calculus extends monadic π-Calculus by allowing tuples of names as well as sorts, data structures and functions to be transferred over channels, whereas monadic π-Calculus only transfers channel names over channels. The higher-order π-Calculus extends polyadic π-

Calculus by allowing functions of arbitrary order to be transferred. It allows processes to be transferred over channels. After a process has been transferred, it can begin its execution.

The idea to use π-Calculus to model mobile agent systems is straightforward. The agent support systems or platforms can be modeled as processes with channels to receive and send messages or agents. Each agent is a process with channels to receive and send messages from other agents or agent systems. When an agent system receives an agent from its channel, it behaves in parallel with the agent process. The channel values of agents are assigned at run time, and they may change when agents move from one place to another. The communication and interaction between processes are through these channels.

2.2.2 Mobile Petri nets

Mobile Petri nets are a variation of colored Petri nets. In mobile Petri nets, the colored tokens are tuples of place names, and an input token of a transition can be used in its post-set to specify a destination. The postsets of its transitions are not static, but dynamically change depending on the colors of the tokens the transitions consume. For instance, considering a print-spooler example [AB96], we have a transition of the following definition:

$$ready(PRINTER, TYPE), job(FILE, TYPE) \rhd PRINTER(FILE) \qquad (2.2.2)$$

The left side of the symbol ⊳ is the pre-condition of the transition, and the right side is the post-condition. The preset of the transition has two places: *ready* and *job*; the post-set of the transition has one place *PRINTER*, whose value is not decided until the transition fires. *PRINTER*, *TYPE* and *FILE* are variables, and their values are instantiated at the time when the transition fires. When the transition fires, it generates a new place with value of *PRINTER*, and it has token *FILE*, which is moved from *job* place to *PRINTER* place.

Dynamic Petri nets are an extension of mobile Petri nets by adding the possibility of creating new nets during the firing of a transition. When a transition fires, the transition might generate a new subnet instead of producing only new tokens. So the current state of the net is not represented any more by a marking, but by a net.

A mobile agent system can be defined as a dynamic Petri net, where some tokens are referred to as agents, which are defined as specifications to generate subnets in post-sets of some transitions. The agent mobility is modeled as firing of transitions where agent nets being added to the model dynamically. The migration of agents is to move tokens from transition preset to their post-set, which includes subnets representing those agents.

2.2.3 Mobile UNITY

Mobile UNITY is an extension from the UNITY methodology [CM88], which is a state-based formalism with the foundation in temporal logic, to model dynamically reconfiguring distributed systems such as mobile agent systems. It extends the UNITY notation to express the computation taking place within the mobile components of a system, and extends the UNITY proof logic to reason about mobile computation. UNITY programs have notations similar to Pascal program style. A UNITY program is a set of assignment statements that execute atomically and are selected for execution with weak fairness, which means each statement is scheduled to execute infinitely often in an infinite computation. A UNITY program includes variable declaration, initialization, and assignments. The semantics of UNITY are given in terms of program properties that can be proven from the text.

UNITY is not adequate to model the mobile computing domain since it describes systems as static collections of components with fixed patterns of connectivity. In Mobile UNITY, each program is a unit of mobility, which has a distinguished location attributes to capture the program movement. The changes of location value reflect the movement of program units. Mobile UNITY adds two new constructs, *transient variable sharing* and *transient action synchronization*, to model communication between mobile units [RM97]. Transient variable sharing allows mobile programs to share data transparently with different programs at different times depending upon their relative locations. Transient action synchronization means a statement owned by one program is executed in parallel with statements owned by other programs when certain spatial conditions are met.

We can model a mobile agent system using Mobile UNITY and verify the system properties using its proof logic. Each agent or agent system is defined as a program, and the agent program has a location attribute, whose value is updated when the agent moves from one place to another. The communications between agents or systems are through transient variable sharing and/or transient action synchronization. The dynamic agent migration property is naturally captured with the changes of location values at run time.

2.2.4 PoliS

PoliS is a coordination language, which focuses on coordination problems in a multi-process system [CFM00] [Mas99] [SM98]. A PoliS specification consists of a collection of tuple spaces, or spaces for short. It has modular and hierarchical structure with a tree of nested spaces that dynamically evolve over time [CFM00]. A space can contain other spaces, which have two types: *ordinary tuples* and *program tuples*. Ordinary tuples are ordered sequences of values and types. Program tuples contain coordination rules that manage activities inside the space they belong to. A program tuple has an identifier and rule codes, which define reaction rules. The execution of a program tuple is an action that can modify a space tree by removing and adding tuples. However, an action can only process the tuples of the space it belongs to or its parent space. The basic communication mechanisms of PoliS are through shared memory and are asynchronous and anonymous. Tuples representing messages are put in the environment by program tuples that have to communicate, and program tuples access messages by pattern matching. Data mobility in PoliS is denoted by rules that are able to consume tuples locally and to produce tuples outside the local space. Code mobility is denoted by rules that are able to consume and produce tuples containing codes.

Model checking technique is used to analyze PoliS specifications. PoliS is the first formalism to build an automatic framework to analyze properties on specifications of systems with code mobility [CFM00]. The model checker in PoliS exploits its modularity features to reduce the space of graphs built for a specification. The algorithm used for verification of properties follows the one presented in [CES86]. The logic is based on temporal logic CTL (Computation Tree Logic) with extension related to the spaces-based coordination model.

An mobile agent systems can be modeled as tuple spaces, which includes program tuples representing mobile agents. The agent mobility can be realized with removing or inserting program tuples from or to particular spaces.

2.3 Discussion

We presented several formalisms that have distinguished flavors and offer different views on mobile agent systems. The two-layer PrT models and reference nets all implement "nets within nets" paradigm, which naturally captures the physical architecture of mobile agent systems. Their mobility mechanism is by reference passing, where some tokens denote other nets. The two-layer PrT net introduces connectors to facilitate the communications between different nets. It keeps the basic semantics of PrT nets in each net, so that its models are clear and it does not add more complexity to analyze models since its analysis rules follow ordinary PrT nets. However, the connectors themselves were defined statically, so they cannot properly solve issues on dynamical configuration and communication between nets on different level. Reference nets provide synchronous channels for transition communication and synchronization. However, the channels are pre-defined so that it is still difficult to model the dynamic configuration of the software architecture of mobile agent systems. Because of the extensions on Colored Petri nets, the semantics of reference nets are different from the basic semantics of colored Petri nets, and there is not a formal definition of reference nets so far. This brings the complexity to formally analyze its models. The π-calculus is the first language offering features to specify movement across channels, but it does not support the location property, which is very important for mobile agent systems. It focuses on the notation for processes, but it does not provide notations for environments of the computation. Mobile Petri nets express process mobility by using variables and colored tokens in an otherwise static net. Moreover, dynamic Petri nets extend mobile Petri nets with mechanisms for modifying the structure of Petri nets. However, they bring forth the complexity of modeling and analysis. These two nets cannot naturally model the movement of one subnet from the preset to the post-set of a transition when it fires. In addition, it is difficult to model a system with dynamic configuration graphically since we cannot draw the post-set of some transitions, and it is much more difficult to define the graphs following these transitions. Mobile UNITY is a state based formalism used for specification of physical and logic mobility. Each mobile process has a special location attribute, and the migration is reflected as changes of location values. PoliS is a coordination model with hierarchical tuple-spaces and multi-set rewriting. Nested spaces that represent software components can move and change their positions in the tree. The communication is specified using the asynchronous mechanism through shared memory.

An important aspect of these formalisms is whether they can provide a means for the verification of properties. The two-layer PrT models and reference net models can be unfolded into ordinary PrT nets models or colored Petri nets models, respectively, so both models can be verified using analysis methods from ordinary high-level Petri nets. The π-calculus is modeled in terms of history dependent automata that lead to automatic verification procedure. Mobile UNITY provides a temporal logic to prove program properties. PoliS analyzes its specifications with a model checker that tests formal properties of the system specified.

3 Model Checking Concurrent Systems

Software architecture has been identified as a promising approach to bridge the gap between requirements and implementations in the development of complex systems [MKG97]. In order to support defining software architecture, it emphasizes a separation of concerns: an architecture description language for describing component structures and component functionalities, and a formal analysis method. A sound architecture has profound impact on the maintainability, scalability, and extensibility of software's lifecycle. A rigorous approach toward architectural level system design can help to detect and eliminate design errors as early as possible in the development cycle, to avoid costly fixes at the testing stage, and thus to reduce overall development cost and to increase the quality of the systems. To achieve the above advantages, a more formal and rigorous way to software architectural specification, design and analysis is required [HYS03]. There are many ADLs, but research on software architecture development and analysis is not enough [Sha01]. Theorem proving, testing, model checking and simulation are most popular approaches to analyze software architecture. Theorem proving needs user interaction during proving and the tedious work make it unsuitable for complex systems. Testing for software architecture needs a complex support environment, but it cannot guarantee the system correctness. Simulation suffers the same problems as testing. However, model checking is a powerful technology for analyzing software architecture and the verification is completely automatic.

In this dissertation, we use an extended PrT nets to define system behavior models, LTL to define system properties, and the model checking tool SPIN to verify the properties. In the following part, we describe Petri nets, PrT nets, LTL and model checking including model checking tool SPIN.

3.1 Petri Nets

Petri nets are a graphical and mathematical modeling tool applicable to many systems, and they are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic. In the following sections, first we introduce the low-level Petri nets (or Petri nets) and its properties, and then we define a simplified PrT nets.

3.1.1 Definition of Petri Nets

Formally, a Petri net can be defined as follows [Wan98]:

Definition 3.1.1 (Petri nets): A Petri net is a 5-tuple $N = (P, T, I, O, M_0)$, where: $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places*; $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions*, $P \cup T \neq \varnothing$, and $P \cap T = \varnothing$; I: $(P \times T) \rightarrow N$ is an *input function* that defines directed arcs from places to transitions, where N is a set of nonnegative integers; O: $(P \times T) \rightarrow N$ is an *output function* that defines directed arcs from transitions to places; and M_0: $P \rightarrow N$ is the *initial marking*.

A *marking* is an assignment of *tokens* to the places of a Petri net. A token is a primitive concept for Petri nets (like places and transitions). Tokens are assigned to, and can be thought to reside in, the places of a Petri net. The number and position of tokens may change during the execution of a Petri net. The tokens are used to define the execution of a Petri net.

A Petri net graph is a Petri net structure as a bipartite directed multi-graph. A *circle* represents a place; a *bar* or a *box* represents a transition. Directed arcs (arrows) connect the places and the transitions, with some arcs directed from the places to transitions and other arcs directed from transitions to places. An arc directed from a place p_j to a transition t_i defines p_j to be an *input place* of t_i, denoted by $I(t_i, p_j) = 1$. An arc directed from a transition t_i to a place p_j defines p_j to be an *output place* of t_i, denoted by $O(t_i, p_j) = 1$. If $I(t_i, p_j) = k$ (or $O(t_i, p_j) = k$), then there exist k directed (parallel) arcs connecting place p_j to transition t_i (or connecting transition t_i to place p_j). A circle contains a *dot* representing a place containing a token.

3.1.2 Transition Firing

The execution of a Petri net is controlled by the number and distribution of tokens in the Petri net. A Petri net executes by firing transitions. We now introduce the enabling rule and firing rule of a transition, which governs the flow of tokens:

Enabling rule: A transition t is said to be *enabled* if each input place p of t contains at least the number of tokens equal to the weight of the directed arc connecting p to t, i.e., $M(p) \geq I(t, p)$ for any p in P.

Firing rule: An enabled transition t may or may not fire depending on the additional interpretation; firing of an enabled transition t removes from each input place p the number of tokens equals to the weight of the directed arc connecting p to t. It also deposits in each output place p the number of tokens equals to the weight of the directed arc connecting t to p.

3.1.3 Properties of Petri Nets

As a mathematical tool, Petri nets have a number of properties. Here we provide an overview of, from the practical point of view, some of the most important behavioral properties. They are reachability, boundedness, conservativeness, and liveness [Wan98].

Reachability: The set of all possible markings reachable from a given initial marking is called *reachable set*, and denoted by $R(M_0)$. The set of all possible firing sequences from M_0 is denoted by $L(M_0)$, and let $\sigma \in L(M_0)$, then the reachable state of M_0 is denote by $M_0\ [\sigma > M_i$.

Definition 3.1.2 (Reachability): For a given Petri net $N = (P, T, I, O, M_0)$, if there is a $\sigma \in L(M_0)$ such that $M_0\ [\sigma > M_i$, then M_i is said to be reachable from M_0.

Boundedness and Safeness: The Petri net property that helps to identify the existence of overflows in the modeled system is the concept of *boundedness*.

Definition 3.1.3 (Boundedness): A place p is said to be k-bounded if the number of tokens in p is always less or equal to k (k is a nonnegative integer number) for every marking M *reachable* from the initial marking M_0, *i.e.*, $M \in R(M_0)$. It is safe if it is 1-bounded.

Definition 3.1.4 (k-bounded): A Petri net $N = (P, T, I, O, M_0)$ is k-bounded (safe) if each place in P is k-bounded (safe).

Conservativeness: It indicates that there is exactly the same number of tokens all places in every reachable marking of a Petri net. Here is a broader definition of conservation:

Definition 3.1.5 (Conservativeness): A Petri net $N = (P, T, I, O, M_0)$ is said to be conservative if there exists a vector $w = (w_1, w_2, \ldots, w_m)$ where m is the number of places, and $w_i > 0$ for each $p_i \in P$, such that

$$\sum_{i=1}^{m} w_i M(p_i) = const. \tag{3.1.1}$$

Liveness: A Petri net modeling a deadlock-free system must be *live*. This implies that for any reachable marking *M*, it is ultimately possible to fire any transition in the net by progressing through some firing sequence. Different levels of liveness for transition *t*, and marking M_0, were introduced in [Com72] [Lau75].

Definition 3.1.6 (Liveness): A transition *t* in a Petri net $N = (P, T, I, O, M_0)$ is said to be:

1. *L*0-live (or dead) if there is no firing sequence in $L(M_0)$ in which *t* can fire.
2. *L*1-live (potentially fireable) if *t* can be fired at least once in some firing sequence in $L(M_0)$.
3. *L*2-live if *t* can be fired at least *k* times in some firing sequence in $L(M_0)$ given any positive integer *k*.
4. *L*3-live if *t* can be fired infinitely often in some firing sequence in $L(M_0)$.
5. *L*4-live (or live) if *t* is *L*1-live (potentially fireable) in every marking in $L(M_0)$.

Definition 3.1.7 (*Lk*-live): A Petri net $N = (P, T, I, O, M_0)$ is said to be *Lk*-live, for marking M_0, if every transition in the net is *Lk*-live, $k = 0, 1, 2, 3, 4$.

3.2 PrT Nets

Here we give the definition of PrT nets, which is different from ordinary PrT nets in [Gen87] [GL81]. We define PrT nets same as the PrT nets in [XYD03].

Definition 3.2.1 (PrT net): A PrT net is a tuple $(P, T, F, \Sigma, L, \varphi, M_0)$, where:

1. P is a finite set of predicates (first order places), T is a finite set of transitions $(P \cap T = \emptyset$, $P \cup T \neq \emptyset)$, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation, or simply a set of arcs. (P, T, F) forms a directed net.
2. Σ is a structure consisting of some sorts of individuals (constants) together with some operations and relations.
3. L is a labeling function on arcs. Given an arc $f \in F$, the labeling of f, $L(f)$, is a set of labels, which are tuples of individuals and variables. The tuples in $L(f)$ have the same length, representing the arity of the predicate connected to the arc. The zero tuple indicating a no-argument predicate (an ordinary place in Petri nets) is denoted by the special symbol $<\not{c}>$.
4. φ is a mapping from a set of inscription formulae to transitions. The inscription on transition $t \in T$, $\varphi(t)$, is a logical formula built from variables and the individuals, operations, and relations in structure Σ.
5. M_0 is the initial or current marking.

$$M_0 = \bigcup_{p \in P} M_0(p) \qquad (3.2.1)$$

where $M_0(p)$ is the set of tokens residing in predicate p. Each token is a tuple of symbolic individuals or structured terms constructed from individuals and operations in Σ.

The above definition describes the simplified general PrT nets [Gen87] [GL81] in two ways: 1. An arc labeling is a set of tuples (labels) $\{l_i\}$ rather than a formal sum $c_1l_1+c_2l_2+ \ldots +c_nl_n$. 2. Accordingly, the marking of a specific predicate under a certain state is a set of tokens instead of a formal sum of tokens. The simplification results in efficient analysis compliant with the concurrent semantics of Petri nets.

Let ${}^\bullet t = \{p \in P : (p,t) \in F\}$ and $t^\bullet = \{p \in P : (t,p) \in F\}$ be the pre-condition predicates and the post-condition predicates of transitions t, respectively. Let ${}^\bullet p = \{t \in T : (t,p) \in F\}$ and

$p^\bullet = \{t \in T : (p,t) \in F\}$ be the sets of input transitions and output transitions of predicate p, respectively. For a subset of predicates $Q \subseteq P, {}^\bullet Q = \bigcup_{p \in Q} {}^\bullet p$ and $Q^\bullet = \bigcup_{p \in Q} p^\bullet$. Basically, a transition t in a PrT net is enabled under marking M_0 if there is a substitution θ such that $l/\theta \in M_0(p)$ for any label $l \in L(p, t)$ for all $p \in {}^\bullet t$ and $\varphi(t)$ evaluates true with regard to θ, where l/θ yields a token by substituting all variables in label l with the corresponding bound values with regard to θ. The firing of an enabled transition t removes all tokens in $\{l/\theta: l \in L(t, p)\}$ from each input predicate $p \in {}^\bullet t$, and adds all tokens in $\{l/\theta: l \in L(t, p)\}$ to each output predicate $p \in {}^\bullet t$. After the firing of t, we get a new marking M'. Formally, $M_1(p) = M_0(p) - \{l/\theta: l \in L(t, p)\}$ for any $p \in {}^\bullet t$, and $M_1(p) = M_0(p) \cup \{l/\theta: l \in L(t, p)\}$ for any $p \in {}^\bullet t$. We denote a firing/occurrence sequence as

$$M_0[t_1\theta_1 > M_1[t_2\theta_2 > M_2 ... [t_n\theta_n > M_n \qquad (3.2.2)$$

or, simply, $t_1\theta_1 t_2\theta_2 ... t_n\theta_n$, where $t_i (1 \leq i \leq n)$ is a transition, $\theta_i (1 \leq i \leq n)$ is the substitution for firing t_i, and M_i $(1 \leq i \leq n)$ is the marking after t_i fires, respectively.

Considering the fact that a token in a PrT net may carry structured data and a PrT net is a structure, we will allow a PrT net to be packed up as a part of a token in another PrT net. Besides, additional constraints may be imposed on the enabling of transitions in a PrT net. In order to facilitate the communication and synchronization between nets, we extend PrT nets with channels.

3.3 Temporal Logic

Temporal Logic is a formalism for describing sequences of transitions between states in a reactive system. Properties like *eventually* or *never* are specified using special temporal operators. The formula ***Fq*** is true in the present if q is true at some moment in the future. Similarly ***Pq*** is *true* in the present if q is true at some moment in the past. The formula Gq is equivalent to $\neg \boldsymbol{P} \neg q$, meaning that q is true at every moment in the past. These operators can give surprisingly concise expressions of sentences with complex tense structures [Mcm93]. These operators can also be combined with boolean connectives and can be nested arbitrarily. There are a variety of temporal logics, for example, the branching time temporal logic and the linear time temporal logic. They mainly differ in the operators that they provide and the semantics of those operators [CGP99]. To be more concrete, the branching time logic and the linear time logic differ in how they handle branching in the underlying computation tree. In branching time temporal logic such as CTL, the temporal operators quantify over the paths that are possible from a given state. In the linear time temporal logic such as LTL, operators are provided for describing events along a single computation path [CGP99].

3.3.1 Linear Time Temporal Logic

LTL is a common way to specify properties of reactive systems. It has enough expressive power for most purposes and with relatively simple syntax and semantics. LTL is interpreted over infinite sequences of executions that make it appropriate to specify properties of the executions of Kripke structure.

Definition 3.3.1 (Kripke structure): A Kripke structure $K = (S, T, I, L)$ consists of:

1. A set of states S
2. A total transition relation $T \subseteq S \times S$
3. A non empty set of initial states $I \subseteq S$
4. A labeling of states with atoms $L: S \rightarrow 2^A$

LTL Syntax:

Xp: next time p holds (immediately after the current state p holds)

Gp: basic safety property (p holds globally after any number of steps p holds)

p **U** q: p holds until q holds (after a finite number of steps q holds and on the way to this point p continuously holds)

The set of atomic propositions is A, and LTL formulas are defined inductively as:

1. Every member $p \in A$ is a LTL formula
2. If p, q are LTL formula, then so $\neg p$, $p \vee q$, Xp, Fp, Gp, $p\ U\ q$ are LTL formula
3. There are no other LTL formula

Definition 3.1.2 (Path): Paths of a Kripke Structure:

1. A state s has a transition to a state t is defined as: $s \rightarrow t$ *iff* $(s, t) \in T$
2. A path π is an infinite sequence of states $\pi = (s_0, s_1, \ldots) \in s^{\omega}$ with $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$
3. $\pi(i) \equiv s^i$ and $\pi^i \equiv (s_i, s_{i+1}, \ldots)$

LTL semantics:

We recursively define f to be valid on path π, written $\pi \models f$ as follows:

- $\pi \models p$ *iff* $p \in L(\pi(0))$ (first state of π is labeled with p)
- $\pi \models \neg g$ *iff* $\pi \not\models g$ (g is not valid on π)
- $\pi \models g \wedge h$ *iff* $\pi \models g$ *and* $\pi \models h$ (g and h are both valid on π)
- $\pi \models g \vee h$ *iff* $\pi \models g$ *or* $\pi \models h$ (one of g or h is valid on π)
- $\pi \models g \rightarrow h$ *iff* $\pi \models g$ *then* $\pi \models h$ (if g is valid on π then h too)
- $\pi \models \mathbf{X}g$ *iff* $\pi^1 \models g$ (g is valid on π that first state chopped off)
- $\pi \models \mathbf{F}g$ *iff* $\exists i[\pi^i \models g]$ (g is valid on some suffix of π)
- $\pi \models \mathbf{G}g$ *iff* $\forall i[\pi^i \models g]$ (g is valid on every suffix of π)
- $\pi \models g\ \mathbf{U}\ h$ *iff* $\exists i[\pi^i \models h$ *and* $\forall j < i\ [\pi^j \models g]$ (g is valid until h holds)

3.4 Model Checking

Model checking is an automatic analysis technique for verifying finite state concurrent systems [CGP99]. The method has been used successfully in mission critical system development and to verify complex sequential circuit designs and communication protocols [GH02] [PMH02]. Moreover, it has become an important verification method in hardware development. In our previous work, we successfully found an error in a flexible manufacturing system (FMS) model [HDD02] using SMV. SPIN is a LTL model checking tool, and SMV is an example for CTL model checking [CGP99]. CTL model checker is mostly used in the development of the early tools for hardware verification, and LTL model checking technique is much more popular for software verification. In theory, CTL model checker is more efficient than LTL one, but in practice, there is no measure that can reliably tell which method can solve a given problem more efficiently since the LTL verification algorithm can more easily be implemented with an on-the-fly verification strategy to avoid constructing a whole system graph [Hol03]. Model checking technology suffers from the state-space explosion problem, because the systems are composed of many parallel processes; and in general, the size of the state space grows exponentially with the number of processes [GL94]. According to how to address this issue; we can distinguish model checking technology as symbolic verification and explicit verification. Symbolic verification such as SMV uses symbolic representations for sets of states and transition relations can check very large state space (10^{100} or more states) systems. Explicit verification model checker such as SPIN uses partial order to reduce state-space, and it is more powerful in software verification than symbolic model checking in this verification field [EP02].

3.4.1 Process of Model Checking

The following definition formally describes the model checking technique:

Definition 3.4.1 (Model Checking): Given a Kripke structure $M = (S, R, I, L)$ that represents a finite state concurrent system and a temporal logic formula f expressing some desired specifications, then to find the set of all states in S that satisfy f: $\{s \in S \mid M, s \models f\}$. S is a finite set of states, $R \subseteq S \times S$ is the transition relation, with $(s, t) \in R$ meaning that t is an immediate successor of s, $L : S \rightarrow 2^{AP}$, is the valuation of atomic propositions in each state, where AP is a finite set of atomic propositions. A non empty set of initial states $I \subseteq S$. In order to describe the model checking algorithm, the nodes represent the states in *S*, the arcs in the graph give the transition relation *R* and the labels associated with the nodes describe the function *L*.

Model checking consists of several tasks:

1. *Modeling,* The first task is to convert a design into a formalism accepted by the model checking tool, such as using Petri nets or Promela to define system models.

2. *Specification,* Specification is to state the properties that the design must satisfy. The specification is usually given in some logical formulas. It is common to use Temporal Logic, such as LTL or CTL.

3. *Verification,* The model checking algorithm evaluates a given specification formula by computing the set of states for which it is *true*. The formula and the set of states satisfying it are identical. Ideally, the verification should be completely automatic. However, in practice, it often involves human assistance. One such manual activity is the analysis of the verification results. In the case of a negative result, the user is provided with an error trace based on counter examples.

3.4.2 SPIN and PROMELA

SPIN is a generic automatic verification tool to formally analyze the logical consistency of distributed systems, which are defined using Promela (PROcess MEta Language). SPIN has some important features [Hol03]:

1. SPIN is used for software verification, and it has been used to trace logical design errors in complex software systems. It reports deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

2. It uses on-the-fly technology to avoid constructing global state graph, exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques to optimize the verification run.

3. SPIN is a full LTL model checking system. It defines systems correctness properties using LTL formulas, and these properties can also be specified as system or process invariants.

4. SPIN has three basic functions: 1. As an exhaustive state space analyzer for rigorously proving the validity of user-specified correctness requirements. 2. As a system simulator for rapid prototyping. 3. As a bit-state space analyzer that can validate large protocol systems with maximal coverage of the state space.

Promela is a verification modeling language with C programming language style. It provides a way for making abstractions of distributed systems that suppress details that are unrelated to process interaction. A Promela program consists of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which processes run. Here is a Promela program example for two processes mutually exclusively access critical section [Hol03]:

```
#define true            1
#define false           0
#define Aturn           1
#define Bturn     0

bool x, y, t;

proctype A()
{
      x = true;
      t = Bturn;
      (y == false || t == Aturn);
      /*Critical Section*/
      x = false
}
Proctype B()
{
      y = true;
      t = Aturn;
      (x == false || t == Bturn);
      /*Critical Section*/
      y = false
}
init {run A(); run B()}
```

Figure 2.2 A Promela program

In a Promela program, the process starts with *proctype* except the required *init* process, which serves as the program entry point and is used to initialize process instances. Processes can run concurrently and can be synchronized using global variables or channels. There are six data types: *bit, bool, byte, chan, short, int*. In addition, there are three constant types: *String constants, Enumeration constants* and *Integer constants*. The correctness claims have three styles: *assertion statement, label* and *never claim*. One Promela program may have more than one *assertion* or *label* statements in each process, but it only has one *never* statement. The *assertion* and *label* statements are used to check the model state properties (such as a specific statement can reach or not), and *never* statement can be used to check the mode execution properties (such as some property should hold at any time or any step) [Hol03]. In Promela, a *never* claim is essentially defined using LTL statements, but it is translated from LTL statements to Promela statements.

3.5 Analyzing Petri Nets/PrT nets

The verification of the correctness of Petri nets or PrT models can be done by demonstrating that a property specification S holds in a behavior model B, i.e. $B \models S$. [HDD00]

In the following sections, we will present general ideas of two approaches to fulfill $B \models S$:

1. The reachability graph technique, and
2. The model checking technique.

3.5.1 Analysis Using the Reachability Graph Technique

Let B be the behavioral model defined using CE nets, and S be the paired property specification in temporal logic. The idea of the analysis is to construct a reachability graph G from the behavioral model B, and then evaluate S in G. A reachability graph is the representation of all possible execution sequences of the net. Thus from $G \models S$ we get $B \models S$.

In reachability graph, each node is a marking of the net, and the directed arc from one node to another is the firing transition causing the change of the marking. A reachability graph is generated through the following steps:

Algorithm 3.5.1 (construct a reachability graph):

1. Choose the initial marking M_0 as the start node for the graph G

2. Fire each enabled transition from the initial marking M_0 one at a time

3. If the fired transition t_1 generates a new marking M_1, i.e. $M_0[t_1 > M_1$, the new marking is a new node of the graph G. Then connect M_0 to M_1 with a directed arc labeled with the fired transition t_1.

4. If the fired transition t_1 generates a marking M_1, which is an existing node of the graph G, then if there is not a directed arc from M_0 to M_1, connected M_0 to M_1 with a directed arc and labeled it with the fired transition t_1. If there is already an existed directed arc from M_0 to M_1, then choose next enabled transition t_2 to continue to generate new nodes and arcs.

5. The reachability graph is complete when no new node or no new arc can be generated.

As soon as the reachability graph G is generated from B, we can check the satisfiability of S along each path π in G starting from the node M_0. The checking can be done systematically and automatically by traversing the reachability graph G [HDW00].

For high-level Petri nets such as PrT nets, we can unfold their nets into low level Petri nets since high-level nets can be considered as structurally folded versions of low-level nets if the types of tokens are finite [Mur89]. Then we use the above algorithm to generate reachability graphs.

3.5.2 Analysis Using the Model Checking Technique

The basic idea of analysis using model checking technique is to transform a reachability graph G of net B into a state graph SG, and then go through SG to verify property specification S using model checking algorithms. It is straightforward to transform G into SG by replacing the node vectors with the set of atomic propositions true in the node. Thus, a reachability graph G is an intermediate representation of a finite state transition system M. Then we verify $B \models S$ through $SG \models S$.

Model checking considers all possible execution traces of a state transition model. Hence, it has to handle huge number of system states that may cause the state-space explosion problem.

There are different strategies to combat this problem. The main methods are symbolic method and explicit verification with partial order reduction. Symbolic method is extremely effective when it is used in domain of hardware verification, and partial order verification performs exceptionally well in domain of software verification [Hol03].

3.5.3 Analysis Using SMV

In order to verify Petri nets models, we need to know how to represent a CE net (1-safe Petri nets) using SMV input language. There are several possible ways to do so. The first method is based on SMV processes. For each transition, one process is instantiated. The places in a CE net are represented as boolean variables in the *main* module, and the *ASSIGN* declaration specifies their initial values. For each transition, a process is created. These processes are instances of parameterized modules that describe the behavior of different kinds of transitions. The *main* module defines a formula to verify deadlocks and contains the system specification [Wim97]. A solution to fairness is to add the declaration *FAIRNESS* running to every transition module. We propose to encode CE nets by specifying the transition relation directly using *TRANS* and *INIT*. The flexibility provided by these declarations made it possible to translate CE nets into an SMV specification that is easy to understand and reasonably efficient.

The SMV specification generated from a CE net consists only of a *main* module. In the *VAR* part, one Boolean variable P_i is declared for each place p_i. The *INIT* part specifies the initial state of the system, which contains a Boolean formula of a place variable only when it is marked in the initial marking. The *TRANS* part specifies the transition relation as a Boolean formula. It consists of one sub-formula *TRANS* t for each transition t. The formula is true if the transition is enabled and the next values of the place variables corresponding to the marking of the CE net after the transition has fired. The transition relation is the disjunction of all these sub-formulas, which ensures a valid successor is reached when one of the transitions is enabled. To ensure the generations of infinite paths required by CTL semantics, SMV eliminates all states with no successors from the model. To make it possible to verify CE nets containing deadlocks, a sub-formula specifying deadlock condition is added to the transition relation to allow the system to stay in its current state if a deadlock occurs.

The Petri nets model and SMV code for the Producer-consumer is shown in Figure 2.3 and Figure 2.4.

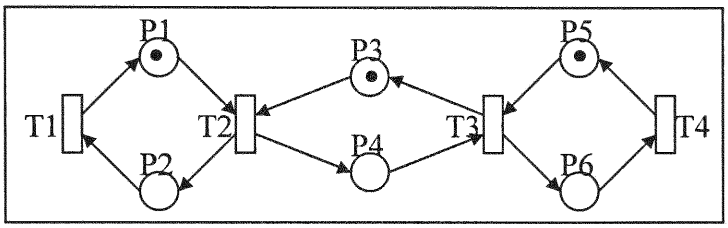

Figure 2.3 A Petri net model of a producer-consumer system

The following program is a SMV definition of the producer and consumer system given in Figure 2.3 with property specification ∀**G** (¬*T1* ∨ ∀ **F** *T4*)):

```
MODULE main
VAR
P1: boolean;
P2: boolean;
P3: boolean;
P4: boolean;
P5: boolean;
P5: boolean;
INIT
(P1=1&P2=0&P3=1&P4=0&P5=1&P6=0)
TRANS
(
--T1
T1.enabled&next(P1)=1&next(P2)=0&next(P3)=P3&next(P4)=P4&next(P5)=P5
&next(P6)=P6
|
--T2
T2.enabled&next(P2)=1&next(P4)=1&next(P1)=0&next(P3)=0&next(P5)=P5&n
ext(P6)=P6
|
--T3
T3.enabled&next(P3)=1&next(P6)=1&next(P4)=0&next(P5)=0&next(P1)=P1&n
ext(P2)=P2
|
--T4
T4.enabled&next(P5)=1&next(P6)=0&next(P1)=P1&next(P2)=P2&next(P3)=P3
&next(P4)=P4;
|
-- selfloop for deadlocks
```

```
deadlock&next(P1)=P1&next(P2)=P2&next(P3)=P3&next(P4)=P4&next(P5)=P5
&next(P6)=P6;
)
DEFINE
T1.enabled:=P2;
T2.enabled:=P1&P3;
T3.enabled:=P4&P5;
T4.enabled:=P6;
deadlock:=!(T1.enabled|T2.enabled|T3.enabled|T4.enabled);
SPEC
AG(!(T1.enabled)| AF(T4.enabled))
```

Figure 2.4 A SMV program for the producer-consumer system

3.5.4 Analysis Using SPIN

In order to verify PrT net models using SPIN, we must translate the PrT model into a Promela program. It is straightforward to translate low-level Petri nets models into Promela models. From intuition, we can translate a high-level Petri nets model into a low-level model, and then translate the low-level Petri nets models into a Promela program. But even though it is possible in theory to translate high level Petri nets this way; it is not practical since there is no good and general way to translate a high level Petri nets model into a low level one except by unfolding the high level model. However, the unfolded model sometimes will be huge even impossible if the predicate type is infinite, and the translation is tedious but the solution is low efficiency [GP98]. We limit predicate type to be finite, and any element in an arc label can only be an enumerable type.

The basic idea to translate PrT nets into Promela programs is to translate predicates in nets into variables in Promela, and translate each transition from nets into an atomic sequence. In the atomic sequence, each combination situation of the input variables of the transition is tested, and its corresponding output is to set its related variables. The initial marking is translated into variable initialization in Promela program. Global variables and channel variables are used to synchronize different processes. To a model includes several PrT nets, each PrT net is translated into as a process. This is a clear and simple way, but the process body may be huge if the net has many transitions. The following figure is the framework to translate a PrT net into Promela program:

```
#define const            value
......
declaration of global variables
proctype PrTModel1(parameters)
{
declaration of local variables
initialization
do
::atomic { t1 → do //list each possible value for t1 input
                  //variables
                ::case1 →
                     assign values to t1 input variable;
                     case1 = false;
                ::......
                ::casen →
                     assign values to t1 input variable;
                     casen = false;
                od
                unless { (input tokens are satisfied) &&
                         (t1 condition is satisfied) ||
                         (!(case1||......||casen)) };
                if
                ::!(case1||......||casen) → t1 = false;
                ::else → t1 fired and marking updated
                          t1 = true;
                fi;
                case1 = true; ...... ;casen = true;
          }
::atomic {t2 → do ......
              ......
         }
   ......
::atomic {tn → do ......
              ......
         }
::!(t1||t2|| ...... ||tn) → goto dead
od;
dead: deadlock = true;
}
init
{ atomic { run t1(initialization value);
                run t2(......); ...... ;
                run tn(......)
         }
}
```

Figure 2.5 A Framework to translate a PrT net into a Promela program

According to above discussion, we give an example to illustrate the idea of translating a PrT net into a Promela program:

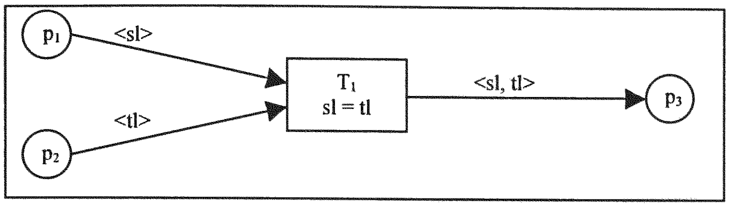

Figure 2.6 A PrT net model

```
#define true        1
#define false       0

proctype Test(byte p1, byte p2, byte sl, byte tl)
{
 bool case1, case2, case3, case4;
 bool t1 = true;
 byte p3 = 0;
 case1 = true; case2 = true;
 case3 = true; case4 = true;

 do
   ::atomic { t1 → do
                  ::case1 → sl = 0; tl = 0; case1 = false;
                  ::case2 → sl = 0; tl = 1; case2 = false;
                  ::case3 → sl = 1; tl = 0; case3 = false;
                  ::case4 → sl = 1; tl = 1; case4 = false;
                  od
                  unless { (p1 > 0 && p2 > 0) && (sl == tl) ||
                           (!(case1||case2||case3||case4)) };
                  if
                  ::!(case1||case2||case3||case4)→t1 = false;
                  ::else → p1 = p1 - 1; p2 = p2 - 1;
                           p3 = p3 + 1;
                           t1 = true;
                  fi;
                  case1 = true; case2 = true;
                  case3 = true; case4 = true;
       }
   ::!(t1) → goto dead
   od;

   dead: deadlock = true;
 }

 init { atomic { run Test(1, 1, 0, 0 ) } }
```

Figure 2.7 A Promela program of the Figure 2.6

Here we use *init* process to initialize the p_1 and p_2 token number and the predicates *sl* and *tl*, but this only reflect one scenario of the PrT net running. If we need to check all scenarios, we set p_1 and p_2 with all possible values, so the atomic sequence will check each possible combination of *sl* and *t1*. The method to initialize the initial marking in *init* process results in a simpler model and is highly efficient. We can even initialize multiple processes to simulate diffident initial marking, so we can check all situations at the same time, and it is easier to find errors in the model. But it is obvious this method is not suitable for complex initialization since it is a tedious work to manually set the initialization in *init* process, but we can let system randomly select possible values for variables if we set all predicates ready before they try all situations.

CHAPTER III

Predicated/Transition Nets Extended with Channels

1 Introduction

In order to facilitate the interaction and communication between nets, we introduce dynamic channels to PrT nets. A channel is a mechanism associated with transitions to send and receive messages between transitions in different nets. Channels have two forms — an output channel for sending messages to the channel, which is a channel name followed by an exclamation mark and real parameters, such as *mc!(type, msg);* and an input channel for receiving and removing messages from the channel, which is a channel name followed by a question mark and parameters such as *mc?(type, msg)*. Channel names can be variables or constants, and they are effective in the whole model. We define the type of channel variable as a finite set of strings (pre-defined a set of names). When any value with primitive type is assigned to a channel variable, its type is converted to a string automatically. Any structured type of value cannot be assigned to a channel variable directly. The parameters of an output channel must exist in its inscriptions of the channel input arcs. An output channel sends values of its parameters (structured data such as *(type, msg)*) into the channel when the transition with the output channel fires. When a transition with an input channels fires, it removes data from the corresponding channel and sends it to its output places. In addition, an input channel is also a condition to enable the associated transition. We treat each input channel as a proposition in the transition inscription, and if the channel is empty, its value is *false*. If the input channel has the corresponding data, then the channel proposition is *true*. For simplicity, we define a synchronization channel, which is a special channel with the buffer size being zero. When a synchronization output channel is enabled (actually its transition is enabled), it outputs one structured datum to the channel. However, its transition cannot fire until the corresponding input channel is enabled and then they fire at the same time. We put a zero on right top of channel name such as c^0 to denote a *synchronization channel*. Synchronization channels are our most interested channel type, which we chose to model the software architecture of mobile agent systems. However, in order to precisely define the synchronization channels, we introduce the general dynamic channels firstly, and then discuss synchronization channels with some restrictions on the general dynamic channel.

Communication between two transitions in different nets only happens when these two transitions have matched input and output channels. An input channel matches an output channel when they have the same channel name, same number of formal parameters, and each corresponding parameter has compatible type. For example, there is an output channel *dl!(msg)*, it sends *msg* to channel *dl*, then only *dl?(ms)* but not *dl?(ms, x)* can get the data *msg*. This restriction is helpful to share channels. However, it may also cause some problems. Suppose there are two transitions t_1 and t_2 with channels *dl!(msg)* and *dl!(type, msg)*, and there are two other transitions e_1 and e_2 with channels *dl?(ms)*, and *dl?(type, msg)*. 1. If t_1 fires, and then t_2 fires, the channel *dl* has (*msg*) and (*type, msg*). If each real parameter in channel is an independent data unit, then e_1 does not know which message it needs to pick up in the channels when it fires. 2. If we design channels as sequential data structure, which means each datum saved in a channel has an order, and accessing data in channels must follow the order, it may cause deadlock. Suppose channels are first in and first out (FIFO). If t_1 fires, and then t_2 fires, the channel *dl* has (*msg*) and (*type, msg*). If e_1 is only enabled after e_2 fires, however, e_2 is enabled only when it can get (*type, msg*) firstly. We found e_1 has to fire and remove (*msg*) from the channel firstly, and then e_2 has chance to fire. Then the deadlock happens. Therefore, we treat channels as a non-ordered structure. That means data in channels can be accessed randomly. In addition, a parameter of a channel has a structured data type, and concrete data are wrapped as structured data according to its formal parameters.

When we send one message to several channels, we can use multiple channels such as *dl!da!(msg)*, that means message *msg* is output to channel *dl* and *da* at the same time. In the example *dl!da!(msg)*, the *msg* is input to channels *da* and *dl, and dl?(type)da?(msg),* the *dl* gets data for *type* from *dl,* and *da* gets data for *msg* from *da*. The value for channel names (channel variables) can be empty (represented as *¢*). When a channel is empty, it is ignored. Such as in *dl!da!(msg)* and *dl?(type)(da?(msg)*, if *da* is *¢*, then *dl!da!(msg)* is the same as *dl!(msg)*. There are maybe several channels in one transition, but each transition only has one type of channel. These channels may work concurrently using the *and* operator *&&*, or work in conflict using the *or* operator ||. The input or output operations for channels are also qualified with transition conditions or other conditions. If transition has the following inscription: *((da = ¢) && dl!(msg))* || *((da ≠ ¢) da!(msg))*, this means if *da* is *¢*, then *msg* is sent to channel *dl*, if *da* is not *¢*, *msg* is sent to channel *da*.

2 Formal Definition of CPrT Nets

Based on the informal discussion about the PrT nets extended with channels, we will formally define its structures and behaviors in this section. We call the PrT nets extended with channels as CPrT nets. Then we will discuss the behavior equivalence between PrT nets and CPrT nets. Finally, we will introduce the two-layer paradigm of EOS into CPrT nets, and discuss the communication, cooperation of nets between different nets. For convenience, we introduce an operator \ in the following discussion. The operator \ is used to remove items from an expression, such as $\varphi(t)\backslash c$ means removing expression *c* from expression $\varphi(t)$, and if *c* is a channel, the operation removes the channel expression (channel name, its type and parameters) from the expression $\varphi(t)$. We define $\sigma(x)$ as the value of *x*, and *dom(x)* as the possible values of *x*.

2.1 Definition

Definition 2.1.1 (Channel). A *channel C* is an interaction relation between transitions in a PrT net model, $C = (T, E, \rho)$ where:

1. T is a transition in a *PrT* net $N = (P, T, F, \Sigma, L, \varphi, M_0)$, and it is associated with an output channel $mc!(d_1, \ldots, d_n)$, where mc is the channel name, and $d_1, \ldots, d_n$ are formal parameters for mc.
2. E is a transition in a *PrT* net $N' = (P', E, F', \Sigma', L', \varphi', M'_0)$, and it is associated with an input channel $mc'?(d'_1, \ldots, d'_n)$, where mc' is the channel name, and $d'_1, \ldots, d'_n$ are formal parameters for mc.
3. T can communication with E through channel mc, if $\sigma(mc) = \sigma(mc')$,and $dom(d_i) \subseteq dom(d'_i)$, $1 \leq i \leq n$, under marking (M, M'), where M is the marking of N, and M' is the marking of N'.
4. The *buffer* size of *a* channel is finite, and messages in a channel are accessed randomly.
5. $\rho \subseteq T \times E$ is the interaction relation. The transition associated with output channel such as T sends values *of* $(d_1, \ldots, d_n)$ to channel mc when it fires, and transition such as E associated with input channel gets values for $(d_1, \ldots, d_n)$ when the values in channel mc are available and removes the data from the channel mc when the transition fires.
6. The input channel mc is a guard condition to enable transition *E and T.* If mc is empty, the transition E cannot fire. If channel mc is full, then the transition associated with $mc!$ cannot fire until mc has available space.

A channel or channel expression has three parts: channel identifier, channel type, and parameters. A channel identifier could be a variable or a constant. If it is a variable, it should occur at adjacent input arcs of the transition that has the channel. A channel has two types: *input channel,* which is a channel name followed by a '?', and *output channel,* which is a channel name followed by a '!'. The parameters define the kind of information that can be passed through a given channel. An output channel expression has the form like $c!(p_1, p_2)$, where c is channel identifier, and (p_1, p_2) are its parameters, which is a structured data. A channel expression is part of the transition inscription if the transition has that channel. Channel expressions combine with other inscriptions using '&&' (*and*) or '||' (*or*) operator. Based on the definition of channels, we give the definition of CPrT nets:

Definition 2.1.2 (CPrT nets), A *CPrT* net is a tuple $(P, T, F, \Sigma, L, \varphi, M_0, C, W)$, where:

1. $(P, T, F, \Sigma, L, \varphi, M_0)$ is a *PrT* net.
2. C is a finite set of channels, $\forall c \in C$, $c = (CI, CT, CP)$
   1.1 CI is the channel identifier, *and* $CI \in \Sigma$
   1.2 CT is the channel type, *and* $CT \in \{!, ?\}$
   1.3 CP is the channel parameter or data passed through the channel, and $CP \in \Sigma$
2. W *is* a finite set of transitions, *and* $W \subseteq T$, $\forall t \in W$, $\exists c \in C$, $\varphi(t) \backslash c \neq \varphi(t)$.

2.2 Behaviors of CPrT Nets

Since channel is a new concept to PrT nets, we discuss behaviors of CPrT nets especially channel behaviors in this section. The only difference between CPrT nets and PrT nets is some transitions in CPrT nets include channel expressions, which affect the firing rules of these transitions. Adding input channel expressions to a given transition constrains its enabling, but it does not affect transitions with output channels. However, transition firings with output channels enable some transitions with input channels.

Let ${}^\bullet t = \{p \in P : (p,t) \in F\}$ and $t^\bullet = \{p \in P : (t,p) \in F\}$ be the pre-condition predicates and the post-condition predicates of transitions t, respectively. Let ${}^\bullet p = \{t \in T : (t,p) \in F\}$ and $p^\bullet = \{t \in T : (p,t) \in F\}$ be the sets of input transitions and output transitions of predicate p, respectively. We treat an input channel expression as a boolean expression, and it evaluates *true* when its formal parameters are concreted, and *false* if its parameters are empty. We treat parameters of each channel as a structured data structure. Take channel expression $c!(p_1,p_2)$ as an example. Here p_1 and p_2 compose the data structure (p_1, p_2). If channel c is empty, that means nothing with structure (p_1, p_2) in the channel c, so the expression evaluates *false*, but (¢, ¢) means its parameters have value (¢, ¢), and the expression $c!(p_1,p_2)$ evaluates *true*. In the following section, we discuss behaviors of three different transitions: transitions without channels, transitions with output channels, and transitions with input channels.

1. Transitions without channels: transition t does not have any channel expression within its inscription $\varphi(t)$. Transition t is enabled under marking M_0 if there is a substitution θ such that $l/\theta \in M_0(p)$ for any label $l \in L(p, t)$ for all $p \in {}^{\bullet}t$ and $\varphi(t)$ evaluates *true* with regard to θ, where l/θ yields a token by substituting all variables in label l with the corresponding bound values with regard to θ. The firing of an enabled transition t removes all tokens in $\{l/\theta\colon l \in L(t, p)\}$ from each input predicate $p \in {}^{\bullet}t$, and adds all tokens in $\{l/\theta\colon l \in L(t, p)\}$ to each output predicate $p \in {}^{\bullet}t$. After the firing of t, we get a new marking M'. Formally, $M_1(p) = M_0(p) - \{l/\theta\colon l \in L(p, t)\}$ for any $p \in {}^{\bullet}t$, and $M_1(p) = M_0(p) \cup \{l/\theta\colon l \in L(t, p)\}$ for any $p \in {}^{\bullet}t$.

2. Transitions with output channels: transition t has a necessary output channel expression c within its inscription $\varphi(t)$. We denote $c.CI$ as the channel identifier, $c.CT$ as the channel type, and $c.CP$ as the channel parameters. Transition t is enabled under marking M_0 if there is a substitution θ such that $l/\theta \in M_0(p)$ for any label $l \in L(p, t)$ for all $p \in {}^{\bullet}t$ and $\varphi(t)\backslash c$ (without considering the output channel expression c) evaluates to *true* with regard to θ. Where l/θ yields a token by substituting all variables in label l with the corresponding bound value θ, and substituting $c.CI$ and $c.CP$ with the value θ $(c.CI/\theta, c.CP/\theta)$. The firing of the enabled transition t removes all tokens in $\{l/\theta\colon l \in L(p, t)\}$ from each input predicate $p \in {}^{\bullet}t$, and adds all tokens in $\{l/\theta\colon l \in L(t,$

p)} to each output predicate $p\in{}^{\bullet}t$. It also adds a new output place λ and $\lambda = c.CI/\theta$ for *t*, where $t^{\bullet} = \lambda \cup t^{\bullet}$, an arc γ from *t* to place λ, and a label ψ on γ where ψ is <*c.CI/θ, c.CP/θ*>, $L(t,p) = L(t,p) \cup \psi$, $p \in t^{\bullet}$, and adds token *(c.CP/θ)* into place λ. The following diagram shows the firing rule of output channels. For simplicity, we rewrite the transition inscription as *φ(t)&&c*, where *c* is a channel expression, and *φ(t)* does not include any other channel expressions.

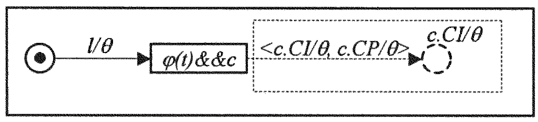

Figure 3.1 Firing a transition with an output channel

3. Transitions with input channels: transition *t'* has a necessary input channel expression *c'* within its inscription *φ'(t)*. The transition *t'* is enabled under marking M_0 if there is a substitution θ' such that $l/\theta' \in M_0(p)$ for any label $l \in L(p, t')$ for all $p\in{}^{\bullet}t'$ and $\varphi(t')\backslash c'$ (without considering the input channel expression *c'*, *c'* is *false* at this time) evaluates to *true* with regard to θ'. Where l/θ' yields a token by substituting all variables in label *l* with the corresponding bound value θ', and substituting *c'.CI* and *c'.CP* with the value θ', i.e. *c'.CI/θ'* and *c'.CP/θ'*. At the same time, a transition *t* has an output channel expression *c* within its inscription *φ(t)*, and *t* is enabled under marking M_0 if there is a substitution θ. If *c'.CI/θ* = *c.CI/θ'*, *c'.CT* = *?*, *c.CT* = *!*, and *c'.CP/θ'* = *c.CP/θ* (their parameters match). When transition *t* fires, channel expression *c'* becomes *true*, so *φ(t')* is true, and transition *t'* is enabled. If there is more than one transition when *c'* is enabled, and then which transition will fire is non-deterministic. The firing of *t* follows rules in (2), and then adds a place λ (added in (2) for output channel) $\lambda = c.CI/\theta$ as a new input place for *t'*, ${}^{\bullet}t' = \lambda \cup {}^{\bullet}t'$, an arc γ from place λ *to* transition *t'*, and a label ψ on γ where ψ is <*c.CI/θ, c.CP/θ*>, $L(p,t') = L(p,t') \cup \psi$, $p\in{}^{\bullet}t'$. The firing of the enabled transition *t'* removes all tokens in

$\{l/\theta': l \in L(p, t')\}$ from each input predicate $p \in {}^{\bullet}t'$, and adds all tokens in $\{l/\theta': l \in L(t', p)\}$ to each output predicate $p \in t'^{\bullet}$. The value of channel expression c' depends on the matched output channel when t' fires. After firing of t and t', we get a new marking M'. Formally, $M'_1(p) = M'_0(p) - \{l/\theta: l \in L(t, p)\} - \{l/\theta': l \in L(t', p)\}$ for any $p \in {}^{\bullet}t$ or $p \in {}^{\bullet}t'$, and $M'_1(p) = M'_0(p) \cup \{l/\theta: l \in L(t, p)\} \cup \{l/\theta': l \in L(t', p)\} \cup \{l/\theta: l \in L(t, p)\}$ for any $p \in t'^{\bullet}$. The following diagram shows the firing rule of input channels. For simplicity, we rewrite the transition inscription as $\varphi(t')\&\&c'$, where c' is a channel expressions, and $\varphi(t')$ does not include any channel expression.

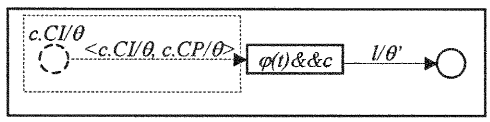

Figure 3.2 Firing a transition with an input channel

4. Other cases. In the above paragraphs, we discussed a basic situation of channel expressions, which are required items in transition expressions, and each inscription expression only has one channel expression. In this section, we discuss the general situations where channel expressions are combinations of several channels expressions. All of these cases can be transformed to the basic case.

4.1 For a transition t with inscription $\varphi(t)||c!(p_1, p_2)$ or $\varphi(t)||c?(p_1, p_2)$, the transition t can be split as two transition t_1, and t_2. The transition t_1 keeps the original structure but with inscription as $\varphi(t)$. The transition t_2 has the same input, output places, arcs and labels as t_1, but with inscription as $c!(p_1, p_2)$, or $c?(p_1, p_2)$.

4.2 For a transition t with inscription $c_1!(p_1, p_2)\&\&c_2!(s_1, s_2)$, when it fires, two places $c1.CI$ and $c2.CI$ are added with suitable arcs and labels according to the discussion in (2).

4.3 For a transition t with inscription is $c_1?(p_1, p_2)\&\&c_2?(s_1, s_2)$, it includes $c_1.CI$ and $c_2.CI$ as its input place, $c_1.CI, c_2.CI \in {}^{\bullet}t$.

4.4 For all other cases, we can reorganize models according to the rules we discussed and the rules of PrT nets, and transform the models to basic cases. Such as for a transition that has both an input channel expression and an output channel expression, we can split the transition into two transitions, one with only input channel expressions, and another with only output channel expressions.

3 Transform CPrT Nets into PrT Nets

Before we can simulate and analyze models that are defined using CPrT nets, we formally define CPrT nets semantics. We interpret CPrT nets semantics using ordinary PrT net semantics through transforming CPrT nets to PrT nets, and then prove that these two models are behaviorally equivalent, which means there is one to one correspondence between markings and enabled steps of the two nets [CH94]. We explain CPrT nets using PrT nets, but we never really transform CPrT nets into PrT nets when we describe a system. We always define a system directly using CPrT nets without constructing the equivalent PrT nets.

In CPrT nets, some transitions are associated with input or output channels. This is the only difference between CPrT nets and PrT nets. We can transform these transitions into regular transitions through adding some predicates and input/output functions. After we transform all transitions in a CPrT net into regular transitions, this special PrT net is transformed into an ordinary PrT net. We can use ordinary PrT net rules to interpret, analyze and simulate the transformed models.

3.1 Output Channels

We regard each output channel as an output place for the transition that has the output channel, and the parameters for the channel as inscriptions on the arc that is from the transition to the output place. The place is assigned the same name as that of the channel, and it is unique within the global domain. The following diagram shows the idea of the transformation, and the place name p is a variable that is assigned with a real value at run time. We call this diagram as the dynamic view of channels.

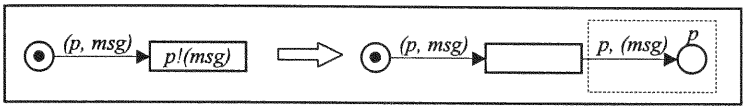

Figure 3.3 A dynamic view of an output channel

When a transition with an output channel fires, it instantiates the channel name (if the channel name is a variable) and the channel parameters according to the tokens in its pre-conditions and inscriptions on the arcs, wraps the instantiated parameters as structured data, and send the data to the channels. It is straightforward to transform an output channel as a post place of the transition, and inscriptions on the arc from the transition to the place are the channel name and the parameters of the channel, and channel expression in the transition inscription is removed. When a transition with one output channel fires, the instantiated parameters are saved in the output channel. This is equivalent to output a structured token to a place, which represents the output channel. The token has the same structure as the channel parameters, and it is instantiated with the same values as those in the output channel. This is guaranteed by the inscriptions, which are the same as output channel parameters, on the arcs.

Although we need to transform CPrT nets into PrT nets for analysis purposes, the extension brings us great convenience to model dynamic configuration and communication between different transitions of especially multi-level models. Since channel names might be variables, their values are assigned at run time. Place names in regular PrT nets are pre-defined, so they cannot change at run time. This is an important difference between PrT net places and channel places. When we transform a channel transition into a regular transition, the transformed transition is connected to a set of places through auxiliary places and transitions. Each place in the set has a unique name from the possible values of the channel, and each possible value of the channel has one corresponding place that has the value as its name. We call this view of transformation as the static view of channels. The following diagram shows a static view of an output channel:

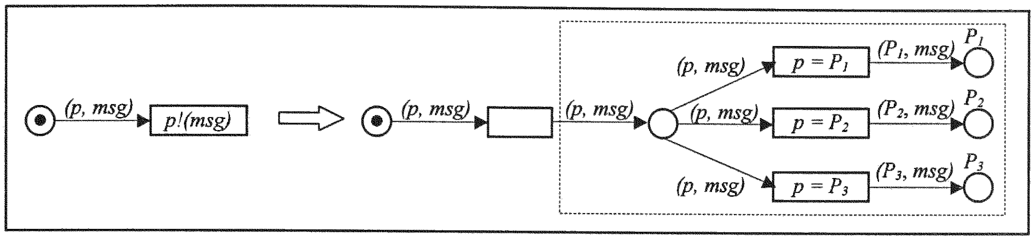

Figure 3.4 A static view of an output channel

In Figure 3.4, the right side is an output channel, and left side is its transformed PrT net. Suppose p only has three possible values: P_1, P_2, and P_3. The set of channel values is always finite since channels are finite for any system, and each possible value has one corresponding transition to put data into particular place that represents the channel. The subnet within the dashed square in Figure 3.4 equals to the dash area in Figure 3.3.

3.2 Input Channels

We treat each input channel as an input place (also a pre-condition) for the transition that has the input channel, and the parameters for the channel as inscriptions on the arc which directs from the input place to the transition. The place is assigned a name same as the channel name, and it is unique within the global domain. The following diagram shows the transformation:

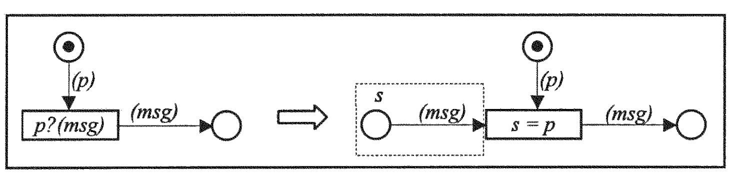

Figure 3.5 A dynamic view of an input channel

An input channel should have at least one possible corresponding output channel. If an input channel name is a variable, the run time value of the input channel name should have a matched output channel, i.e. the output channel name has the same value as the input channel name at that time. The pre-condition representing the input channel becomes *true* when the corresponding transition with the output channel fires. If the input transition is enabled, it may fire and remove data from the channel to its output places according to the firing rules. If there are several transitions with the corresponding input channel expressions are enabled at the same time, then which transition will fire is non-deterministic.

It is straightforward to transform an input channel as an input place of the transition, and inscriptions on the arc from the place to the transition are parameters of the channel. When the transition with an input channel fires, the values of the channel parameters are removed from the channel and put into the output place of the transition. This is equivalent to moving out a structured token from the input place, which represents the input channel, and put the token into output places of the transition. The token in the input place has the same structure as the parameters of the corresponding output channel. Channel names might be variables with values instantiated at run time. When we transform a channel transition into a regular transition, the transformed transition is connected with a set of places. Each place in the set has its unique name from the possible values of the channel, and each possible value of the channel has one corresponding place that has the value as its name. The following diagram shows a static view of an input channel:

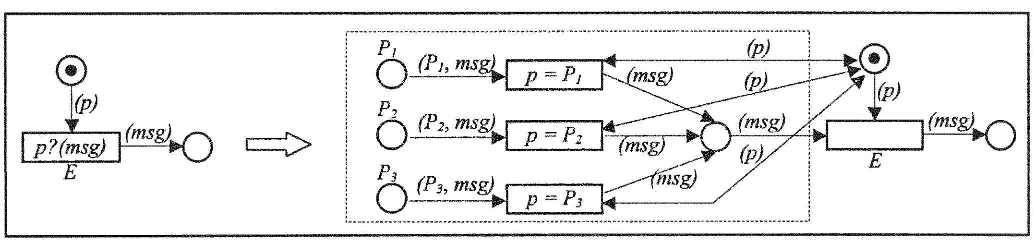

Figure 3.6 A static view of an input channel

In Figure 3.6, the right side is an input channel, and left side is its transformed PrT net. Suppose channel p has three possible values: P_1, P_2, and P_3. Transition E gets tokens from particular place (or channel) according to the run time value of p. If p is a constant, only one channel place is required to connect with transition E, then other auxiliary places, transitions, and arcs are not necessary any more. The subset within the dashed area in Figure 3.6 is equal to the dashed area in Figure 3.5.

3.3 Communication between Channels

In order to facilitate the communication and interaction between different transitions, we extend PrT nets with channels. Since each output channel sends data to its channel, and the corresponding input channel gets and removes data from the channel, we need to connect output channels with its matched input channels in PrT nets to study the communication between transitions with matched channels. After we transform channels to PrT net, each output channel transition connects with their corresponding input channel transition by merging places with the same name. The following diagram shows the dynamic view of connection:

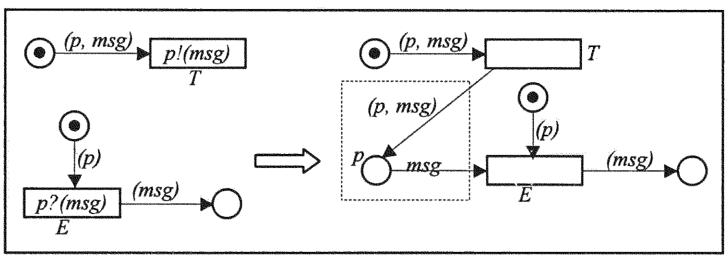

Figure 3.7 A dynamic view of the communication

When a transition with one output channel fires, it puts tokens into the place representing the channel that has the channel name as its name. The place representing the channel is one of the preconditions for all transitions that have the input channel expression in their inscriptions as required conditions. If any of these input transitions fires, the tokens in the place is moved to the post-condition of the transition. Therefore, we can transform a CPrT net into a PrT net, and the net follows the ordinary PrT net rules. The place *p* here is shared by transition *T* and *E* as post-condition and pre-condition respectively. This is a dynamic view of the connection since *p* is a variable. We can unfold *p* as a set of places with some auxiliary places and transitions to form a static view of the connection. We transform output channels and input channels into PrT nets, and merge channel places that have the same names. Then input channel transitions and output channel transitions are connected as a PrT net without channels, and the communication and interaction between these transitions follow the ordinary PrT net rules. The following diagram shows a static view of the communication:

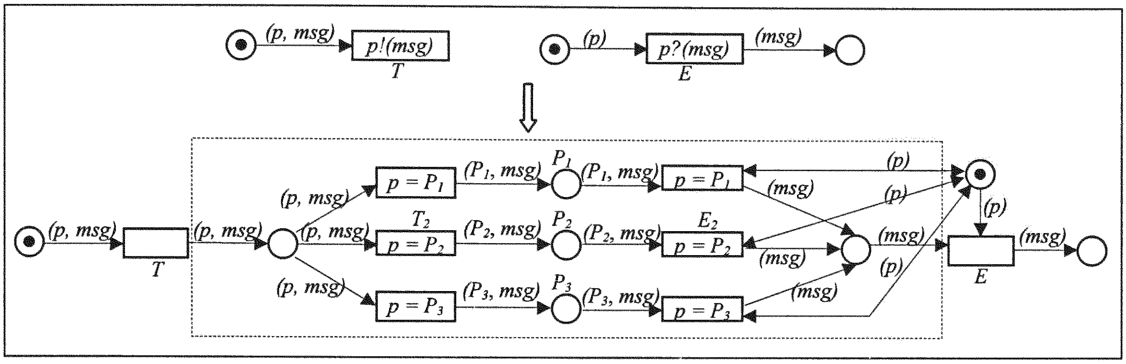

Figure 3.8 A static view of the communication

In Figure 3.8, output channel *T* and input channel *E* are transformed into PrT nets. Suppose channel *p* has three possible values: P_1, P_2, and P_3. Transition *T* sends tokens to particular place of P_1, P_2, or P_3 according to the run time value of *p*. Transition *E* gets tokens from particular place of P_1, P_2, or P_3 according to the run time value of *p*. Then places with the same name are merged, but there is no any other change. The subset within the dashed area in Figure 3.8 is equivalent with the dashed area in Figure 3.7.

In the diagrams for CPrT nets in Figure 3.7, transition *T* with channel *p* is enabled. Suppose value of *p* is P_2, and *msg* is *MSG* at one time. Before *T* fires, transition *E* with channel *p* is not enabled since there is no value for *msg* in channel P_2. As soon as *T* fires, channel P_2 gets data *MSG*. Then transition *E* is enabled. When *E* fires, it gets and removes data *MSG* from channel P_2, and sends *MSG* to its output place. Therefore, message is sent from input places of output channel *T* to output places of input channel *E*. The communication between transitions *E* and *T* is completed through the channel *p*. In the static view of transformed PrT net in Figure 3.8, transition *T* is enabled when its input place has tokens with structure (*p*, *msg*), which has value (P_2, *MSG*). Transition *E* is not enabled since one of its input places has not any token. When *T* fires, it only sends the token to place P_2 since only transition T_2 can fire. When P_2 get token (P_2, *MSG*), E_2 is enabled. When E_2 fires, E is enabled. E sends data *MSG* to its output place when it fires. Then message *MSG* from input place of T is sent to output place of E, the communication is completed.

4 Behavioral Equivalence of CPrT Nets and PrT Nets

In the previous sections, we discussed how to transform a CPrT net into an ordinary PrT net. We showed that CPrT nets can be transformed into behaviorally equivalent PrT nets. This means although adding channels to PrT nets increases the possibility for creating compact and comprehensive models, its computational power is the same as regular PrT nets. By behavioral equivalence, we mean a CPrT net has the same behaviors as its transformed PrT net. In other words, there is a one to one correspondence between the markings and the enabled steps of the two models. Therefore, we can generalize the basic concepts and the analysis methods of regular PrT nets to CPrT nets, and a CPrT net has a given property if and only if the equivalent PrT net has the corresponding property [CH94].

We call transitions with output channels as output transitions, and transitions with input channels as input transitions. Each auxiliary place has a unique name except those particularly addressed. We assume each transition has at most one channel, but the algorithm is easy to extend for transitions with multiple channels.

Algorithm 4.1 (Transform CPrT nets to PrT nets): Given a channel *PrT* net $CPrT = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$, *CPrT* can be transformed into a *PrT* net $N = (P', T', F', \Sigma', L', \varphi', M'_0)$ using the following steps:

1. Transform output channels into subnets without channels

   For each output transition $t \in T$, there is a channel c in the inscription $\varphi(t)$, and $c.CT = !$.

   Do the following steps:

   1.1 Add a place λ, and a directed arc γ from t to λ, the inscription on γ is $(c.CI, c.CP)$.

1.2 For each element λ_i in dom$(c.CI)$, *i.e.* $\lambda_i \in dom(c.CI)$, add a transition τ_i with inscription $(c.CI = \lambda_i)$. Add a directed arc γ_i from λ_i to τ_i with inscription on γ_i is $(c.CI, c.CP)$. $|dom(c.CI)| = |\bigcup \tau_i|$.

1.3 Add a place with name λ_i for each τ_i, and add a directed arc γ_j from τ_i to λ_i with inscription on γ_j is $(\lambda_i, c.CP)$.

1.4 Remove channel expression c from $\varphi(t)$ in t, $\varphi(t) = \varphi(t)\backslash c$.

Repeat the above steps until there is no output transition in *CPrT*.

2. Transform input channels into subnets without channels

For each input transition $t \in T$, there is a channel c in the inscription $\varphi(t)$, and $c.CT = ?$. Do the following steps:

2.1 For each element λ_i in $dom(c.CI)$, *i.e.* $\lambda_i \in dom(c.CI)$, add a place with name λ_i.

2.2 Add a transition τ_i for each place λ_i with inscription $(c.CI = \lambda_i)$, and add a directed arc γ_i from λ_i to τ_i with inscription on γ_i is $(\lambda_i, c.CP)$. If $s \in {}^{\bullet}t$, and there is a directed arc γ from s to t, and the inscription ψ on γ includes a required item $c.CI$, then add a bi-directed arc £ between s and τ_i with inscription $c.CI$. $s \in {}^{\bullet}\tau_i$ and $s \in \tau_i^{\bullet}$.

2.3 Add a place λ, and a directed arc γ_j from τ_i to λ, the inscription on γ_j is $(c.CI, c.CP)$.

2.4 Add a directed arc ξ from λ to t with inscription on ξ *is* $(c.CP)$

2.5 Remove channel expression c from $\varphi(t)$ in t, $\varphi(t) = \varphi(t)\backslash c$.

Until there is no input transition in the *CPrT* net.

3. Merge the same predicates

If two places have same name, there are fused as one place without other changes.

4. $N = (P', T', F', \Sigma', L', \varphi', M'_0)$, where P' is P combining with new generated predicates from channels, T' is T uniting with new generated transitions from channels, T' is F adding

with new generated arcs from channels, $\Sigma' = \Sigma$, $L' = L\bigcup(c.CI, c.CP)$, $\varphi' = \varphi \backslash c$, $M'_0 = M_0$ with new generated places is ϕ.

Based on above discussion, we arrive at a conclusion:

Proposition 4.1: Let a *CPrT* net $N = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$ is a *PrT* net with channels, there is a matching *PrT* net $N' = (P', T', F', \Sigma', L', \varphi', M'_0)$.

Proof: The proof can be derived from Algorithm 4.1.

5 Synchronization Channels

Synchronization channels are channels that buffer sizes are zero, and the input channel and the output channel have to fire at the same time when they communicate. For simplicity but without affecting expressive capacities to model mobile agent systems, we define input channel names as constants in synchronization channels. Synchronization channels are used for the communication between transitions that are in different nets. Each transition only has one type of channel so that there is no any direct circle between two communication transitions. There is no group communication or broadcast among channels. We put a zero on right top of a channel variable (not a constant) such as c^0 to denote a synchronization channel. When we model mobile agent systems using CPrT nets, we chose synchronization channels as the only one channel type to facilitate the communication. Therefore, we ignore the zero on any channel variable. Synchronization channels behave different to the general dynamic channels since both communication transitions have to fire at the same time. As soon as the two communication transitions fire, the communication completes and these two transitions have not synchronization relationship any more until they need to communicate again.

5.1 Behaviors of Synchronization Channels

There are two CPrT nets N_1, N_2 in a model, one net N_1 has a transition T with an output channel $c!(p_1, p_2)$, and another net N_2 has a transition E with an input channel $C?(p_1, p_2)$. Under certain marking (M_1, M_2), M_1 is the marking of N_1, and M_2 is the marking of N_2, T and E are enabled without considering channel expressions. Then if there is a substitution θ such that $l/\theta \in M_1(p)$ for one label $l \in L(p, T)$, where $p \in {}^{\bullet}T$ and c/θ evaluates the value of c as C, T and E fire at the same time, token (p_1, p_2) is removed from the input place of T and sent to output place of E. The new marking M'_1 of N_1 is $M'_1(p) = M_1(p) - \{l/\theta: l \in L(p, T)\}$ for any $p \in {}^{\bullet}T$, $M'_1(p) = M_1(p) \cup \{l/\theta: l \in L(T, p)\}$ for any $p \in T^{\bullet}$. When E is enabled under M_2, there is a substitution θ' such that $l/\theta' \in M_2(p)$ for any label $l \in L(p, E)$ for all $p \in {}^{\bullet}E$ and $\varphi(E)$ (without considering the input channel expression) evaluates *true*. The new marking M'_2 of N_2 is $M'_2(p) = M_2(p) - \{l/\theta': l \in L(p, E)\}$ for any $p \in {}^{\bullet}E$, $M'_2(p) = M_2(p) \cup \{l/\theta': l \in L(E, p)\} \cup \{l/\theta: l \in L(E, p)\}$ for any $p \in E^{\bullet}$.

5.2 Semantics of Synchronization Channels

The basic idea behind the transformation of a CPrT net with synchronization channels to an equivalent PrT net is to merge transitions that involve in the channel communication. When the transition with an output channel is merged with the transition that has the communication input channel, the arcs of the merged transition are the union of the arcs of the communication transitions. The guard condition of the merged transition is formed by the conjunction of the guards from the communication transitions and an expression to decide the equivalence of channel names of the input and output channel. Because the bindings of the communication transitions involve in a channel communication are independent, we have to make sure that set of variables of the communication transitions are disjoint before we merge the transitions [CH94]. For each communication transition t, we rename each variable $v \in var(t)$ with a new variable s of the same type as v, and make sure that the names of new variables are different. The following diagrams illustrate the transform procedure.

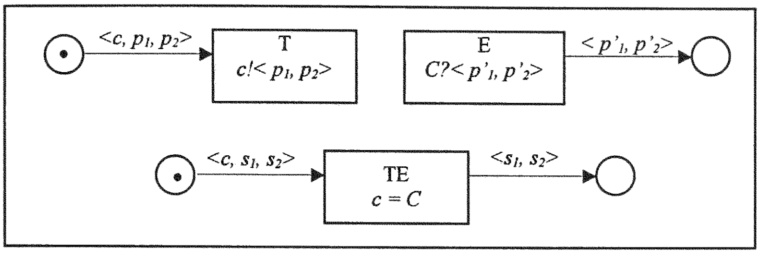

Figure 3.9 A transformation of a synchronization channel

The transition T and E are in different nets, and both of them are enabled under certain marking (M_1, M_2). The value of output channel variable c equals to C under the marking (M_1, M_2). The numbers of parameters of the input channel and the output channel are equal, and type of each corresponding parameter is compatible. Then transition T and E fire at the same time, and token of $<p_1, p_2>$ are removed from the input place of T to the output place of E. The enabling and firing sequences of the CPrT net are the same as the transformed PrT net, and the results of the CPrT net firing is the same as the results of the transformed PrT net.

Since the name of each input channel is unique in a model, the name of an output channel only can match at most one input channel at each time. However, it is possible to have more than one output channel matches one input channel under certain marking at the same time. In that case, only one output channel can communicate with the input channel at each time. Which output channel is chosen to communicate with the input channel is non-deterministic.

6 Semantics and Analysis of Two-layer CPrT Nets

The paradigm of two-layer Petri nets is defined in EOS [Val98]. A two-layer Petri net model includes a system net and some token nets, and token nets are packed as tokens in their system net. Tokens in a PrT net are structured data, which may include another PrT net. In other words, a PrT net can be packed as part of a token in other nets. We call the PrT net with net tokens as system net, and the PrT net, which is wrapped as tokens, as token net. This paradigm brings a hierarchical structure for PrT nets. Although we can design multiple layer models using PrT nets with channels, we limit our discussion only to a two-layer structure. In this section, we will discuss the communication and interaction between two layer PrT nets using channels.

The token nets are defined before they can be used in system nets. Each token net is a template. We treat these token nets as classes, and tokens as objects or instances, as in object-oriented systems. A two-layer CPrT net system may include several separate system nets and multiple token nets from the static point of view. Each token net has a unique identifier, which is the type identifier. Therefore, each object or instance of a token net is uniquely identified by its instance identifier and type identifier. We denote a token net instance as *(TI, TN)*, where the *TI* and *TN* is the instance identifier and the type identifier, respectively. Each token net may have multiple instances in system nets. However, instances are independent to each other except they are explicitly defined to cooperate. This restriction brings us much more convenience to formalize and implement this method.

6.1 Basic Situations

In PrT nets, tokens are simply predicates not embodying another net. In CPrT model, a token net is like a traditional CPrT net in this case. System nets are different, however, in that it may have tokens as other nets. In this section, we focus on the system nets with regard to the following basic situations:

6.1.1 Sequence

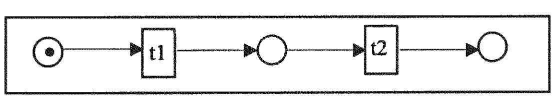

Figure 3.10 Sequence

If transition t_1 is enabled, then t_1 will fire. The token is moved to the next place. The state of the token net in the next place depends on the interaction between the system net and the token net. If t_1 does not interact with any transition in the token net, the state of token net does not change when it is move from input place of t_1 to its output place. It is called *transport*. If t_1

interact with some transition in token net and they fire at the same time, then the state of the token net in the output place of t_1 is updated with the firing of the transition. This is called *interaction* [Val98].

6.1.2 Synchronization

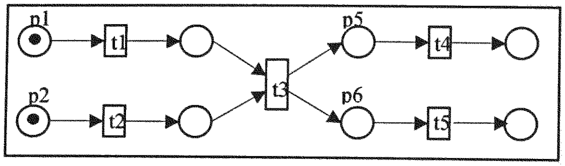

Figure 3.11 Synchronization

The tokens in p_1 and p_2 are independent instances of some agent nets. Their states are not related to each other, but they are synchronized at transition t_3. After t_3 fires, the tokens in place p_5 and p_6 still are different instances of some agent nets.

6.1.3 Conflict

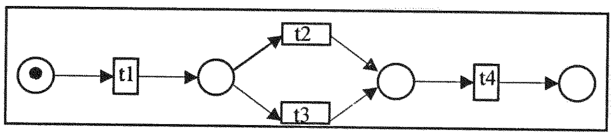

Figure 3.12 Conflict

The token states may be different either t_2 or t_3 fires. It depends on whether t_2 or t_3 directly or indirectly interacts with the token net and which transition fires.

6.1.4 Concurrency

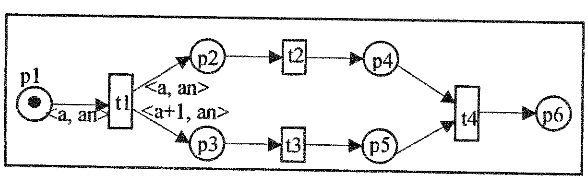

Figure 3.13 Concurrency

If t_1 fires, then tokens in place p_2 and p_3 are instances from the same token net if the token in p_1 is a token net. The states of tokens in p_2 and p_3 are the same but with different identifiers. We treat tokens in p_2 and p_3 as two independent instances, and the states of token nets in p_4 and p_5 are independent from each other. If t_4 fires, it generates a new token in p_6, but these two instances do not merge their states automatically so that the model has to explicitly design the merging if these two instances need to be merged.

Another situation is that the tokens in p_2 and p_3 referring to the same instance (the inscription on the label from t_1 to p_3 is $<a, an>$), so these two tokens still have same identifier. Therefore, at any time, these tokens at any place should share the same state. If t_2 fires, but t_3 does not fires, then the token states in p_3 and p_4 are the same, in other words, token state in p_3 is updated after t_2 fires if t_2 interacts with the token net. Then t_3 fires, the token state in p_4 are updated to the token state in p_5 if t_3 interacts with the token net. We do not consider this situation in our model even we can simulate this semantics using our model.

6.2 Semantics and Analysis

In this section, we will give formal definitions of the two-layer CPrT nets, the communication between system nets and tokens nets, and the communication between token nets.

Definition 6.2.1 (Two-layer CPrT nets): A two-layer *CPrT* net is a tuple $STN = (SN, TN, \rho)$, where:

1. SN is a finite set of system nets, $SN = \{SN_1, SN_2, ..., SN_n\}$, and SN_i $(1 \le i \le n)$ is a *CPrT* net, $SN_i = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$.
2. TN is a finite set of token nets, $TN = \{TN_1, SN_2, ..., TN_m\}$, and TN_i $(1 \le i \le m)$ is a *CPrT* net, $TN_i = (P', T', F', \Sigma', L', \varphi', M'_0, C', W')$.
3. $TN_i \in \bigcup_{i=1}^{n} SN_i \bullet \sum$
4. $\rho \subseteq W \times W'$ is the interaction relation

Now, we discuss occurrence rules of two-layer PrT nets. We focus on the interaction between system nets and token nets. According to which net activates the occurrence, we can distinguish three types of occurrences, which are system autonomous, interaction, and object autonomous.

This classification and concepts come from [Val98]. Suppose the marking of system net is M, and the marking of token net is M', then the marking of whole model is (M, M').

System autonomous means a transition in the system net fires and may move a token net from its input place to its output places, but the instance of the token net does not change its state, i.e. there is not any transition firing in the token net when the system net updates its state. That means the fired transition is a transition without channels, or the channels on the transition have not matched enabled synchronous transitions in the token nets. If the fired transition is t, and $M[t > M_1$, then the marking of the model changes as: $(M, M')[t > (M_1, M')$.

Interaction means a transition in the system net fires with a transition in token net at the same time. That means the fired transition is a channel transition and it has a matched synchronous transition in the token net. In other words, if the fired transition in system net with an output channel $c!$, there is an enabled transition with channel $c?$ in the token net. Interaction also can be activated by the token net. When a transition in the token net fires, it activates or enables a transition in the system net, and then the enabled transition fires, i.e. the token net and the system net update their states at the same time. The system update its marking after transitions fire: $(M, M')[(t, t') > (M_1, M_1')$, where t is the fired transition in system net and $M[t > M_1$, and t' is the fired transition in token net and $M'[t' > M'_1$.

Object autonomous means a transition of a token net instance fires and updates its state, but its system net does not fire any transition, i.e. a token net instance updates its state within a place of the system net. In other words, the fired transition in token net does not enable or activate any transition in system net. The system update its marking after transitions fire: $(M, M')[t' > (M, M_1')$, where t' is the fired transition in token net and $M'[t' > M'_1$.

We define the occurrence rules similar to the definition 2.2 in [Val99], but we extend them with channels instead of the texture synchronization variables. Suppose there is one system net

SN, a token net *TN*. The instance of *TN* in *SN* is *(TI, TN)*. We use *TI* to represent *(TI, TN)* if there is no confusion.

Definition 6.2.2 (Occurrence rules): There are three different occurrence rules:

1. *System autonomous:* A transition $t \in SN \cdot T$, and $TI \in SN.P$, M is the marking of *SN*, and M' is the marking of *TI*; t fires, $M[t > M_1$; but *TI* does not fire any transition, the marking of *TI* still is M'. Then the system marking is (M_1, M')

2. *Interaction:* A transition $t \in SN \cdot T$, and $TI \in SN.P$, M is the marking of *SN*, and M' is the marking of *TI*; t has an output channel such as $c!(p_1, p_2)$, $t' \in TI \cdot T'$ is an enabled transition with the input channel $c?(p'_1, p'_2)$, t fires, and then t' fires, $M[t > M_1$, $M'[t'> M'_1$. Then the system marking is (M_1, M'_1). *Or* $t' \in TI \cdot T'$ has an output channel such as $c!(p'_1, p'_2)$, $t \in SN \cdot T$ is an enabled transition with the input channel $c?(p_1, p_2)$, t' fires, and then t fires, $M[t > M_1$, $M'[t'> M'_1$. The system marking is (M_1, M'_1).

3. *Object autonomous:* A transition $t' \in TI \cdot T'$, $t \in SN \cdot T$, $TI \in {}^{\bullet}t$, or $TI \in t^{\bullet}$, M is the marking of *SN*, and M' is the marking of *TI*; t' fires, $M'[t'> M'_1$, but *SN* does not fire any transition, the marking of *SN* still is M. Then the system marking is (M, M'_1).

The interaction of occurrence rule defines the basic communication between system nets and token nets. We define the procedure of communication between system nets and token nets in the following definition. The operator $\leftarrow$ means assign right side values to left side variables.

Definition 6.2.3 (Communication between system nets and token nets): A transition $t \in SN \cdot T$, and $TI \in SN.P$, M is the marking of *SN*, and M' is the marking of *TI*; t has an output channel such as $c!(p_1, p_2)$, the inscription of t is $\varphi(t)$ && $c!(p_1, p_2)$. $t' \in TI \cdot T'$ is a transition with the input channel $c?(p'_1, p'_2)$, the inscription of t' is $\varphi'(t')$ && $c?(p'_1, p'_2)$.

1. Sending messages from the system net to a token net: If the value of $\varphi(t)$ && $c!(p_1, p_2)$ is true. t' is enabled, the value of $\varphi'(t')$ is true, $c?(p'_1, p'_2)$ is false ((p'_1, p'_2) is empty in channel c). Then t fires, $M[t > M_1$, $p'_1 \leftarrow p_1$, $p'_2 \leftarrow p_2$, and $(p_1, p_2) \leftarrow ¢$, $\varphi'(t')$ && $c?(p'_1, p'_2)$ becomes true, and t' fires, $M'[t'> M'_1$, The system marking is (M_1, M'_1).

2. Sending messages from a token net to the system net: It is symmetry as sending messages from a system net to a token net except the output channel is in token net, and input channel is in the system net.

If there are at least two instances of token nets in system nets, they may communicate each other. We define the procedure of communication between instances of token nets (we call these instances as object nets) in the following definition.

Definition 6.2.4 (Communication between object nets): There are two transitions $t_1 \in TI_1 \cdot T'$, $t_2 \in TI_2 \cdot T'$, and $TI_1 \in SN.P$, $TI_2 \in SN.P$. M_1 is the marking of TI_1, and M_2 is the marking of TI_2. t_1 has an output channel such as $c!(p_1, p_2)$, the inscription of t_1 *is* $\varphi(t)$ && $c!(p_1, p_2)$ and the value of $\varphi(t)$ && $c!(p_1, p_2)$ is *true*. $t_2 \in TI_2 \cdot T'$ is an enabled transition with the input channel $c?(p'_1, p'_2)$, the inscription of t_2 is $\varphi'(t_2)$ && $c?(p'_1, p'_2)$ and the value of $\varphi'(t_2)$ is true, $c?(p'_1, p'_2)$ is false ((p'_1, p'_2) *is* ¢). t_1 fires, $M_1[t > M'_1$, $p'_1 \leftarrow p_1$, $p'_2 \leftarrow p_2$, and $p_1 \leftarrow ¢$, $p_2 \leftarrow ¢$, then $\varphi'(t_2)$ && $c?(p'_1, p'_2)$ becomes true, t_2 fires, and $M_2[t'> M'_2$, $(p'_1, p'_2) \leftarrow ¢$. The system marking is (M, M_1, M'_1), where M is the marking of SN.

7 Concluding Remarks

In this chapter, we extend PrT nets with channels for synchronous communication between different transitions especially transitions within different nets. In addition, we also discuss how to introduce two-layer modeling paradigm from EOS to PrT nets. There are some related works. The first one is reference nets [Kum98] [KW99], which is a multiple-layer colored Petri nets extended with channels and other operators. We already introduced reference nets in the previous section. Here we just compare the differences between reference nets and CPrT nets. In reference nets, channel names are constants, but channel names are variables in CPrT nets. We call channels with constant names as static channels, and channels with variable names as dynamic channels. Dynamic channels are flexible and easy to model mobile computing systems especially modeling system architectures with dynamic configuration. Suppose there are three processes P_1, P_2, and P_3 in a system, and P_1 communicates with P_2 or P_3 through channels at difference time. In CPrT net, the name of output channel in P_1 dynamically changes at run time according its context to match the name of the input channel of P_2 or P_3, and then the communication changes from between P_1 and P_2 to P_1 and P_3. However, the model using static channels is more complex. In reference nets, the process P_1 has to define a set of conflict transitions that communicate with transitions in P_2 and P_3. All of these transitions have to be pre-defined. At the worst case, it has to define all possible communication between P_1 and all other processes statically. In other words, communication channels between processes are not shared, but they are used exclusively by two processes. However, channels are shared and created at run time, and channel name are variables, which are instantiated at run time by processes. This mechanism brings a more compact model and much more convenient to model dynamic reconfiguration of mobile agent systems. In reference nets, communication on channels are bi-direction on information flow, however, CPrT net distinguishes input channels and output channels. Bi-direction channels are useful to exchange messages between synchronization transitions. However, bi-direction channels also bring complexity of analysis, and they have side effects such as one synchronization transition does not want the partner synchronization transition to change some communication data. On the other side, uni-direction channels bring more works if one synchronization transition wants to exchange data with its communication partner transition but not just sending or receiving data. Reference nets extended on colored Petri nets with some new operators, which we have already introduced in previous chapters. CPrT net does not add any new operator to PrT nets since it has enough expressive power to model systems such as mobile agent systems. Certainly, reference nets are based on colored Petri nets, while CPrT nets are extended on PrT nets. In paper [SH94], channels are first introduced to colored Petri nets. The channel in the paper is the same as reference nets except with some syntax differences. Another important related work is EOS [Val99] [Val98], which defines the hierarchical Petri net. We extend this idea from Petri nets to PrT nets, and synchronous communication between nets is through channels not text labels. In addition, EOS has more difficulty than CPrT nets to deal with the dynamic interaction between system net and token nets. π-calculus [Mil99] is also an important related work because the dynamic channel concept in CPrT nets is similar to the channel concept in polyadic π-calculus.

CHAPTER IV

A Formal Architectural Model of Mobile Agent Systems

1 Introduction

Mobile agents are programs that can move from hosts to hosts in networks. In order to support the functionalities of mobile agents, each host needs at least one agent support system, which provides services and management. Therefore, we look at a mobile agent system as a set of agent support systems interconnected via networks, and a group of agents running within and migrating among them. We chose CPrT nets to model the software architecture of mobile agent systems, which includes a set of agent models and agent supporting system models. The following diagram shows the top-level architecture of a mobile agent system:

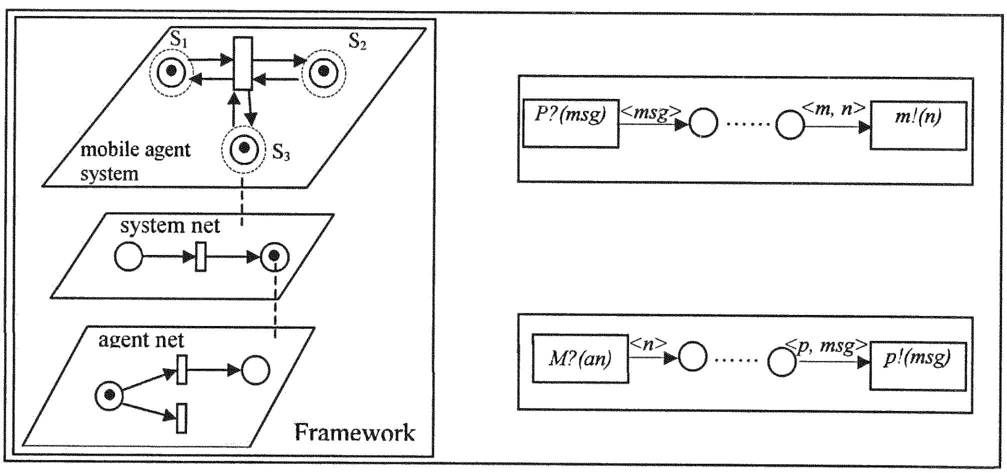

Figure 4.1 An architecture of mobile agent systems

We model the architecture of mobile agent systems as a hierarchical model: the top level is the system level model, the next level is the support system model, and the low level is agent model, which is running on support system models. In Figure 4.1, the S_1, S_2, and S_3 are three agent support systems, and these systems are located in different places (hosts) and connected with networks. The transition in the top-level diagram representing the mobile agent system is the inter-connection of agent support systems and agents can move among them. If we model low-level communication protocols, we can expand this connection as a communication model. The dashed circles in the diagram for a mobile agent support system represent agent support system models. We only show one place of each agent support system model in the dashed circle. The places in dashed circles are places where mobile agents are staying when they execute their tasks. The tokens in the top-level diagram represent agents that are packed as tokens in agent support systems, and they are modeled as agent models in the low-level models. The communication between nets is through dynamic channels. The output channels send out messages to channels, and input channels get and remove messages from channels at the same time. In Figure 4.1, *p* and *m* are channels with two types: input channels and output channels, where *p!*, *m!* is output channels, and *P?*, *M?* is input channels. The transitions with matched channels can communicate through these channels.

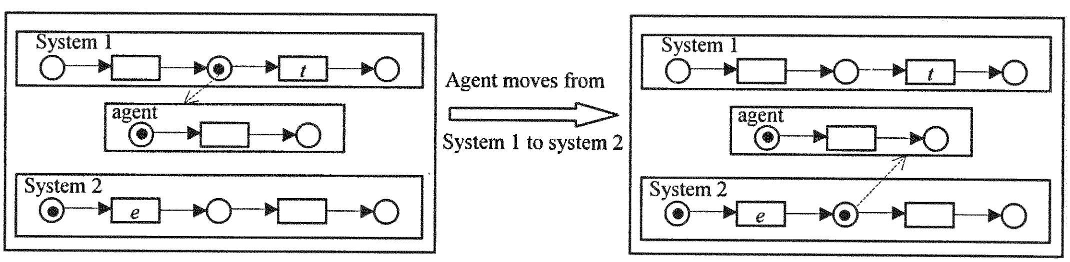

Figure 4.2 A dynamic configuration of mobile agent systems

The architecture of mobile agent systems is dynamically changed with agents creation, destroying, migration, and with some agent systems joining or leaving the system. The most difficult issue is to naturally capture this dynamic property since agent nets and system nets are statically defined, but contexts or environments of communication objects (agents or agent systems) are always changing. We resolve this issue through integrating dynamic communication channels into fixed defined PrT nets so that communication between objects can update with the changing of their contexts. From logic point of view, each CPrT net has at least one input channel to accept tokens from other nets, and one output channel to send tokens to a particular channel that connects to other nets or transitions. In Figure 4.2, the agent is in the system 1 before it moves out, then the transition *t* in system 1 fires, it moves the agent from system 1 to system 2 (we do not consider the synchronization between *t* and *e* here). The migration of the agent changes the system architecture. Figure 4.2 illustrates the dynamic configuration of mobile agent systems:

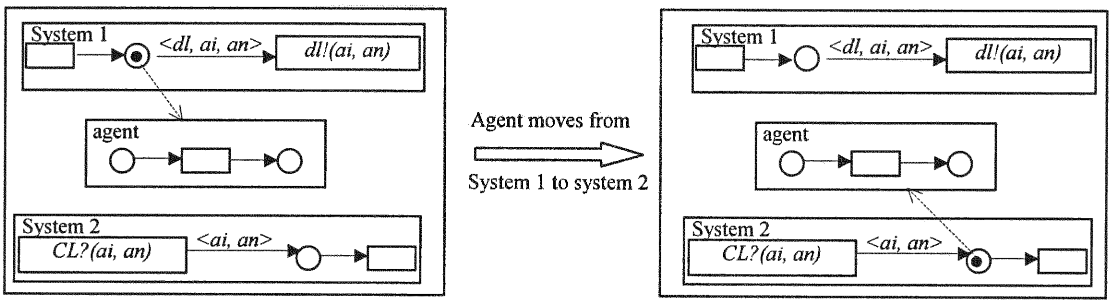

Figure 4.3 A communication between CPrT nets

Through channels, the migration of an agent from system 1 to system 2 is easy to be defined. The Figure 4.3 shows the basic idea of the agent migration and communication with dynamic channels in mobile agent systems. When system 1 moves the agent to system 2, it sends the token representing the agent to the output channel *dl* that connects to system 2, and suppose the variable *dl* = *CL*. System 2 gets the message from input channel *CL*, so the agent is moved from system 1 to system 2 since the input channel *CL?* has matched parameters (*ai, an*) as the output channel *dl!*. The communication between the agent and system 2 is realized through channels *dl!* and *CL?*. In the diagram Figure 4.3 *dl!(ai, an)* means sending the agent identifier *ai* and its net *an* to channel *dl*, which is a variable assigned value such as *CL* at run time. *CL?(ai, an)* means this transition try to get object *(ai, an)* from channel *CL*, *CL* is a constant and it is unique in the global doman and only for system 2. So different nets are connected with their channels, and the communication relationship is decided at run time. When the transition with *dl!(ai, an)* in system 1 fires, it sends the agent token (*ai, an*) to channel *CL*. Then the transition with *CL?(ai, an)* in system 2 fires, it gets the agent token from channel *CL* and puts it into its postset places. With the agent token moving from system 1 to system 2, the agent net disconnects from system 1 and then connects with system 2.

2 A Formal Architectural Model

Mobile agent systems essentially have a hierarchical structure – agents are supported by agent support systems, and agent support systems are supported by operating systems or other services and network infrastructures. The two-level CPrT nets naturally capture the multi-level properties of mobile agent systems. In the top level, a mobile agent system consists of a few support systems connected with networks, and they are modeled as host nets with connections. Each agent support system provides an execution platform for mobile agents, and agents cooperate with it to accomplish their tasks. Therefore, agents are packed as tokens in top-level models, which are defined as CPrT nets, and each agent is defined as a CPrT net in the low-level models. These CPrT nets are statically defined, but each mobile agent systems dynamically configure its architecture at run time when agents migrate from one system to another, or when agent systems become active or inactive in the network. In order to model this dynamic property, CPrT nets provide a dynamic channel mechanism to facilitate the dynamic communication and interaction between nets at run time. In this model, each agent support system supports all types of mobile agents, so we do not discuss interoperability issue in this paper. We do not consider security and other specific detailed issues such as locating an agent. However, we can plug these features into our models when we need to do further researches for those specific areas.

2.1 System Architecture

From the top-level view, a mobile agent system consists of agent support systems that are interconnected with connections, and agents run or migrate within these systems. We model this infrastructure as a set of agent support systems and a set of connections. The connection is the network connecting for agent systems, and it provides transportation services for mobile agents and agent systems. We only consider total connection situation, so that each agent can reach any other system that is connected and active in the network. Therefore, there is only one connection in our model. However, we define the connection as a set of connections for future extension.

We model each agent as a CPrT net, called agent net. The interfaces, behaviors, and states of an agent are modeled by some input/output transitions with channels for incoming/outgoing messages, the transitions, and the predicates of the agent net, respectively. The input/output transitions are transitions that send/get data to/from channels, which connect different CPrT nets. Particularly, a concrete state of the agent is the marking of the agent net. Besides, each agent uses input/output transition to dynamically connect to its agent support system when it moves into or move out from the agent support system. The following is the formal definition of agent net:

Definition 2.1.1 (Agent Net). Agent net *AN* is a tuple $AN = (P, T, F, \Sigma, L, \varphi, T_{in}, T_{out}, M_0, C, W)$, where:

1. $(P, T, F, \Sigma, L, \varphi, M_0, C, W)$ is a *CPrT* net
2. T_{in} $(T_{in} \subseteq W \subseteq T)$ is a finite set of input transitions associated with input channels for receiving incoming messages.
3. T_{out} $(T_{out} \subseteq W \subseteq T)$ is a finite set of output transitions associated with output channels for sending out outgoing messages.
4. The input transitions T_{in} and output transition T_{out} are the interfaces to communicate with agent support systems and other agents.
5. $\{<dt, dl, da, sl, sa, type, command, message>\} \subseteq P$

We define an agent *A* as a tuple $A = (A_i, MN_i)$, where A_i is the unique agent identifier, and MN_i is the corresponding agent net for A_i. Agents are distributed in agent systems by means of packing agents up as parts of tokens in system nets. In addition, each predefined instruction such as MOVE or GOTO is also contained in the structures of CPrT nets. There is one token type in agent net, which is *<dt, dl, da, sl, sa, type, command, message>*, where *dt* is the destination type, such as agent net or host net. *dl* the destination host, *da* the destination agent of the message, *sl* the source host of the message, *sa* the source agent of the message, *type* the message type, *command* the command for messages, and *messages* the content of the messages.

We model each agent system with a CPrT net, called a host net. The agent system provides facilities for agent execution (e.g., execution place, activation and deactivation). The interactions between agents and its system are through dynamic channels. Each host net has input/output interfaces to connect with other agent systems or agents, which are running within this agent system. The following definition is the formal definition of system net:

Definition 2.1.2 (Host Net). A host net *SN* is a tuple $SN = (P, T, F, \Sigma, L, \varphi, T_{in}, T_{out}, P_a, M_0, C, W)$, where:

1. $(P, T, F, \Sigma, L, \varphi, M_0, C, W)$ is a PrT net
2. T_{in} $(T_{in} \subseteq T)$ is a finite set of input transitions associated with input channels to receive incoming messages from the channels.
3. T_{out} $(T_{out} \subseteq T)$ is a finite set of output transitions associated with output channels to send outgoing messages to the channels.
4. T_{in} *and* T_{out} are the interfaces to communicate with other agent nets or system nets.
5. P_a $(P_a \subseteq P)$ is the only place where agents execute tasks.

In the structure Σ of a host net, we define $MN(P, T, F, \Sigma, L, \varphi, T_{in}, T_{out}, M_0, C, W)$ as structured data representing the structure of an agent, where $(P, T, F, \Sigma, L, \varphi, T_{in}, T_{out}, M_0, C, W)$ is an agent net. We use *MN* to represent agent net if it does not cause confusion. There is one token type in host net. The type structure is *<dt, dl, da, sl, sa, type, command, message>*, where: *dt* is destination object type, in this model, it is a boolean value, *true* means destination is an agent, *false* means the destination is a host. $dt \in \{true, false\}$. *dt* is added to make sure messages are only sent to unique channel at any time. *dl* is the destination host of the message, *da* is the destination agent of the message, *sl* is the source host of the message, *sa* is the source agent of the message. *type* is the message type, and it has two values, one is agent, another is regular messages. *type* $\in \{MSG, AN\}$. If type = *MSG,* then *command* is a command message. This command could be *MOV* which means to move out the source agent, or ¢ which means the message is regular data. If type = *AN,* then *command* is the agent identifier, and *message* is agent net *MN.*

Based on above concepts, we model a mobile agent system as a structurally composed model by a finite set of host nets, a finite set of agent nets and a logical connection. The logical connection provides facilities for communications and interactions among agents and agent systems. The logical connection is modeled through channels, and dynamic configuration of mobile agent system is reflected in the dynamic changes of channels. The following description is the formal definition of a mobile agent system (MAS) model:

Definition 2.1.3 (MAS model). A mobile agent system is a tuple $\Pi = (SYS, SAN, CONN)$, where:

1. *SYS* is a finite set of agent support systems $SYS = \{(DL_1, SN_1), (DL_2, SN_2), \ldots, (DL_n, SN_n)\}$, DL_i is the agent support system identifier and SN_i is the system net $(P_i, T_i, F_i, \Sigma_i, L_i, \varphi_i, T_{iin}, T_{iout}, P_{ia}, M_{i0}, C_i, W_i)$.
2. *SAN* is a finite set of agents $SAN = \{(DA_1, AN_1), (DA_2, AN_2), \ldots, (DA_p, AN_p)\}$, DA_i *is* the agent identifier and AN_i is the agent net $(P_i', T_i', F_i', \Sigma_i', L_i', \varphi_i', T_i'_{in}, T_i'_{out}, M_i'_0, C_i', W_i')$, and $AN_i \in \bigcup_{i=1}^{n} P_i$
3. *CONN* is a logic connection $CONN = \{CN_1, CN_2, \ldots, CN_m\}$, CN_i is a transition with channels, $CN_i \in \left(\bigcup_{i=1}^{n} T_i \bigcup \bigcup_{j=1}^{p} T'_j \right)$

2.2 Modeling Agent Systems

A mobile agent support system (we also call it as agent system or host system if it does not cause confusion) provides services and managements for agents. Agent systems are pre-installed in hosts and each one has its own location property that identifies it in the network. Agents and agent systems use location information to locate a specific system within a mobile agent system. Agent systems may have different capacities, but we only model the most general behaviors of agent systems. An agent system can create mobile agents according to user requirements, send agents to other agent systems, receive agents from other agent systems and provide reasonable services for agents, communicate with agents or other agent systems through message passing, monitor agent running and may force them to move out. Agent systems receive messages from other systems or agents, and these messages could be data, commands, or agents. Tokens in system nets mainly have two types: one is message, which is not associated with any agent net; another is agents, whose nets are wrapped as tokens with identifiers. That means an agent token always include two attributes: one is agent identifier *AI*, and another is agent net *MN*, and they consist of a structured data (*AI, MN*). We use *MN* to represent $MN(P, T, F, \Sigma, L, \varphi, T_{in}, T_{out}, M_0, C, W)$ for simplicity if no confusion caused.

When an agent system receives a message (token) from other systems or agents, it processes this token according to its type. If the token is a message, the system processes this message. Then the processed message is sent to other agents or hosts if the message is regular data (the token has the form: *<dt, dl, da, sl, sa, MSG, ¢, message>*), and an agent is move out if the message is a command to move out an agent *<dt, dl, da, sl, sa, MSG, MOV, ¢>*). If the token is an agent, the agent is started and its state is recovered from the stop point when it moves out. Only after an agent starts its task, it can receive incoming or send out outgoing messages. In the model, incoming agents stay in the particular place P_a of its host system net to run their tasks until they are moved out from the host net. Agents only can run their tasks in this place. Agent systems can send messages to any agents within it, and agents can send messages to their host systems. In order to facilitate the communication between agent systems, an agent system also can send messages to other agent systems directly. However, if an agent system needs to send messages to other agents that are in other agent systems, it should know the location of the destination agent system and identifier of the agent. The messages are sent to the destination system firstly, and then the destination system forwards the messages to the receiver agent. The communications between agent systems and agents are through channels. We do not consider group or broadcast communication in this model. Here is the agent system model:

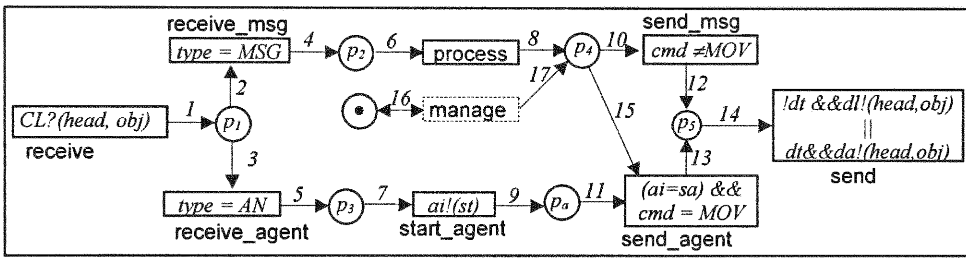

Figure 4.4 An agent system model (host net)

Table 4.1, Legend of Figure 4.4

place/transition/inscription	Description
p_a	The place mobile agent stay in, <*dt, dl, da, sl, sa, AN, ai, MN*>
p_1/ p_5	The incoming/outgoing messages from/to channels, <*dt, dl, da, sl, sa, type, command, message*>
p_2, p_4/ p_3	Messages, messages <*dt, dl, da, sl, sa, MSG, command, message*>/Agents <*dt, dl, da, sl, sa, AN, ai, MN*>
receive	Input transition, get tokens from *CL* channel
send	Output transition, send messages to *dl* or *da* channels according *dt*
receive_msg	Receive messages (incoming tokens are messages)
receive_agent	Receive agents (incoming tokens are agents)
process	Agent system processes the received messages
start_agent	Start the received agent
send_msg	Send messages to other agent systems or agents within this system
send_agent	Send agents to other systems
manage	System monitors agents
1, 14, 16, 17	<*dt, CL, da, sl, sa, type, command, message*>
2, 4, 6, 8, 10, 12	<*dt, CL, da, sl, sa, MSG, command, message*>
3, 5, 7, 9	<*dt, CL, ¢, sl, sa, AN, ai, MN*>
11	<*¢, ¢, ¢, ¢, ¢, AN, ai, MN*>
13	<*false, nl, ¢, CL, sa, AN, ai, MN*>
15	<*false, dl, da, CL, sa, MSG, MOV, nl*>

In Figure 4.4, *dl* represents destination location of this message or mobile agent, *da* is the destination agent, which receives the message, if this token is an agent, then *da* is empty ¢. *sl* is the location of the source agent system. *sa* is the source agent which sends the message. *nl* is the next destination of agent *sa.* For simplicity, we use *obj* and *head* to represent complex structure data, where *obj* ∈ {<*CMD, DATA*>}, *CMD*∈{*MOV, STOP, AI*} is a management command or the identifier of the sending agent, *DATA* represents the message contents, it could be regular data or the agent net. *head* = <*dt, dl, da, sl, sa, type*>, *type* ∈ {*MSG, AN*}. *CL?, dl!, da!, ai!* is channel name, and *CL* is constant. <*ai, MN*> where *ai* is the agent identifier, and *MN* is its net. *dt* is destination type, means the destination of a message is a host or an agent, it is a boolean variable, *dt* is *false* when destination is a host, *dt* is *true* when the destination is agents.

In Figure 4.4, we model the basic functions of a mobile agent system. When channel *CL* has data with structure <*head, obj*> available, the input transition *receive* is ready to fire. The data or token is moved from channel *CL* to place p_1. If the token is a message, the token is sent to place p_2. Then if this message is data for an agent, then the data is sent to place p_4 through transition *process*, and the data is delivered through transition *send_msg* to place p_5, and input transition *send* put the data into channel *da*, and then the agent *da* will get the data. If the message is a command to move an agent, such as the message is <*dt, CL, ¢, CL, sa, MSG, MOV, nl*>, the agent *sa is* move out from place p_a to destination host *nl* through transition *send_agent* to place p_5. Then the output transition *send* put that agent into output channel *dl,* and agent system *nl* will receive this agent. If input transition gets the token from input channel *CL* is an agent *da (*suppose it is *DA)*, the agent *DA* is sent to place p_3 from p_1, and then transition *start_agent* sends current location information *CL* to the channel *DA*. When the agent gets *CL* from its input channel *DA*, it starts its task and is ready to receiving other messages from its input transition. Then agent *DA* is sent to place p_a. *CL* is the system location and it represents a channel as well. The destination variable *dl* and *da* are channel variables and they are assigned real values at run time. Each agent system has a unique channel *CL*, and it only get incoming messages from channel *CL*. The input tokens of an output transition have the destination information, which decides the values of the output channel names. Since the input tokens of output transitions change dynamically, so that the channel values also change dynamically, and then the output transition dynamically connect with the input transition of the destination. The communication between output transitions and input transitions are through channels, which link two communication transitions at run time. Both the input transition and the output transition for a communication fire at the same time and disconnect the communication link as soon as they fired.

2.3 Modeling Mobile Agents

A mobile agent is an independent program with its own task on behalf of users. We view an agent as an encapsulated entity consisting of interfaces, behaviors, and states. It is an interactive object capable of receiving message from and sending messages to other objects. In the meantime, it has its own states, and methods to process messages as well as to change the state. Agent systems can send messages to agents, and these messages could be regular data for processing or commands for managing agent's resources. Before an agent moves out, it stops running and wraps up its state, and then it is delivered to the destination agent system. The destination agent system starts the execution from the stop point when the agent moves out. Agent itineraries are assigned when agent are created and updated at run time. Each agent may have its own knowledge base, which decides agent decisions during its life span. Agents are different since they have from simple to complex functionalities (we use a dashed box to represent the running task). Although we only model the general behaviors of mobile agents, other specific tasks or services can be modeled as modules to plug in this model. An agent identifier represents its agent net, and each agent includes a location property, which is the location of the agent system where this agent stays. When an agent moves from one agent system to another, its location is updated to the destination location. We define each type of agents using a CPrT net. The agent creation is the initialization of a pre-defined CPrT net, and an agent token is an instance of a type of agent nets. The instance has its own identifier and the structure of the agent net based on the pre-defined CPrT net. These instances are independent even though they may cooperate with each other. For example, if agent a_1 clones another agent a_2, then a_1 and a_2 are two separated agents. The change of a_1's state does not affect the state of a_2 except when they cooperate with each other explicitly.

An agent net has input and output interfaces for receiving incoming messages or sending outgoing messages. Each agent net has a unique identifier *DA*, which is assigned when the agent is created. Its location is the location of its host net. Each agent gets incoming messages from channels *DA* and only from this channel except the start information from its host net. The input transitions also make sure messages from its current host system or agents who are in the same host system. Since agents may move to different hosts at any time, it is difficult to send messages to other agents who are not in the same space. This limitation on sending message makes sense and many mobile agent systems include this limitation. Agents cannot receive or send messages until they are started and they are in particular place p_a of the host net except receiving the starting command from host nets. The input transition for receiving start signal in agent net is inactive until agent is stop. There are two types of incoming messages, one is regular data, and another is command. Agents process messages and commands at run time. Agents have their own tasks and they may send some requisitions at any time. Before an agent moves out, it stops its execution and saves its current state. According to the location of destination system, the agent updates its current location information before it moves out. If the agent wants to move out, it sends a message *<false, CL, ¢, CL, DA, MSG, STOP, ¢>* from transition *run* to place p_2. Then the transition *stop_agent* is enabled and fired, it gets the next destination for this agent. After transition *run* fires, the transition *stop* fires since *nl* is different current location *cl*, which stops running tasks of the agent. The message *<false, CL, ¢, CL, DA, MSG, MOV, nl>* is sent to place

p_2, and transition *send_msg* will deliver the message to current host net. As soon as current host net accepts the message, it is sent to place p_4 of the host net in Figure 4.4. Then the corresponding agent *DA* is move out from place p_a in the host net *CL* to next host net *nl*. When the destination system accepts it, the system sends a message to start the agent. The agent starts its running and recovers its state. Its location property is changed, and its input transition for receiving messages is ready to receive messages. We use channels *DA* to get the starting information from its current host net. When the agent arrives at p_3 in host net, the host net sends a message to the agent through channel *DA*. Then the agent starts and its receiving interface is enabled. Each agent has its own knowledge base *kb*, which decides the behaviors of the agent. An agent sends messages to other objects through its output transition/interface, which has output channels to connect with other objects. The following diagram is the mobile agent CPrT model:

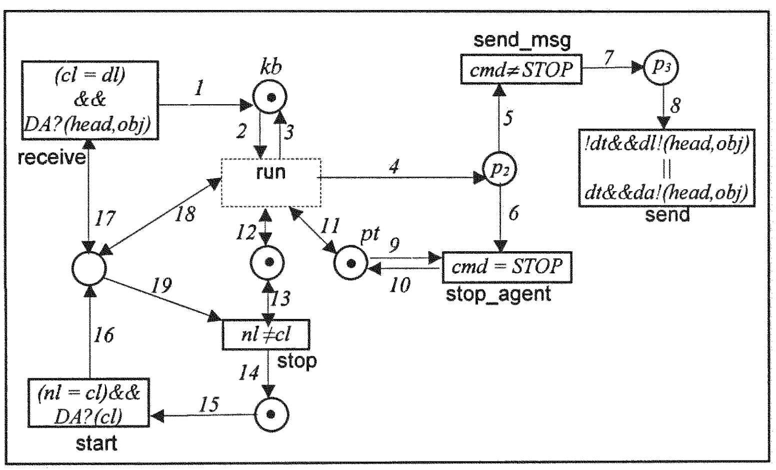

Figure 4.5 A mobile agent model (agent net)

Table 4.2 Legend of Figure 4.5

place/transition/inscription	Description
receive	Get incoming message from *DA* channel
send	Send outgoing messages to *dl* or *da* channels
send_msg	This agent sends messages to current system
stop_agent	This agent sends a request to move out, its itinerary is updated, and its state is wrapped. The moving out agent command is sent to current agent system when transition run fires, and the execution of agent is stop.
run	This agent runs its tasks, gets its itinerary.
stop	Stop the running of this agent
start	Start running of this agent
kb	Knowledge base or received data for this agent
pt	The itinerary of this agent
1, 2, 3	*<head, obj>*, where *head* =*<dt, dl, da, sl, sa, MSG>*, and *obj* = *<cmd, msg>, cmd* ∈ *{MOV, STOP}*
6	*<dt, dl, da, sl, sa, MSG, STOP, msg>*
4, 5, 7, 8	*<dt, dl, da, sl, sa, MSG, cmd, msg>*
9, 10	*<cur>, <cur = (cur + 1)/N>, N* is the number of total destinations
12, 13, 14, 15	*<nl>*
16, 17, 18, 19	*<cl>*
11	*<nl>, nl = cur(pt), pt* is the itinerary, and *next* is the reference pointing to current location.

In Figure 4.5, where *dl* represents the destination location of this message or mobile agent, *da* is the destination agent, which will receive the message. *sl* is the location of the source agent system, where the message is sent. *sa* is the source agent sending the message. *nl* is the next destination of current agent. *obj* ∈ {*<cmd, msg>*}, *cmd* ∈{*MOV, STOP*} is the management command, *msg* represents the message content. *head = <dt, dl, da, sl, sa, type>, type ∈ {MSG, AN},* we use *head* here for simplicity. *dl!, da!, DA!* is channel name, and *DA* is constant representing current agent. *cl* is variable for location of agent system which starts this agent. The destination variable *dl* and *da* are variables and they are assigned with real values at run time. *dt* is a boolean value referring to whether the destination of a message is system or agent. *dt* is *false* when destination is host nets, and *dt* is *true* when the destination is agents.

In Figure 4.5, we model the basic functions of mobile agents. When channel *DA* has data with structure *<head, obj>* available, the input transition *receive* is ready to fire. The data is move from channel *CL* to place p_1. Then the agent processes the data using transition *process* with statements from *run*, and put results into place p_2. If the token is a command such as stopping agent <*dt, dl, DA, sl, sa, MSG, STOP, ¢*>, transition *stop_agent* is going to active transition *update* to get next destination location. Then transition *run* sends out token to activate *stop* transition, which stops current agent execution. The current destination location *cl* is updated to the value of next destination. A message <*false, cl, ¢, cl, DA, MSG, MOV, nl*> to move out current agent is sent to place p_2, and then transition *send_msg* sends the message to current agent system, which sends out this agent. In our models, stopping an agent means to move it out, however, its destination depending on its itinerary. An agent support system does not send a command to move out an agent directly. *DA* is the agent identifier, and it represents a channel for this agent as well. The input transition has a guard condition to guarantee messages from current host system or agents that are in the current host system. The destination variable *dl* and *da* are channel variables and they are assigned with real values at run time so that the agent can communicate with different host systems and different agents.

2.4 Dynamic Connection

In order to capture the social ability of agents and to bridge the gap between agents and systems, we enable agents to connect with host systems dynamically. Representing such connections is a challenge for the Petri net formalism because it is statically defined, whereas the number of mobile agents changes over time [XYD03]. It is impractical for each system to provide separate ports for connection with each agent. Instead, we introduce channels to connect agents with their host systems at run time to facilitate the dynamic configuration of mobile agent systems. Here we show how agents dynamically connect to host systems and migrate among them. The following diagram is a snapshot of a mobile agent system with logic connection using channels. We only define their interface transitions and their parameters are simplified for this specific case:

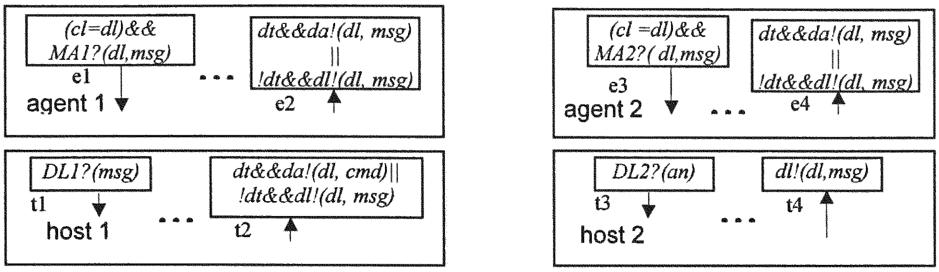

Figure 4.6 A logic connection model

In Figure 4.6, agent 1, agent 2, host 1 and host 2 have channel MA_1, MA_2, DL_1, and DL_2 to receive incoming messages, and all of them are unique in the network. Moreover, the channel variable *cl* is assigned with values at run time. When these nets send out outgoing messages, the input tokens and the inscriptions on the input arcs of output transitions decide the value of *dt* and the output channels *dl* and *da*. The *cl* has the value same as the current location of the agent. Based on above description, we discuss several general communication scenarios:

1. An agent sends messages to its host system. If agent 1 is in host 1 and it sends a message to host 1. Then in agent 1: $dl = DL_1$, $da = ¢$, $dt = true$, and *msg* is the message. The output transition e_2 has the output channel with values as: $DL_1!(true, DL_1, ¢, DL_1, MA_1, MSG, ¢, msg)$. When e_2 fires, DL_1 channel has that message, then t_1 gets data *msg* and other information from channel DL_1. When t_1 fires, it sends the data to output places according to PrT firing rules and it removes the token from channel DL_1. The agent system in host 1 starts to process the message.

2. A host system sends messages to an agent that stays within it. If agent 1 is in host 1, and host net 1 sends a message to the agent. Then in host net 1: $dl = DL_1$, $da = MA_1$, $dt = true$, and *msg* is the message content. The output channel of transition t_2 has values as: $MA_1!(false, DL_1, MA_1, DL_1, ¢, MSG, msg)$. When t_2 fires, MA_1 channel has the message. When e_1 fires, it sends the data to output places according PrT firing rules and it removes that message from channel MA_1. Agent 1 can start to process message *msg*.

3. An agent moves from one host system to another. If agent 2 is in host net 1, and it wants go to host 2. First, agent 2 sends this requisition $<false, DL_1, ¢, DL_1, MA_2, MSG, MOV, DL_2>$ to the

agent system in host 1, and then host 1 sends agent 2 to host 2. Transition e_4 assigns values: $dl = DL_1$, $sa = MA_2$, $dt = false$, and $CMD = MOV$, $msg = DL_2$, and the output channel of transition t_2 has values as: $DL_1!(false, DL_1, MA_2, DL_1, MA_2, MSG, MOV, DL_2)$. Before agent 2 moves out, it stops its running, updates its location with value $DL2$. Then system 1 sends agent MA_2 to channel DL_2 with values: $DL_2!(true, DL_2, ¢, DL_1, MA_2, AN, MA_2, MN_2)$. When t_3 fires, it sends the MA_2 to output places according PrT firing rules and it removes message in channel DL_2. Then system in host 2 sends starting messages through channel MA_2 with its location value DL_2 to agent 2 so that it starts. The agent is sent to place p_a in the system net of DL_2, and then agent MA_2 starts to work.

4. An agent sends messages to another agent that is in the same agent system. If agent 1 and agent 2 both are in system 1, and agent 1 sends a message to agent 2, then in agent 1: $dl = DL_1$, $da = MA_2$, $dt = true$, and msg is the message content. The output transition t_2 has the input channel with values as: $MA_2!(true, DL_1, MA_2, DL_1, MA_1, MSG, ¢, msg)$. When t_2 fires, MA_2 channel has message msg, then e_3 gets the message from channel MA_2. When e_3 fires, it sends the data to output places according to PrT firing rules and it removes messages from channel MA_2. Agent 2 starts to process message msg.

5. One host system sends messages to another host system. If agent system in host 2 sends a message msg to agent system in host 1, then in system 2 output channel: $dl = DL_1$, $dt = false$, and msg is the message content. The output channel of transition t_4 has values: $DL_1!(fasle, DL_1, ¢, DL_2, ¢, MSG, ¢, msg)$. When t_4 fires, DA_1 channel gets the message, then t_1 gets the same message from channel DA_1. When t_1 fires, it sends the message to output places according PrT firing rules and it removes the message from channel DA_1. System 1 starts to process message msg.

If agent 1 is in host system 1, agent 2 in host system 2, agent 1 wants sending messages to agent 2, then agent 1 only can send messages to host system 2, and host system 2 forwards the messages to agent 2, or agent 1 and agent 2 must move to the same place to complete the communication.

3 Related Works

In our previous model (LAM) for mobile agent systems, we used connectors to facilitate communication and interaction between different nets [XYD03], while our current method uses channels to provide functionalities for communication and interaction among different nets.

In LAM, a mobile agent system consists of a finite set of components and a finite set of connectors. Each component includes an identifier for the component, a system net, and an internal connector. The connectors (also called external connectors) interconnect host systems (or agent system in LAM). Each host system has one input predicate receiving messages from connectors and one output predicate sending messages to connectors. In addition, arcs of connector nets are supposed to be properly labeled so that a migrating agent is always transferred to a single destination since they do not consider agent cloning or broadcast agent transferring. External output place of one system net may connect to all other components. Here is an example of connector models:

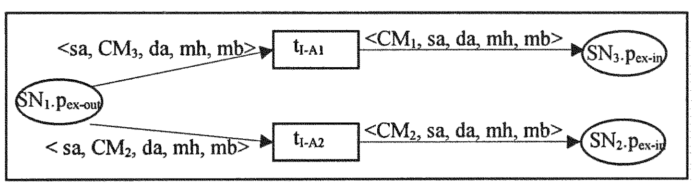

Figure 4.7 A connector net of LAM

Essentially, connectors are pre-defined. When any agent system joins in or leaves, this connector has to be re-defined. The channel method has not this problem since channels are dynamically created according to run time situations (token or message values). We can look all agent systems are connected with inactive channels, and channels are activated according to the system or agent outputs. It naturally captures the essentially dynamic property of mobile agent systems.

Each LAM component has one internal connector to connect the system net with all mobile agents residing in the current component. Such an internal connector depends on the internal interfaces of environment, the running agents, and their interfaces. The following diagram shows an example of internal connector for two agents:

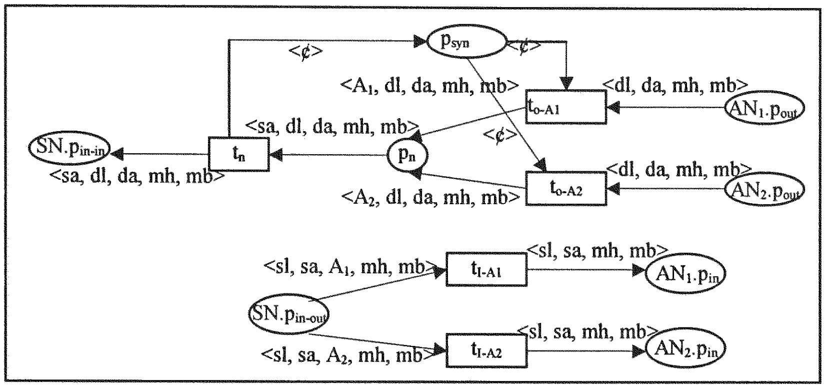

Figure 4.8 An internal connector net of LAM

It is difficult to capture the dynamic connection between agents and their systems because the internal connector structure has to be changed with new agents arriving or leaving. Our current method avoids this problem since it uses channels to connect agents with their systems. Channels for connecting agents with systems are dynamically chosen according to PrT firing rules. It smoothly integrates dynamic channels with static PrT nets. When we model a mobile agent system, the system architectural model changes its structure with run time activities of agents and agent systems. The channel method is more flexible to model mobile systems such as mobile agent systems and other systems with code mobility. It provides a powerful mechanism to model synchronous communication between distributed objects. In order to demonstrate its capacities, we will use channels to model other paradigms of code mobility in the following section.

4 Modeling the Code Mobility

In addition to the mobile agent, the remote evaluation and the code on demand are other paradigms of code mobility. The classification is based on the location of components before and after the execution of the service, the computational component that is responsible for execution of codes, and the location where the computation of services actually take place [FPV98]. We discuss the client server paradigm even there is no code mobility because all these paradigms of code mobility are special forms of client server paradigm. We already discussed the modeling and analyzing mobile agents using CPrT nets. In this section, we study the other three paradigms to illustrate the express capacities of CPrT nets on code mobility.

Client server paradigm is a widely used classical distributed style. It has two components: one is the server that provides a set of services, and another is the client that requests services from servers. The server has programs and resources for all services in its site, and the client sends the specifications of requests to servers. When one server receives a request from clients, it starts the corresponding service and returns results to the client. If the server has not the service, the request fails. In this paradigm, there is no code migration among components. Remote evaluation paradigm has two components: one component (we call it client) requests services, and another one (we call it server) executes the service and delivers results back to the client. Clients have programs of services, but they have not required resources, which are located on servers. A client sends the program of a service to a server that has required resources, and the server executes the program and sends results back to the client. Code on demand paradigm is the reverse style of remote evaluation. It consists of two components: one is the client that requests services, and another one is the server that provides programs of services to clients. Clients have resources for requested services on their sites, but they have not corresponding programs. Servers have required programs of services, so that servers send programs to clients according to demands from clients. Then clients run those programs and get required services.

4.1 Modeling Client-Server

For client server paradigm, all resources and programs are on the server side. The CPrT net model has two parts: one is for the client, and another is for server. The communication between client and server is through channels. The server model is a two-layer CPrT net, the system net models the environment, and the token nets represent the programs of services. Token nets are instantiated as instances or object nets in system nets. In other word, object nets are packed as tokens in the system net. Each token net represents one service, and the service can serve different clients at the same time. If one service is serving several clients, there are several tokens in the system net, which are instances of the token net representing the service. The following diagram shows the model:

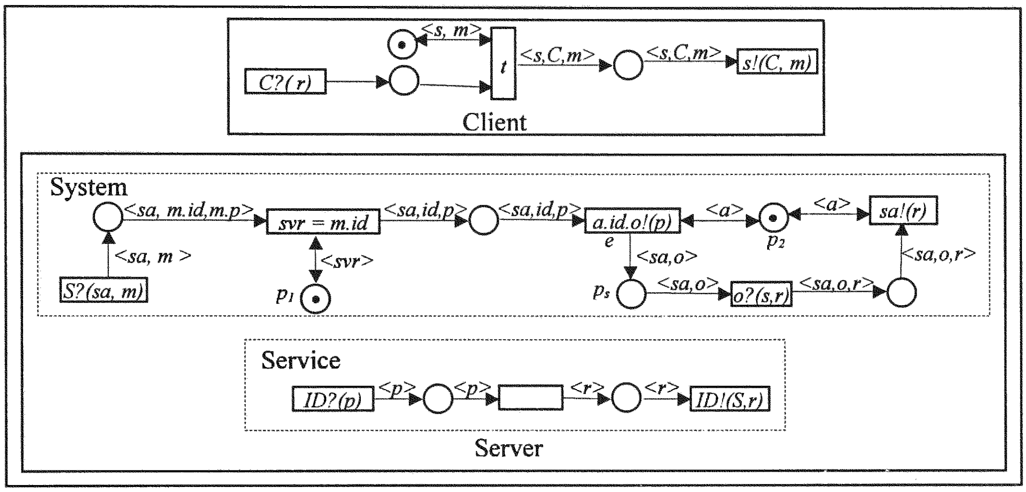

Figure 4.9 A CPrT net of Client-Server systems

In the client net, C is the channel representing this client to receive messages from servers. The parameter of channel C is the result r, which is a structured variable. The transition t sends a request to a server for some services. The request includes the server name s, the client name C, and structured data m. The m includes the service name ID and parameters p, which is also a structured parameter for a set of simple parameters.

In the system net, S is a channel representing the server to receive messages from clients. S has two parameters: sa for the source client, and m for the service requisition. The place p_1 has the list of services that the server provides. For simplicity, we do not consider the reject conditions. The place p_2 is the configuror, which is a set of all active object nets in this system. The a is a structured parameter, which has the form $\{ID_1(obj_1, obj_2, \ldots, obj_n), ID_2(obj_1, obj_2, \ldots, obj_m), \ldots, ID_p(obj_1, obj_2, \ldots, obj_k)\}$. ID_i is the type of the service, and obj_i is the instance of this type of service, and service instances are staying in p_s during its life span. The transition e generates an instance (start the service) of a type of service, and then forwards the requisition data to the object. The unique instance identifier in the server is assigned according to a in p_2, and e updates a when it generates an object. When result r is ready and sent back to client, the corresponding object becomes inactive, and it is removed from configuor p_2.

In the service net, channel *ID* is a dummy channel name representing the service. *ID* is instantiated as a real value (the instance identifier) when an object is instantiated from this service and move out from p_s. The result of the computation is sent back to the system when the service completes.

4.2 Modeling Remote Evaluation

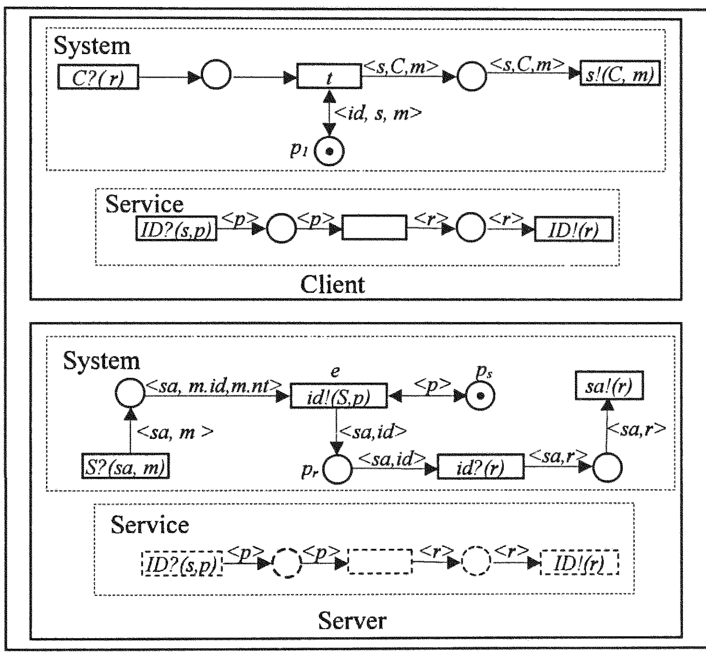

Figure 4.10 A CPrT net of remote evaluation systems

Remote evaluation is a special style of mobile agent paradigm. However, a remote evaluation system does not move resource with its mobile programs. In other words, it does not support strong mobility. Programs, which migrate to remote sites, have not itineraries since remote evaluation only works for one hop situation. The model of remote evaluation has two parts: one is the client that sends programs to remote sites for evaluation, and another is the server that receives and runs theprograms from clients. The client model has token nets, which are moved to remote server sides and instantiated as object nets. The results are delivered back to clients after the evaluation finishes. Figure 4.10 shows the model of remove evaluation.

In the client net, it has a system net with its token nets that represent services. The channel *C* represents the client to receive the evaluation results *r* from servers. The transition *t* sends a program *id* with its net *nt* to server *s*. The program identifier *id* and its net *nt* are combined as a structured data *m*. The p_1 has the information of available programs in the client. The client sends a copy of the token net to the server side, and this token net is instantiated as an object net and only one object in the server side.

In the service net, channel *ID* represents the type of a service and it severs as an object name as well since only one object of the service is in each server at any time. The result *r* of the computation on resource *p* is sent back to the system when the service completes. The reason we make the input channel has different parameters with output channel because we need make sure that the object sends result to its system not itself.

In the system net of the server side, *S* is a channel representing the server to receive messages from clients. *S* has two parameters: *sa* for the source client, and *m* for the service requisition. The place p_s has all required resources for services. The received service (*m.id, m.nt*) is instantiated as object with the service name. The transition *e* generates the instance (start the service) of the type of service (place p_r is the instantiated service working place), and then forwards the requisition data to the object. When result *r* is ready and sent back to the client, the corresponding object becomes inactive.

4.3 Modeling Code-on-Demand

Code-on-Demand is the reverse style of the remote evaluation since it migrates code from servers to clients, while remote evaluation moves code from clients to servers. Code-on-Demand is also the reverse style of mobile agent paradigm since each client requests programs to move from server sites to clients so that codes are passively moved to clients. However, in the mobile agent paradigm, agents actively move from clients to remote server. Same as remote evaluation, contents of migration between clients and servers are programs without states. The model of code-on-Demand has two parts: one is the client that requests programs from remote sites to the client side, and another is the server that provides required programs to clients. The server model has the system net and token nets, which represent services. Programs or codes are moved to the client and instantiated as object nets. Programs run in the client and provide results directly to the client. Figure 4.11 diagram shows the model of code-on-demand paradigm.

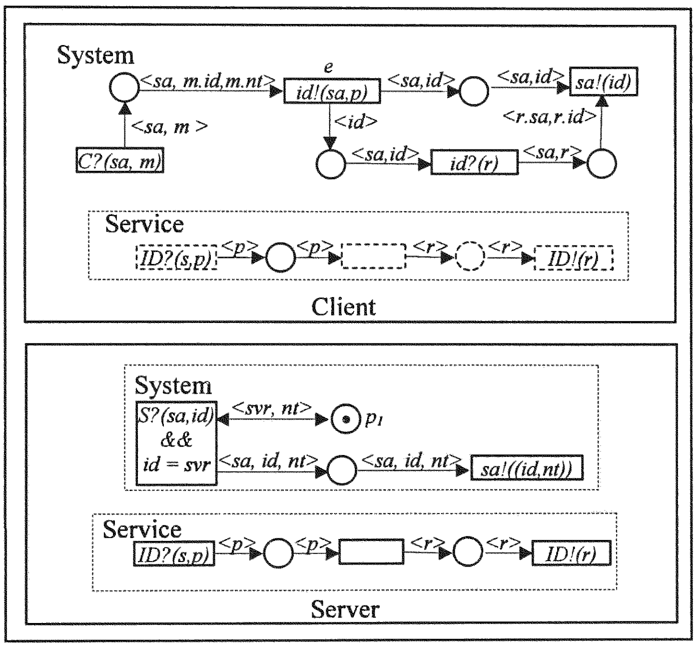

Figure 4.11 A CPrT net of Code-on-Demand systems

In the client net, the channel *C* represents the client to receive the requested code *m* (i.e. *(id, nt)*) from servers. The transition *e* starts the code or sends a request to server. When the client starts the code, it creates an object of the service net and the object name is same as the service type. The reason is there is only one object of a particular type service in one client at any time. The request for code also may happen during running of some services. The dashed service net is the object net from remote server and it is instantiated in the client.

In the system net of the server side, S is the channel representing the server to receive requests from clients. S has two parameters: sa is for the source client, and id is for the specification of a request code. The place p_1 has all required codes in the server. If the server has the request program (codes), the program identifier and its net are packed as a token and sent to the client.

In the service net, channel ID represents the type of a service and it serves as an object identifier as well since only one object of the service is in each client at any time. The result r of the computation with resource p is sent back to the system of the client when the service completes.

From modeling of these three different paradigms of code mobility, and the modeling and analysis of mobile agent systems, we demonstrate that two-layer CPrT nets are a powerful tool to formally model and analyze systems with code mobility. The two-layer method naturally captures the structure of systems with code mobility, and the channel mechanism smoothly integrates models in different layer. The high-level Petri nets provide a tool for modeling systems with higher abstraction and more compact models.

5 Concluding Remarks

In this chapter, we use two-layer CPrT nets to model the software architecture of mobile agent systems and models of other systems with code mobility. From successful modeling and analysis of these systems, we conclude that CPrT net is a power formal tool to modeling mobile computing systems. These models demonstrate some advantages: 1. The two-layer modeling paradigm smoothly transform physical models of mobile computing systems to their formal architecture models. Since agents and agent support systems are related independent systems, this method brings us convenience to focus on a particular sub-system without involving the complexity of its environments at each time. Moreover, it is helpful to analyze these models since we can analyze models on a particular level with abstraction of another level. 2. We chose dynamic channels to facilitate the synchronous communication between different nets. It naturally captures the dynamic configuration property of mobile computing systems. Communication objects change their communication topologies with the changes of their environments at run time since channel values are dynamically assigned during execution. The dynamic channel provides a mechanism to construct easy-to-understand and compact models, since each dynamic channel is a finite set of static channels.

CHAPTER V

Analyzing Software Architecture of Mobile Agent Systems

1 Introduction

An important goal of formal modeling is to facilitate the system simulation as well as the specification analysis. We already defined the software architecture of mobile agent systems using CPrT nets. In order to analyze the software architecture, we formally define its semantics, which cannot be interpreted by the semantics of CPrT nets because of the dynamic property of the architecture. A mobile agent system consists of several agent support systems and a group of agents. The structures of these support systems and agents are statically defined, but the software architecture has to be reconfigured with mobile agents moving in or moving out. In the system level, we treat agents as tokens. We can analyze agent systems like regular CPrT nets if we look at agents as regular data. Then the analysis is addressed on one level CPrT nets. However, we have to consider agent nets within system nets if we need to analyze the interaction between agents and support systems and the dynamic configuration of the software architecture with migration of agents. Then we have to connect agent nets with system nets as a whole net when agents move in, and disconnect agent nets from system nets when agents move out. In the agent level, there is no token net so that we can analyze agent nets as regular CPrT nets if we consider agents with predefined interfaces that represent agent support systems. Then the analysis is addressed on one level CPrT nets. We consider the cooperation between two agents who are within the same space. We analyze the cooperation of two agents as a special case of interaction since communication between agents is through the agent support system where both agents are staying in. The analysis is based on the connected net of the two agent nets and the simplified host net. We chose a hierarchical analysis method to analyze the software architecture of mobile agent systems. We analyze the software architecture on system level, component level and interaction level. The system level analysis focuses on system properties such as mobility, safety or liveness of a system. The component level analysis focuses on individual component properties such as properties of an agent net or a host net, but without considering other components. The interaction analysis focuses on dynamic configuration of system architecture, interaction and communication between agents and systems.

2 Hierarchical Analysis Method

System level analysis treats agent nets as regular data or tokens within its places. Since all host nets are statically determined, their composition is to connect them together based on their channels. If one property we analyze involves only a single host system, we only need to analyze the single CPrT net of the host system, and if that property involves several host systems, we have to analyze the CPrT net consisting of those host nets. The analysis is directly addressed on these CPrT nets connecting with channels, which form a whole net for the mobile agent system. Component level analysis is used to analyze individual agent models or agent support system models. When we analyze an agent net, its environment or agent support system is abstracted as several interfaces. These interfaces send values to or get values from agent channels. The simplest way to construct these interfaces simulates the interactions between agents and environments. If a transition in agent nets has one input channel, which has a partner output channel in its system net, then the interface is a subnet to setting values for the input channels when necessary. If a transition in agent nets has one output channel, which has a partner input channel in its system net, then the interface is to remove values from the output channels when necessary. In order to analyze complex properties, it is necessary to construct complex interfaces, which are beyond the scope of this paper. The most complex interface is the original host net, which cannot be reduced. In that case, we have to analyze the whole net that consists of agent nets and their host nets.

When we need to analyze the interaction between agents and their host nets, we have to use this method, which is called interaction analysis or composition level analysis. Interaction analysis is used to analyze properties involving agent nets and the system net or the cooperation among different agents. Agent tokens in system nets are unfolded into agent nets, and system nets are connected with those agent nets based on channels. When we analyze the cooperation between two agents, the analysis is based on the connected net that consists of the two agent nets with states. These agent nets maybe instantiated from the same predefined agent net, but they have different states (markings). Because of the migration of agents, the model structures are dynamically configured at run time. The migration of agents is determined with the itineraries of agents, and these itineraries maybe updated at run time. In the next section, we will discuss the method to analyze the dynamic configuration of architectural models.

2.1 Component Level Analysis

Component level analysis is used to analyze individual component properties such as properties of agents or agent support systems. In the architectural model of mobile agent systems, there are only two components: one is host nets, and another is agent nets. Since these individual components are part of a whole system, we have to transform each individual component as an independent model for analysis purpose.

2.1.1 Analyzing Host Nets

Each host net has at least one transition with input channels, and one transition with output channels. When we analyze host net, the functionalities of channels are reduced to receive tokens and send tokens. It is not necessary to consider on the dynamic communication since the agent is already set in a particular environment. We transform input channels as input places of the transitions that have these input channels. We transform output channels as output places of the transitions that have these output channels. Then the component model is merged with its interfaces to from an independent model. The following diagram shows the transformation. The top part of the diagram is the transformation for input channels, and the bottom part shows the transformation for output channels.

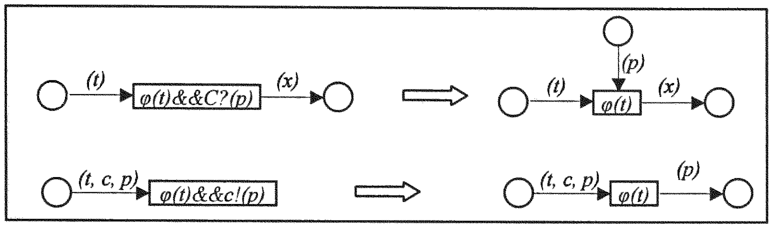

Figure 5.1 The transformation of channel expressions

If one transition has more than one channel, we have to transform these channels into regular transitions according to their relationships and PrT net rules. We restrict that there is no transition with two different type channels, so that there are only two different combinations of channel expressions: one is the *AND* relation between two channels, and another is the *OR* relation between two channels. If two input channels have an *AND* relation in a transition, then these two channels are directly transformed into two input places of the transition. If two input channels have a *OR* relation in a transition, then these two channels are transformed into two concurrent input places with two concurrent transitions, and then these two concurrent transitions output to a place, which is the input place of the transition with these channels. If two output channels have a *AND* relation in a transition, then these two channels are directly transformed into two output places of the transition. If two output channels have *OR* relation in a transition, then these two channels are transformed into two concurrent output places with two concurrent transitions, and then these two concurrent transitions has one same input place, which is the output place of the transition with these channels. The following diagram shows these transformations. In this diagram, the top two nets show the transformation of input channels, and the bottom two nets show the transformation of output channels.

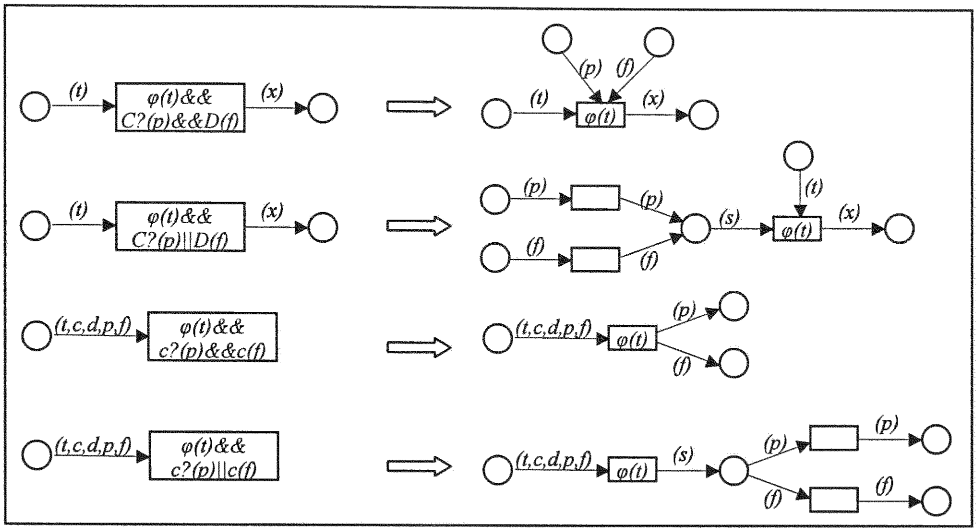

Figure 5.2 The transformation of complex channel expressions

After we transform these channels into regular transitions, we can analyze host nets based on regular PrT nets. The analysis methods include such as the reachability tree technique, the temporal logic proof technique, the structural induction technique, and model checking [HD02] [HYS03].

2.1.2 Analyzing Agent Nets

Agent net analysis is to analyze agent properties, which do not involve interactions with hosts or other agents. The easiest way is to transform channels of an agent net into ordinary transitions. Then the analysis is based on the ordinary PrT nets. However, many agent properties involve other agents and their environments. In that case, we have to abstract host nets into simpler nets, and then analysis will based on the simpler host nets and agent nets. Since agents only interact with the host where they are staying in, we can reduce the host net into a transition with one input place and one output place. If we analyze properties of one agent net, we can fuse places of the simpler host net with the places transformed from channels, and the analysis is based on this transformed net. However, if we consider properties of multiple agents, we have to use the method for interaction analysis, which we will discuss in the following section. The following diagram shows the basic idea of the transformation. In the left part of the arrow, the top net represents a host net, and the bottom net is an agent net. In the right side of the arrow, it is a transformed agent net with its an abstracted host net.

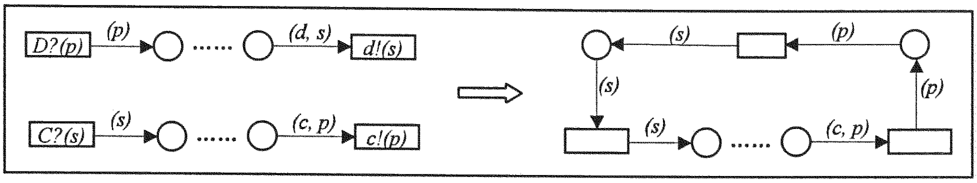

Figure 5.3 A transformation of an interaction

2.2 Composition Level Analysis

Composition level analysis is used to analyze some properties that involve communication and interaction among different nets. An ideal approach is to carry out the composition-level analysis compositionally. In this approach, each subnet such as host net or agent net is analyzed individually, and then the interested properties are synthesized based on properties of individual nets. Despite some existing results on compositional verification techniques in Petri nets, their general use is not yet ready [HD02]. Therefore, even if we can analyze some properties of host nets, agent nets and system nets (considering agent nets as regular tokens) individually, we still need to analyze some properties based on the composing model that consists of different nets and interfaces for their environments. We transform the CPrT nets into PrT nets, and then we use existing analysis techniques of PrT nets to analyze the models.

2.3 System Level Analysis

In system level, we treat agents as regular tokens. The analysis is addressed on two kinds of models: one is the host net, and another is the system net that consists of all host nets. When we analyze properties that only involve particular host net, we can transform the host net into an independent CPrT net for analysis. If we analyze system properties such as mobility, we have to connect host nets to form a logical whole net for analysis. Since each host net has input transitions and output transitions that may involve the communication between agents, we have to transform these transitions into transitions that have not channels involving agents. Especially for analyzing a single host net, we need to transform these transitions into regular transitions since we do not considering the communication between different nets.

From system view, a system net of a mobile agent system consists of all host nets. These host nets communicate with each other through channels. We analyze system properties over the system net, which is a whole net consisting of all host nets. Each host net has at least one transition with input channels, and one transition with output channels. These channels are used not only for communication between host nets, but also for communication between host nets and agent nets. Since we look at agents as regular tokens, the functions of channels, which are used for communication between host nets and agent nets, are reduced to receive tokens or send out tokens. We use the same method that we discussed in above section to transform channels into ordinary transitions. The channel variable for communication between agent nets and agent nets are different to the channel variable for communication between agent nets and host nets. Therefore, we keep channels that for the communication between host nets and transform other channels into regular PrT nets. Then the analysis is based the transformed system net. Then we can analyze the model using the reachability tree analysis technique or other analysis method such as model checking technique.

3 Dynamic Configuration

The dynamic configuration of the software architecture of mobile agent systems is reflected by connecting or disconnecting agent nets with host nets. At the system level, we do not consider dynamic configuration since agents are treated as regular tokens. For simplicity reason, we do not consider reconfiguration of agent systems. In other words, the location of each agent system is fixed, and all agent systems are predefined and ready to accept all agents. The interoperability property is not within the scope of this dissertation. If we need to consider the configuration of agent systems such as some agent systems may join in or leave during run time, we have to change host nets with an additional boolean variable on the inscriptions of channel transitions to indicate whether the system is active or not. This variable is part of a guard, which will disable these transitions with channels when it is *false* so that the agent system net cannot get or send messages to other agent systems or agents. From the system point of view, this agent system is disabled. The dynamic configuration does not exist at agent level either, since we only consider agent nets or the whole net of two connected agent nets, and these net structures are statically determined using one level CPrT nets.

At the interaction level, dynamic configuration brings us much more complexity on analysis. How to analyze dynamic configuration architecture is an interesting and important topic. The architecture of mobile agent systems consists of a group of host nets and a group of agent nets. These host nets and agent nets are statically determined, so the architecture is the static definition of the system. Since each agent net can be instantiated as several instances or objects with different states at run time, the system architecture is dynamically configured at run time when the system net connects with different object nets. Agent nets communicate with other objects through channels, and each object has one unique input interface to receive messages from other objects so that it guarantees messages reach correct destinations. We call this channel as agent channel, and its value is a dummy constant when it is defined in templates of agent nets. However, this dummy value is replaced by a unique real value same as the instance identifier when an instance is instantiated from one template. In order to analyze the dynamic reconfiguration of system architecture, we introduce the configuror, which is used to remember current active agents in each host net. Based on system configuror, we reconstruct and analyze the snapshot of the software architecture.

3.1 System Configuror

There are only finite numbers of object nets in a system net at any time, so we can transform the dynamic view into a static view to study interaction properties. The key issue is how we can transform a dynamic view into a static view at run time. We introduce a configuror concept to define the configuration of agent nets (the instances of agent nets) with their system nets. We do not add any configuror to CPrT nets, but it is used for describing the system configuration when we analyze the models. The configuror is responsible for achieving the dynamic reconfiguration of the system architecture. Each system net has a configuror, which consists of agent instance identifiers, agent types (agent nets) and agent itineraries. When an agent is created, it is assigned an itinerary that decides the visiting path of this agent. Based on the knowledge or itineraries of agents, agents may dynamically update their itineraries at run time. The configuror of system architecture is the combination of all configurors of host nets.

Definition 3.1.1 (Configuror) The configuror of each system net is a list $CON = \{c_1, c_2, \ldots c_n\}$, where $c_i = (AN_i{\cdot}ID, AN_i{\cdot}TYPE, AN_i{\cdot}KB)$, $1 \leq i \leq n$. The n is the number of agent instances in the system net. AN_i is the agent instance in the host net, and $AN_i{\cdot}ID$ is the instance identifier of AN_i, $AN_i{\cdot}TYPE$ is the instance type (the name of the template net of AN_i), and $AN_i{\cdot}KB$ is the instance itinerary of AN_i.

When a host net receives an agent, the agent location is updated as the location of the host system. Then it is put into the special place p_a in the host net, which is the only place the agent can update its states except when the host system starts it. When an agent moves out from the host, it updates its location according to its itinerary and stops its execution until the destination host accepts it. An agent system can generate agents or instances of agent nets (we call instances of agent nets as object nets) according to existing agent types (templates of agent nets), but each object net has its unique identifier and itinerary. When a host net receives or generates an object net, the configuror adds the object into its list, and it removes the object from its list when an object leaves the host net. This configuror is easily constructed from agents within p_a. The static view of interaction between host nets and object nets is a net composing from the host net with a group of object nets within the host net. The following diagram shows the basic idea to analyze the dynamic configuration of host nets.

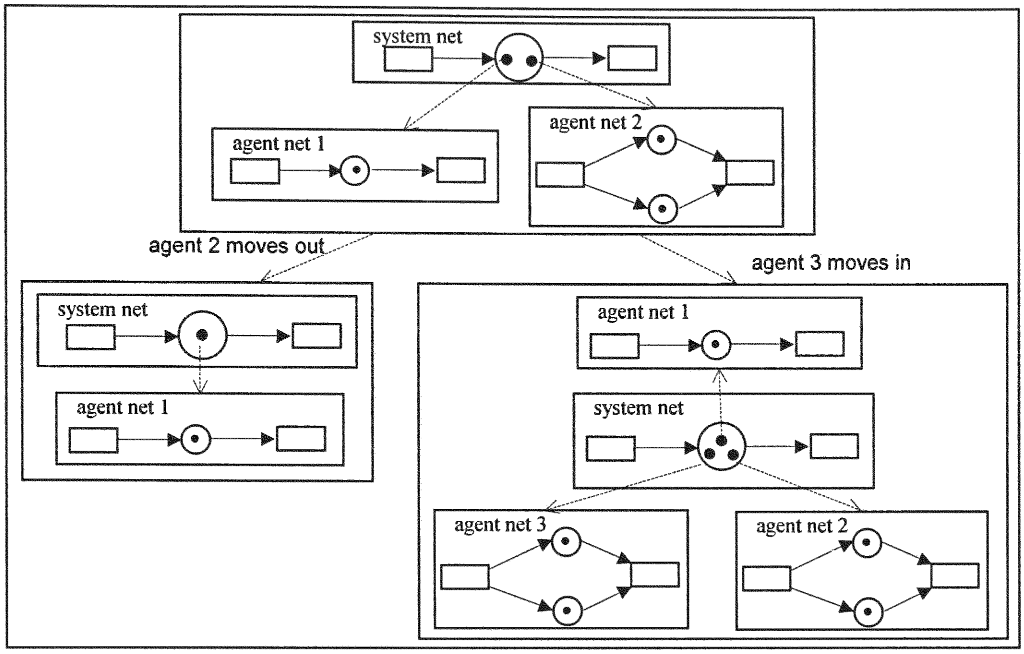

Figure 5.4 A dynamic configuration

In Figure 5.4, the system net or host net has two agents within its place p_a, so that its configuror includes these two agents information, which can be used to construct the static view of the host model with one host net and two agent nets. Then one agent moves out, the configuror removes the agent (agent net 1) from its list, so that the static view of the current host model is the host net and on agent net. When the host net receives an agent (agent net 3), the configuror adds that agent information into its list, so that the static view of the current host model is the host net and three agent nets. Based on static views and configurors, we can analyze the dynamic reconfiguration of the software architecture of mobile agent systems.

3.2 Analyzing Dynamic Configuration

We analyze the interaction between a system net and its agent nets through transforming the dynamic view into static view according to the configuror. We unfold all object tokens (the instances of agent nets) from the system net into agent nets with states, and these nets consist of a logical whole net even if they may not be connected with arcs, but they are logically connected with channels. The analysis is based on these nets and configuors. The occurrence rules for this interaction view are the same as the semantics and analysis on two-layer CPrT nets. The marking of the whole net is the combination of the marking of each net. When an agent moves out from the host net, the configuror removes that object from its list and the corresponding object net is removed from the interaction view or the whole net. When an agent moves in the system, the configuror adds that object from its list and the corresponding object net is added into the interaction view or the whole net.

Definition 3.1.2 (Interaction view): An *interaction view* of a system net is a tuple *IV = (SN, AN, CON)*, where:

1. *SN is a system net,* $SN = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$.
2. *AN is a finite set of object nets,* $AN = \{AN_1, AN_2, \ldots, AN_n\}$, $AN_i = (P_i, T_i, F_i, \Sigma_i, L_i, \varphi_i, M_{i0}, C_i, W_i)$, $1 \leq i \leq n$, $AN \subseteq \Sigma$.
3. *CON is the configuror of SN.*

The dynamic configuration of host net is reflected on the migration of agent nets. Here is the definition of dynamic configuration of a system net, but it can be extended to architecture level since it is the combination of a group of host nets.

Definition 3.1.3 (Dynamic configuration): The dynamic configuration of a system net is reflected on the dynamic changes of configuror of the host net. An interaction view of the host net is *IV = (SN, AN, CON)*, where:

1. When an agent AN_k moves in to *SN*, $AN_k = (P_k, T_k, F_k, \Sigma_k, L_k, \varphi_k, M_{k0}, C_k, W_k)$, then $AN_k \in P$, $CON = CON \cup \{c_k\}$, and $c_k = (AN_k{\cdot}ID, AN_k{\cdot}TYPE, AN_k{\cdot}KB)$.
2. When an agent AN_k moves out from *SN*, $AN_k = (P_k, T_k, F_k, \Sigma_k, L_k, \varphi_k, M_{k0}, C_k, W_k)$, then $AN_k \notin P$, $CON = CON \setminus \{c_k\}$, and $c_k = (AN_k{\cdot}ID, AN_k{\cdot}TYPE, AN_k{\cdot}KB)$.

The occurrence rules and communication between object nets and the system net follow the definitions in CPrT nets.

4 Strong Mobility

Mobility is the most important property of mobile agent systems. The strong mobility means agents can move from sources to destinations along with their states. When an agent moves out from one space, it stops its execution and save its state. As soon as the agent arrives at the destination, it resumes its execution and recovers its state from the stopped point. In order to discuss strong mobility, first we need to clarify the location concept in mobile agent systems. Each agent system has a unique location attribute, and agents within it share the location information. Each host system is fixed with its location, but agents move from hosts to hosts. Therefore, the location information of agents changes with their migration. However, each agent only has one unique location at any time, which means each agent only exists in one agent support system at any time.

Suppose ψ is the finite set of all host nets for a mobile agent system model Π, ω is the finite set of all agent types, and $\Pi = (\psi, \omega)$. δ is the finite set of all object nets or instances of agent type ω in ψ at the analysis time. We use $SN_1 \neq SN_2$ to denote that host net SN_1 and SN_2 are in different locations, and $p_a \in SN$ is the place where agent net can running their tasks.

Theorem 4.1 (The unique of agent location): Given an agent $\alpha \in \delta$, if $\alpha \in SN_1 \cdot p_a$, the agent system $SN_1 \in \psi$, if there is any other agent systems $SN_2 \in \psi$, and $SN_1 \neq SN_2$, then $\alpha \notin SN_2 \cdot p_a$.

Proof: There are only two ways to get agents in a host system, one is the host system generates an agent (creates an instance net from agent type net), another way is to receive agents from other agents.

1. The agent system (host system) SN_1 creates an agent α: before SN_1 creates α, $\alpha \notin \delta$, that means to any agent system $SN \in \psi$, $\alpha \notin SN \cdot p_a$. After SN_1 creates α, α is unique to any other agents in δ because each new created agent has an unique identifier, and $\alpha \in SN_1 \cdot p_a$. So α exists and only exists on SN_1 after it is created, and $\delta = \delta \cup \{\alpha\}$.

2. Agent α in the agent system SN_1 comes from other agent systems: suppose $\alpha \in SN_1{\cdot}p_a$, and $\alpha \in SN_2{\cdot}p_a$, $SN_1 \in \psi$, $SN_2 \in \psi$, and $SN_1 \neq SN_2$, if α is generated in agent system *SN1* or *SN2*, then based on discussion on (1), $\alpha \in SN_1{\cdot}p_a$, or $\alpha \in SN_2{\cdot}p_a$, but it is impossible $\alpha \in SN_1{\cdot}p_a$ and $\alpha \in SN_2{\cdot}p_a$. The only possible is that α was generated from the third agent system SN_3, $SN_3 \in \psi$, $SN_1 \neq SN_2 \neq SN_3$, and then it moves to SN_1 and SN_2. Now we prove this situation is impossible. M_i is the marking of system net SN_i, and $SN_s \in \psi$, and $SN_f \in \psi$, and suppose the migration path ζ of α from SN_3 to SN_1 is:

$$M_{30}[t_{31}\theta_{31} > M_{31}[t_{32}\theta_{32} > \ldots >[t_{3n}\theta_{3n} >M_{3n}$$

$$M_{s0}[t_{s1}\theta_{s1} > M_{s1}[t_{s2}\theta_{s2} > \ldots >[t_{sm}\theta_{sm} >M_{sm}$$

……

$$M_{10}[t_{11}\theta_{11} > M_{11}[t_{12}\theta_{12} > \ldots >[t_{1k}\theta_{1k} >M_{1k}$$

The migration path ζ' of α from SN_3 to SN_2 is:

$$M_{30}[t_{31}\theta_{31} > M_{31}[t_{32}\theta_{32} > \ldots >[t_{3g}\theta_{3g} >M_{3g}$$

$$M_{f0}[t_{f1}\theta_{f1} > M_{f}[t_{f2}\theta_{f2} > \ldots >[t_{fr}\theta_{fr} >M_{fr}$$

……

$$M_{20}[t_{21}\theta_{21} > M_{21}[t_{22}\theta_{22} > \ldots >[t_{2t}\theta_{2t} >M_{2t}$$

Based on above discussion, we know α only can be transform through path ζ or ζ', but not be created. If α goes from $M_{30}[t_{31}\theta_{31} > M_{31}[t_{32}\theta_{32} > \ldots >[t_{3n}\theta_{3n} >M_{3n}$ to $M_{s0}[t_{s1}\theta_{s1} > M_{s1}[t_{s2}\theta_{s2} > \ldots >[t_{sm}\theta_{sm} >M_{sm}$, then α can not go from $M_{30}[t_{31}\theta_{31} > M_{31}[t_{32}\theta_{32} > \ldots >[t_{3n}\theta_{3n} >M_{3n}$ to $M_{f0}[t_{f1}\theta_{f1} > M_{f1}[t_{f2}\theta_{f2} > \ldots >[t_{fr}\theta_{fr} >M_{fr}$, at the same time if $SN_s \neq SN_f$. Then we reach that α does not exist in SN_1 or SN_2 at the same time. If $SN_s = SN_f$, we can prove α does not exist to the next two agent systems of SN_s or SN_f at the same time since α does not exist in SN_1 or SN_2 at the same time.

Based on (1) and (2), we reach the conclusion.

If a mobile agent system supports strong mobility, the mobile agent execution is suspended and its state is saved when a mobile agent moves out. The agent is inactive until it arrives at its destination. Before the agent is put into the place p_a, its location is updated to the location of the destination agent system, and then the state of agent is resumed from the exact point when it leaves from the source host system. The following proposition expresses this definition.

Suppose ψ is the finite set of all agent host nets for a mobile agent system model Π, ω is the finite set of all agent types, and $\Pi = (\psi, \omega)$. δ is the finite set of all object nets or instances of agent type ω in ψ at the analysis time. We use $SN_1 \neq SN_2$ to denote that SN_1 and SN_2 are in different locations. M_{1i} is the marking of SN_1, and M_{2i} is the marking of SN_2, M_{ai} is the marking of α. Then we have the following proposition for strong mobility.

Theorem 4.2 (Strong Mobility): Given an agent $\alpha \in \delta$, if $\alpha \in SN_1{\cdot}p_a$, the agent system $SN_1 \in \psi$, and another agent systems $SN_2 \in \psi$, $SN_1 \neq SN_2$, there is a firing sequence (the sequence of moving out the agent, and the sequence of receiving the agent) ζ for the agent α

$$(M_{10}, M_{a0})[(t_{11}\theta_{11}, t_{a1}\theta_{a1}) > (M_{11}, M_{a1})\ [(t_{12}\theta_{12}, t_{a2}\theta_{a2}) > \ldots >[(t_{1k}\theta_{1k}, t_{ak}\theta_{ak}) >(M_{1k}, M_{ak})$$

$$(M_{20}, M_{ak+1})[(t_{21}\theta_{21}, t_{ak+2}\theta_{ak+2}) > (M_{21}, M_{ak+2})[(t_{22}\theta_{22}, t_{ak+3}\theta_{ak+3}) > \ldots >[(t_{2t-1}\theta_{2t-1}, t_{at-1}\theta_{at-1}) >(M_{2t-1}, M_{at-1}) > [(t_{2t}\theta_{2t}, t_{at}\theta_{at}) >(M_{2t}, M_{at})$$

Where $t_{1k}{}^{\bullet} = {}^{\bullet}t_{21}$ and the channel $c \in \left(\theta_{1k}/\varphi(t_{1k})\right) \cap \left(\theta_{21}/\varphi(t_{21})\right)$, the type of one parameters of channel c is the agent type of α, $\alpha \in c.P$. $\alpha \in M_{10}(SN_1.p_a)$, $\alpha \notin M_{11}(SN_1.p_a)$, and $\alpha \in M_{2t}(SN_2.p_a)$. Then $M_{a1} = M_{ak} = M_{ak+1} = M_{ak+2} = M_{at-1}$, *and* $\forall p \in \alpha{\cdot}P$, $M_{a0}(p) = M_{at}(p)$ except $p_l \in \alpha{\cdot}P$, and $M_{a0}(p_l) \neq M_{ak+1}(p_l)$, p_l is the predicate representing the agent location.

Proof: The proof is straightforward, so we only give the basic idea. When α moves out from $SN_1{\cdot}p_w$, all transitions in α are inactive (we put a guard variable to each transition when we design agent nets) until it arrives at $SN_2.p_w$. So we get $M_{a1} = M_{ak} = M_{ak+1} = M_{ak+2} = M_{at-1}$; $M_{a0}(p_l)$ is the location of SN_1, and $M_{at}(p_l)$ is the location of SN_2, but t_{at} only update the tokens in p_l, so we get $\forall p \in \alpha \cdot P$, $M_{a0}(p) = M_{at}(p)$ except p_l. □

5 Cooperation

The cooperation means the interaction and communication between several agents within the same space (agent system). We do not discuss the cooperation between agents who are not within the same host system because they cannot communicate with each other directly. The cooperation between agents is through channels in agent nets. We demonstrate the cooperation based on one-to-one communication styles since we do not consider group or broadcast communication in CPrT nets. Since agents are staying within agent systems, we have to discuss the cooperation within the context of agent systems. The agent system communicates or interacts with agents through channels, so that we can abstract agent support system as some interfaces from channels. If the analysis focuses on the interaction between the system and agents, we have to analyze the interaction view between agent system and the cooperation agents, which we discussed in above section. When we analyze the cooperation between two agents, we abstract the host net as an interface to forward or receive data to or from other agents. The analysis is based on the whole net (from logical point of view) composing from the two agent nets and the interface for the host net. The following diagram shows the basic idea.

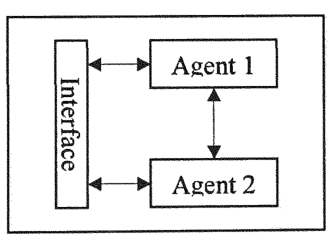

Figure 5.5 A cooperation between agents

Suppose ψ is the finite set of all agent host nets for a mobile agent system model Π, ω is the finite set of all agent types, and $\Pi = (\psi, \omega)$. δ is the finite set of all object nets or instances of agent type ω in ψ at the analysis time. $M_{\alpha i}$, $M_{\beta i}$ is the marking of α and β respectively. Then we have the following definition on cooperation between agents.

Definition 5.1.1(Cooperation between agents): There are two agents $\alpha, \beta \in \delta$, an host system $SN \in \psi$, and $\alpha, \beta \in SN.p_a$. There is at least one transition $t_1 \in \alpha \cdot T$, and one transition $t_2 \in \beta \cdot T$, they have paired channels with matched parameters. There is a θ so that $(M_{\alpha 0}, M_{\beta 0})[t_1\theta, t_2\theta > (M_{\alpha 1}, M_{\beta 1})$, and $\rho \in M_{\alpha 0}({}^{\bullet}t_1)$, $\rho \in M_{\beta 1}(t_2{}^{\bullet})$.

6 Model Checking Software Architecture

In chapter 2, we already discussed the basic ideas of model checking CE nets models and PrT net models. In this section, we discuss the method to model checking CPrT net models of mobile agent systems using model checking tool SPIN.

Model checker SPIN only can directly check models with finite states. In order to check an infinite state system using SPIN, we have to reduce the system model into a model with finite states. In many cases, some properties still hold after reducing a model with infinite states to one with finite states. Therefore, we can reduce models with infinite states into models with finite states as long as the reduction does not affect those properties we need to verify. We model the software architecture of mobile agent systems using CPrT nets, but the input programs of SPIN are defined using Promela. We need to translate CPrT net models into equivalent Promela programs. System properties are defined as correctness claims in Promela programs. Some important system properties are specified as *never* claims, which are translated from LTL formulas. In order to verify different properties and reduce the complexity of the verification, we use SPIN to check the models based hierarchical analysis method. We firstly check individual host nets and agent nets without considering unfold agent nets, and then verify some system properties based on reduced model of the whole system net. We provide a general procedure and rules for model checking the software architecture of mobile agent systems using SPIN.

Step 1: Transform models (individual nets and reduced system nets)

If we check one net such as a host net or an agent net, we need to transform the CPrT net into a PrT net. Since each individual net may include input channels, which are synchronized with some corresponding output channels in other nets, and these input channels are guard conditions to enable those transitions, it is easier to assign the initial marking if we transform these channels into subnets. Moreover, it is convenient to verify results if we transform those output channels into subnets. When we translate each individual model into a Promela program, the transformed PrT nets are useful to verify the consistence between the CPrT model and its Promela program. Based on verified results from individual models, we may transform a system model into a simpler model using the method we already talked in previous parts. In this case, we have to keep channels, and then we transform these channels into Promela programs directly.

Step 2: Reduce states (from infinite state model to finite state model)

First we need to restrict each place p in a model is k-bounded (i.e. $M(p) \le k$, $M_0[>M)$, where k is a constant. Then we define each variable type as enumerable type with finite number of elements. The k is predefined according to system requirements.

Step 3: Specify properties

After we defined system behavior models B using CPrT nets, we specify interested system property specifications S using LTL. The verification procedure is to verify property specification S over behavior models B, i.e, $B \models S$.

Step 4: Translate a CPrT net or a PrT net model into a Promela program

In order to verify a CPrT or PrT model, we have to translate the net model into Promela program. The following steps define the translation rules.

1. Program structure, Each individual net in a system model is translated into a process in Promela program. Each program includes type definitions, global variable declarations, processes, *init* process, and a *never* claim. The type definition defines place and variable types. The global variable declaration defines global variables. Each process defines all transition relations in one net. The *init* process is used to assign initial markings and other initial values. The *never* claim defines system properties.

2. Defining state variables, Each place in a net is translated into a variable in Promela program. The value range of each variable represents the possible markings of the place. Therefore, the number of possible values of a variable is the number of possible markings of the corresponding place. If a place p is k-bounded and $|\varphi(p)|$ is the number of possible values of a token in p, then the number of possible markings of place p is $\sum_{i=0}^{|\varphi(p)|} k^i$. Therefore, the declaration statement for place p has the form:

$$p : 0..\sum_{i=0}^{|\varphi(p)|} k^i - 1 \qquad (6.1.1)$$

Thus, we treat a predicate symbol as a set of proposition symbols. This can be done when each p is bounded and $|\varphi(p)|$ is finite [HYS03].

3. Defining initial state, Initialize each variable with a value, which is corresponding to the initial marking of the place in the behavior model. Initial variables are assigned values through *init* process in the Promela program. Each net has a corresponding process with its variables as input parameters, and *init* process invokes this process with real values (initial values). If there is more than one process in the program, and each one is corresponding to one subnet, then *init* process invokes these processes with their initial values as parallel running processes. Model checker SPIN guarantees the running fairness of these processes.

4. Defining transition relations, There are two types of transitions in CPrT nets, one is transitions without channels, and another one is transitions with channels. We discuss these two transitions separately.

4.1 Defining transitions without channels, Each transition in a net is defined as an atomic statement within a process, which represents the subnet, in a SPIN program. Each atomic statement defines the firing rules of the transition. The atomic statement consists of a series of *case* statements, and each one is corresponding to one possible input of the transition. The body of each *case* statement explicitly defines the translation from input to output. The number of *case* statement for each transition is the permutation number of input variables in inscription expressions of the input arcs of the transition. There are many case statements in some atomic statement, if it has many possible input values. Fortunately, each case statement should be very simple, and we can use tools to help generate all case statements if we could not translate one net to a program automatically. If there is more than one net in a model, and if the CPrT nets are transformed into PrT nets, then shared places between different nets are defined as global variables, so that it is easy to communicate between different nets and synchronization between communication transitions are guaranteed with additional global boolean variables.

4.2 Defining transitions with channels, If there is more than one net in a model, and channels in CPrT nets are not transformed into regular subnets, we have to translate these transitions with channels using different methods to translate them into Promela programs. We separate channel expressions from the transition inscription expressions. Then each channel is declared as a global variable, which has a type with all possible values (finite number of values). In CPrT nets, channel variables share variable names with their input inscription. However, we have to declare different variables for each channel. The variable number is the number of possible values of the channel variable in the net. All of these variables have same type, which is same to the type of input or output parameters of the channels. After we separate channel expressions from transition expression, we define the transition relations. For output channels, when the transition fires, some channel variable is assigned with value according to the output of the transition. Such as one channel has three possible values, P_1, P_2 and P_3, if the output value, which is assigned to the channel in the net, is P_2, then value of P_2 is updated with the values of the output parameters, but P_1, and P_3 do not change. For input channels, according to input tokens and inscriptions on input arcs, we chose one channel variable as part of input conditions of the transition. For example, the channel has three possible values, P_1, P_2 and P_3, and if input tokens realize current input channel is P_2, then P_2 is chosen as part of input conditions of the transition. When the input transition fires, value of P_2 is updated. It is simpler to translate channels into Promela programs directly than to transform CPrT nets into PrT nets, and then to the translate PrT nets into Promela programs.

5. Defining properties to be verified, We define *never* claim in Promela program to verify system properties, which are defined using LTL. *Never* claims can be automatically generated from LTL formulae using SPIN tools. We also can define *accept-state* labels in Promela programs to check properties such as reachability. There are some other Promela constructs such as *basic assertion*, *end-sate labels*, *progress-state labels*, and *trace assertions*. We can use them to define different interested properties in Promela programs.

7 Concluding Remarks

In this chapter, we propose a systematic analysis method to analyze software architecture of mobile agent systems. Because of the dynamic reconfiguration property of the software architecture, we introduce a configuror to record current active agents in each agent support system. The configuror can be generated from agents in the particular place p_a of each host net. Then we analyze the software architecture through transforming the dynamic architecture into a static one based on system configuror. We formally analyze the mobility property of mobile agent systems based on location changes of the migrating agent, and the strong mobility based on the firing sequence of the migration does not change the state of the migrating agent. Because of the relative independency of each individual net in the software architecture and the hierarchical structure of mobile agent systems, we introduce a hierarchical analysis method to analyze the software architecture of mobile agent systems. Based on different properties, we choose component level analysis, system level analysis and composition level analysis method to analyze these properties. Finally, we introduce a method to analyze the CPrT models of mobile agent systems using model checking tool SPIN. We can use model checking technique to analyze much more complex and larger system when it is integrated with hierarchical analysis method.

CHAPTER VI

A Medical Information Processing System

1 Background

Medical analysis data are precious resource to researches such as the disease discovery and the pharmacy development. How to process these data to get useful information is a challenging and tedious work. These data have some properties: 1. Data volume is huge. Each database may have millions of samples, and each sample may have hundreds of data. In order to find useful information, it is necessary to search many databases. 2. Different types of samples and different companies may have different databases, which are distributed on different sites. These databases may have different database management systems and data might be encoded with different security systems. 3. Data on different sites may have different formats even for the same parameter of same type of samples. These differences prevent data interoperability between different systems. 4. Legacy data may only supported by legacy systems, which are not available by some users. Because of these difficulties, retrieving information from medical data normally is restricted to limited data such as data with specific formats or in particular databases. If we can overcome these limitations, we may find much useful information that will be helpful to medical researches. Most medical analysis data is separated from users and providers. Normally, users are research groups, hospitals, or pharmacy companies, while providers are analysis laboratories. Each user may have very limited data on local sites, but big laboratories have much more and complete analysis data from different samples. In addition, In order to find useful information, it is important to process data from different groups. This separation is a natural client server structure, but servers only can provide data and some resources. Servers may provide a few basic services for database access and processing, and clients retrieve databases using these limited functions or services to find useful information. However, those services from servers are far less than requirements of medical information processing. Moreover, because requirements from clients are different, it is impossible to provide near to complete data processing services from servers. The traditional way is users copy data from providers, and then they analyze data in local sites. However, this method brings many problems: 1. Data volume too huge that to copy all data. Even the volume of data is still huge after filtering some unnecessary data. 2. Each server has different system to manage its data even encrypt data for security protections, so that the data is nonsense without support systems. However, users cannot provide all of these systems in their local sites. Even it is impossible to maintain these systems for some clients. Such as many hospitals chose Intersystems Caché as their DBMSs (Database Management System), but many research groups chose Microsoft SQL server as their DBMSs. 3. The increasing of data in server sides is very fast, but clients cannot update their data at real time since clients have to copy data to local sites. 4. Clients have to pay fee based on the volume of data they get. Clients have to reduce their data usage as less as possible, but it may lose much precious information. Because of these difficulties, some clients may provide service programs to servers, and servers install these services in their servers. Even this method is better than the traditional method, but it is difficult to server providers. They have to maintain and update these services for each customer, and they have to provide huge servers for customers. The computation model is one client and one server structure, so clients have to do the tedious work to move intermediate results from one server to others if the task involves data on different servers.

In order to overcome some of these problems, we design a medical information processing system based on mobile agents. The general idea is that services are designed as agents that compute from sever to server for medical information processing. These servers install agent support systems in their agent system servers, which are separated from database servers, and the communication between agents and databases is through some access interfaces. Clients send agents, which are complex data processing programs, to servers and run within agent support systems. Agents move to different servers with their intermediate results according to their predefined itineraries and run time results. The final results are delivered back to agent users when agents finish their tasks. Agent systems are provided by clients and follow requirements from servers. These agent systems provide basic functionalities to support running of agents, and they implement requirements such as the safety, security, performance requirements from servers. Therefore, different clients can share these agent systems. Clients or users implement agents, which include their codes and knowledge bases to process medical data. Each agent moves back to its users when it finishes its tasks, so it is not necessary to maintain many agents or services on servers. The advantages of this computation model are obvious: 1. Since agents are moved to server sides, they can use data locally and access much more data. It is not necessary to copy and maintain data in local sites any more, and it is not necessary to maintain support systems for original data from servers. 2. Since agents are moved to server sides, agents can work offline on servers. It saves bandwidth comparing to traditional client server systems. 3. Clients create agents according their applications, so that they can maximize the usages of data. 4. Since agents are running at server sides, they can access the latest data at run time. 5. Servers do not need to provide any application related services so that the burden of servers is much reduced. 6. Because agents can move to multiple servers with intermediate results automatically, this method provides an automatic processing flow for information processing. Users on clients do not need to process the intermediate results any more, so their works are sending out agents and waiting for results.

2 System Structure

In order to support the medical information processing system based on mobile agents, the following requirements are required. 1. Network connections. All hosts in a computation group should be connected via networks so that they can reach each other. The computation group is a set of hosts where agents will visit. 2. Distributed environments. A distributed environment provides basic services for distributed computing. It supports such as network file systems, naming services, interoperability between different systems and security functionalities. The most popular and commercial distributed environment is CORBA (Common Object Request Broker Architecture) [OMG02]. These services are required to agent systems and agent computing. If some servers have different distributed environments, the cooperation between agents within these servers should be limited. 3. Interface specifications. Agents need to access databases, they cannot access databases directly but through interfaces for security reasons, which limit and control the access from agents to databases. These interfaces should have a common specification so that users easily design their agents to access different databases. The interface specification should conform to some existing interface specifications such as ODBC (Open Database Connectivity) [Gei95]. The following diagram shows the general system structure of a medical data processing system based on mobile agents.

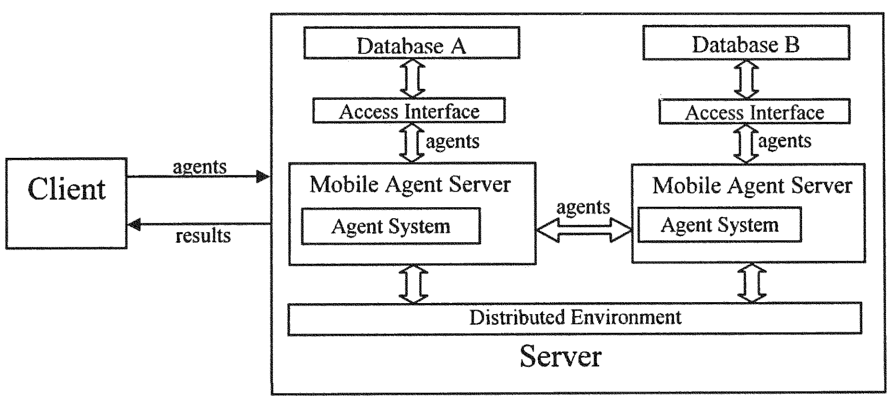

Figure 6.1 A framework of a mobile agent System

2.1 A Mobile Agent System

From logical view, the system consists of clients and servers, and they are connected with networks. The client sends agents representing users to complete some tasks in server sides, and servers provide basic environments and resources supporting the execution of agents. It is similar to the traditional three-tier client server structure, but it has three important differences from the client-server paradigm. 1. The business logic in the server side is supplied by clients, and they are dynamically deployed and configured in the server side by mobile agents. 2. As soon as one agent arrives at its destination server, the connection between the client and the server is not necessary until someone requests re-connection. In client server systems, the connection is required during execution of the required task. 3. Mobile agents can migrate from one server to others during their life spans according to their pre-defined itineraries and intermediate results. However, the configuration between client and sever are statically defined in client server systems.

In each server side, the databases and the agent support system are two different systems. The server system deploys and configures a particular agent system on its host before it can accept the type of agents, monitor and support the execution of agents. Databases in servers keep all data and provide basic database services. Agents can read data from databases through their interfaces, and they cannot write or modify any data on any database. We are going to introduce different concepts of the system: clients, agents, servers, agent support systems, databases and results.

Clients: Clients provides agent support systems or host systems to servers, and these host systems are installed and configured in servers to provide execution environments for agents. In addition to the basic functionalities of agent systems, they are vary on services for different requirements on safety, security and performance. Each client has a particular agent system that can create agents and submit them to servers.

Agents: Each agent includes at least the following five parts: 1. A program to access and process data from databases. 2. An itinerary for its visiting path and a strategy to update its itinerary according to its intermediate results. 3. Authority from its users. 4. A resource requirement specification. 5. A log file for important events.

Servers: The server includes two separate servers. One provides basic database services and other functionalities for the server side, and another is the agent system, which provides basic services and functions to run agents.

Agent Support Systems: Clients create these agent support systems, which should satisfy requirements from servers. Agent systems are deployed and configured at server sides, and these servers are connected via networks. Several agent systems may consist of a region, which may have a common distributed environment such as CORBA to support distributed computation.

Databases: Databases include database management systems, database applications and data storages. There are different databases for different purposes in different sites. However, these databases provide a unified interface such as interfaces based on ODBC to clients. For security and safety purpose, database servers do not provide any service for agents directly. They provide interfaces to bridge the gap between agents and databases. Agents must use suitable interfaces to access databases. The server sides create interfaces publish their specifications for clients.

Results: The results include intermediate and final results. Agents process intermediate results on agent support systems, and they may bring these results to other hosts for continue works. The results are delivered back to clients when the agent finishes its task. Agents cannot write any data to databases in server sides except the databases on agent systems. If they need to manipulate some data from databases, they have to copy these data to its agent system, and then the processing is based on these as-is data since we do not consider data synchronization.

3 An Application

In this section, we introduce an application of the medical information processing system. It is a data processing system for information on human blood cells. We call this system as CIP (Clinical Information Processing) system. Research groups use CIP to retrieve and process medical data from two different databases: one is for the cytometry analysis data, and another is for the hematology analysis data. For some researches, they have to process data from one database, and then process data from another database based on previous results.

Cytometry analysis [Sha03] is the analysis on blood cells for specific diseases such as HIV. Each sample has dozens of parameters such as count of red blood cells (RBC), count of white blood cells (WBC), count of platelets, volume of monocytes, lymphocytes, neutrophis, and eosinophils, and many other different white blood cells. Based on these parameters, it also calculates hundreds of combination data such as one dimension logarithm data (linear), two dimension data, three dimension data etc.. All cytometry data are saved in the specific database for cytometry data. Hematology analysis [RB02] is the analysis on blood cells for routine analysis. Each sample has several dozens of parameters such as complete blood cell count (CBC), count of platelets, and hundreds of combination or processed data based on well-know algorithms. All hematology data are saved in the specific databases for hematology data.

Clients are research groups who require huge samples for researches, and they try different algorithms and protocols to process these data. Each analysis software or agent includes program, algorithms and protocols. The algorithms are rules or procedures to process data, and the protocols decide the selection and combination data with different parameters. In other words, protocols are used to select data, algorithms are used to process data, and programs are used to integrate and run algorithms and protocols. Servers are laboratories providing blood cell analysis data. To each sample, servers analyze as many parameters as possible in order to reuse samples and reduce cost, so that there are huge amount of data in each database. Data in databases include raw data and processed data with preliminary protocols and algorithms. Clients have to pay fee for data access based on the volume of data.

Some practical difficulties prevent the usages of traditional client-server systems in CIP system. 1. Servers cannot provide all possible protocols and algorithms to process data for all clients. Especially for research users, they have to try their different algorithms and protocols frequently. 2. Research groups cannot save all data from different databases to their local sites. Some computation from research groups may involve most of data in databases. In addition, users have to provide same environments and systems to support the data if they are copied from servers. However, it is beyond the capacities of most users if it is not impossible. 3. There are two different databases for different samples, one is for hematology data, and another is for cytometry data. Some computations involve both databases, but these two databases may locate in different laboratories. Users have to coordinate these two databases during the computation. 4. Clients cannot afford copying all data to their local sites, especially the samples in servers increase at every moment, and the requirements from users change frequently.

We implement CIP system using mobile agent technologies. Clients send agents or programs with particular algorithms and protocols for servers. These agents run locally in servers, move to different servers to access different databases based on their intermediate results, and then deliver results to clients when agents finish their tasks. CIP has advantages such as offline computation, saving money on data transferring and maintenance, synchronization with latest data resources, flexibility on different services, and reducing data transportation on networks. It overcomes those difficulties from traditional client server systems.

CIP system structure is the same as the framework we discussed in section 2. Each client has an agent system that is used to create agents, send agents to servers, accept results from remote agents, and manage agents in remote sites. Suppose all servers and clients are connected with the Internet (based on TCP/IP protocol), and each site is installed with CORBA 2.0, which supports the interoperability between CORBA systems on different sites. Each server site has an agent system, which is supplied by clients and configured by servers. Each database has an interface that is responsible for database access, and agents access databases through these interfaces. We suppose these interfaces are implemented based on ODBC, but they only can read data from databases. Each agent system has a database, which is used to save processed or intermediate results from agents.

4 Modeling the CIP System

The CIP system includes three agent systems: one for client, and the other two for servers. The client may create different agents, however, the agent structure is same except the algorithms and protocols are different. In CIP system model, we only model one agent template, which can be instantiated as many different agents. So that the static view of CIP architecture is: $CIP = \{a, C, S_1, S_2\}$, where a is the agent template, C is the agent system for client, S_1 and S_2 are agent system for hematology analysis server and cytometry analysis server, respectively. Agents are created from C based on agent template a, and then they are sent to S_1 or S_2 by C. Agents do not cooperate with each other, but one agent may move between different servers S_1 and S_2 with results, and finally move back to C with their results.

4.1 Modeling Agents

We simplify the agent net in previous general models for this specific application. Each agent only communicates with its host. An agent sends retrieving statements (consisting with protocols) to hosts, and then the agent system in the host searches its database, and sends back data as results. As soon as the agent gets the data, it processes the data and saves the processed data. Each agent executes its task according to its statements in knowledge base and intermediate results. It also sends out requests to move itself out, which will be realized by the agent system in the host.

Tokens in agent net have the structure: *<dt, dl, da, sl, sa, type, cmd, msg>*, where *dt* is destination channel type, *false* means hosts, and *true* means agents. *dl, da, sl, sa* means destination host, destination agent, source host, source agent, respectively. The *type* means the message is an agent or a regular message, *MSG* means it is a ordinary message, and *AN* means it is an agent. The *cmd* could be some predefined commands such as *RST* means message is result, *MOV* means to ask agent system to move this agent out, *STOP* means to stop running of current agent, or *cmd* is an agent identifier. The *msg* could be regular data, or an agent net if *cmd* is agent identifier. From simplicity reason, we use head to represent *<dt, dl, da, sl, sa, type>*, and *obj* represent *<cmd, msg>* in following parts.

The transition *receive* gets messages from channel *DA*, which has the same value as the identifier of the agent. Then the transition *process* forwards the data to its output place. If the message is result data from server, it is sent to the agent for processing, and the result is saved at place *rst*. The agent processes the data using transition *run* according to its algorithms and protocols. The algorithms and protocols are represented as a knowledge base *kb*, and results are defined with a predicate *rst*. The knowledge base consists of a sequence of statements. It has a property *ref*, which points to the current statement *s*. The statement *s* is a structure data, which consists of its type and expression or data *obj*, i.e. *s* = <*TYPE*, *obj*>. There are three different types for *s*: *MSG*, *STOP* and *MOV*. The *MSG* means sending messages *obj* out, *STOP* means to stop running of current agent, and *MOV* means the agent request to move this agent out of current system. The *ref* is move to next statement when the transition *run* fires. The itinerary is defined with predicate *pt*, which is updated with the transition update. In predicate *pt*, there is a *next* attribute, which points to the next location the agent will go. During the processing, the agent may send messages out or request to move itself out. Transition *send_msg* is used to send messages to current host, and transition *stop_agent* is used to update agent itinerary and stop running of the agent. When an agent requests to move out, it has to stop the running of agent and save its current state, which is saved in *rst*. The *stop* and *start* transition controls running of the agent and it only can be started by the current host net. The following diagram shows the agent net for the CIP system.

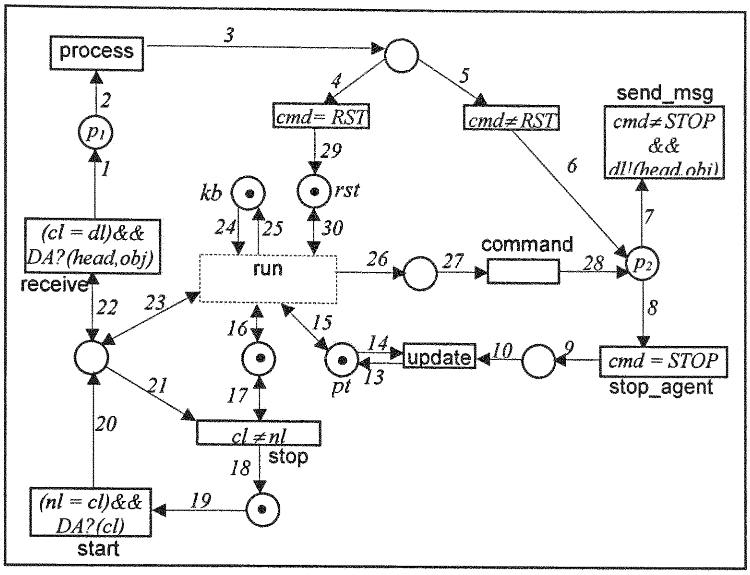

Figure 6.2 An agent net of the CIP system

Table 6.1 Legend of Figure 6.2

place /transition /inscription	Description
receive	Get incoming message from *DA* channel
process	Pre-process received data
send_msg	This agent sends messages to the current system through *dl* channel
stop_agent	This agent sends a request to stop the current agent
run	This agent runs its tasks, processes data from *rst*, saves results to *rst*, and sends out followed commands.
update	Agent update the reference to next location when this agent move out
stop	Stop the running of this agent
start	Start running of this agent
kb	Knowledge base for this agent, it includes algorithms and protocols of this agent
rst	Running results of this agent
pt	The itinerary of this agent
1, 2, 3	*<head, obj>*, where *head =<¢, dl, DA, ¢, dl, MSG, cmd, msg>*, and *cmd* ∈ *{STOP, RST}*
4, 29, 30	*<¢, dl, DA, ¢, dl, MSG, RST, msg>*
5, 6, 8	*<¢, dl, DA, ¢, dl, MSG, STOP, ¢>*
7, 26, 27, 28	*<¢, dl, DA, ¢, dl, MSG, cmd, msg>*
9, 10	*<¢>*
13, 14	*<next>*, *<next = next + 1>*
15	*<nl>*, *nl = cur(pt)*, *pt* is the itinerary, and *cur* is the reference pointing to the current location or the next destination if it is moving out.
16, 17, 18, 19	*<nl>*
20, 21, 22, 23	*<cl>*
24, 25	*<ref, s>*, *<ref = (ref + 1)/M, s>*, *M* is the number of total statements in *kb*. *kb* is the knowledge base which is consisted of statements.

4.2 Modeling Agent Systems

There are three agent system nets in the CIP system: one is for clients, and the other two for servers. The agent system for clients can create agents and save results from agents. The other two agent systems for servers have same functionalities except with different location information. All of three host nets are similar to the general model we gave in previous chapters since agent systems should have similar functionalities.

In the host net for clients, there is a predicate *cb*, which is the set of template of agent nets. Each token in *cb* is a template of agent net. We use transition *create* to generate agents, so that it outputs instances of agent templates to place p_3, and then the agent is started and put into place p_w. When start the agent, the token is *<AI, MN>*. As soon as the agent is start, the *kb* of the agent decides whether it will be move out to other systems or not. The dummy variable *DA* in agent net is replaced with *AI* in *MN.* When an agent returns to its home with results, the system starts it and put it into the place p_w. Then the agent sends a message with results to its home system (the *kb* of the agent has this statement), and the system puts the results into predicate p_6. For simplicity, we do not model agent behaviors after it delivers results. The following diagram shows the agent net for clients in the CIP system.

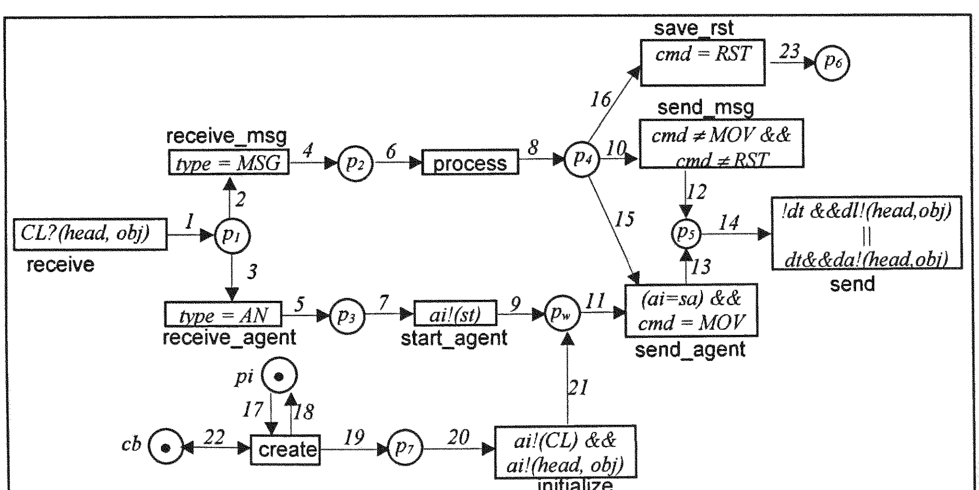

Figure 6.3 A host net for clients in the CIP system

Table 6.2, Legend of Figure 6.3

place/transition/inscription	Description
p_w	The place where mobile agents stay in, <*¢, CL, ¢, CL, sa, AN, ai, MN*>, and *sa* = *ai*.
p_1	The incoming messages from channels, <*false, CL, da, sl, sa, type, cmd, msg*>
p_5	The outgoing messages to channels, <*dt, dl, da, CL, sa, type, cmd, msg*>
p_2, p_4	Messages, <*false, CL, da, sl, sa, MSG, cmd, msg*>
p_3, p_7	Agents, <*false, CL, ¢, sl, sa, AN, ai, MN*>
p_6	Finally results, <*¢, CL, ¢, CL, sa, MSG, RST, msg*>
cb	Agent net templates, <*MN*>
pi	Agent identifier, <*ai*>
receive	Input transition, get tokens from *CL* channel
send	Output transition, send messages to *dl* or *da* channel
receive_msg	Receive messages (incoming tokens are not agents)
receive_agent	Receive agents (incoming tokens are agents)
process	Agent system processes the received messages
start_agent	Start the received agent
send_msg	Send messages to other agent systems or agents within this system
send_agent	Send out agents to other systems
create	Generate agent based on agent template and assign its indetifier
initialize	Initialize the generated agent net <*AI, AN*>
1	<*false, CL, da, sl, sa, type, cmd, msg*>
2, 4, 6, 8, 10, 15, 16	<*false, CL, da, sl, sa, MSG, cmd, msg*>
12	<*false, dl, da, CL, sa, MSG, cmd, msg*>
3, 5, 7, 9, 11	<*false, CL, ¢, sl, sa, AN, ai, MN*>
13	<*false, dl, ¢, CL, sa, AN, ai, MN*>
14	<*false, dl, da, CL, sa, type, cmd, msg*>
17, 18, 19, 20	<*ai*>, <*ai + 1*>, <*ai, an*>, <*ai, an*>
21	<*¢, CL, ¢, CL, ai, AN, ai, MN*>
22	<*an*>, where *an* is name of agent template

In system net for server 1 (for hematology analysis data), there is a predicate *db*, which represents the database with hematology data. The transition *select_data* is used to select data from database according to statements from other objects such as agents. The channel *LB1* represents this server. This agent system provides the basic functionalities of agent systems, so its transitions and places have the same meaning as that we discussed in general system net of mobile agent systems. The following diagram shows the host net for server 1 in CIP system.

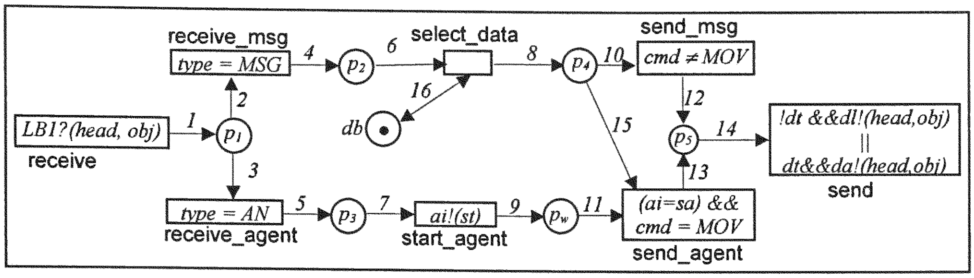

Figure 6.4 A host net for server 1 in the CIP system

Table 6.3, Legend of Figure 6.4

place/transition/inscription	Description
p_w	The place mobile agents stay in, <*¢, LB1, ¢, LB1, sa, AN, ai, MN*>, and *sa* = *ai*.
p_1	The incoming messages from channels. <*false, LB1, da, sl, sa, type, cmd, msg*>
p_5	The outgoing messages to channels. <*dt, dl, da, LB1, sa, type, cmd, msg*>
p_2, p_4	Messages, <*false, LB1, da, sl, sa, MSG, cmd, msg*>
p_3	Agents, <*false, LB1, ¢, sl, sa, AN, ai, MN*>
receive	Input transition, get tokens from *LB1* channel
send	Output transition, send messages to *dl* or *da* channel
receive_msg	Receive messages (incoming tokens are not agents)
receive_agent	Receive agents (incoming tokens are agents)
select_data	Select data from database *db* according request from agents
start_agent	Start the received agent
send_msg	Send messages to other agent systems or agents within this system
send_agent	Move out agents to other systems
1	<*false, LB1, da, sl, sa, type, cmd, msg*>
2, 4, 6, 8, 10, 15	<*false, LB1, da, sl, sa, MSG, cmd, msg*>
12	<*false, dl, da, LB1, sa, MSG, cmd, msg*>
3, 5, 7, 9, 11	<*false, LB1, ¢, sl, sa, AN, ai, MN*>
13	<*false, dl, ¢, LB1, sa, AN, ai, MN*>
14	<*false, dl, da, LB1, sa, type, cmd, msg*>
16	<*msg*>

The system net for server 2 (for cytometry analysis data) is same to server 1 except the location different. The following diagram shows the system net for server 2 in the CIP system.

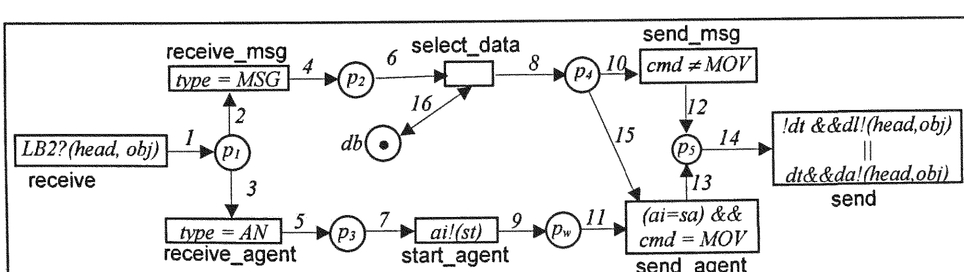

Figure 6.5 An host net for server 2 in the CIP system

5 Analyzing the CIP System

Based on the discussions and models of CIP system, we describe the working procedures of this system. Suppose the system only has one type of agents, i.e. there is only one template for agent net. In addition, there is only one active agent in this system. The agent is created in the client, and then it moves to server 1 for calculating hematology data, and then it moves to server 2 with results from server 1 to process data from the cytometry database. As soon as it finishes its tasks in server 2, it moves back to the client with its results.

From static architecture view, $CIP = \{A, C, S_1, S_2\}$, where A is the template for agent net, C is the system net for the server in client site, S_1 and S_2 is system net for server 1 and server 2, respectively. In the following section, we discuss the procedure from creating an agent to finally getting results from servers.

1. Create an agent template. Before we can create an agent, we must have a template for this type of agent net $AN = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$, which is a CPrT net. The *DA* in the agent template is a dummy value, which will be replaced with a real value of the agent identifier when an agent is created.

2. Create an agent. We model agent creation in the host net for clients. The templates of agent net are saved in predicate *cb*, and each token within it represents a type of agent. The token in *cb* is an agent net *MN*. However, there is only one token in *cb* since we only consider one type of agents. The token in place *pi* represents the identifier (an integer number) of next created agent. The transition *create* generates an agent <*AI, MN*> based on template in *cb* and identifier in *pi*, and agent *AI* is put into place p_5, *AI* plus 1 and sent back to *pi*. The dummy channel name *DA* in agent template is replaced with the real value *AI* in the agent net.

3. Initialize agent *AI*. As soon as agent *AI* is created and put into place p_5, it is initialized and put it into the agent place p_w. The transition *initialize* starts the agent, and then *ref* of *kb* is set

to point to the first statement in *kb*. Then the agent *AI* controls the computation, it may send message to server for requesting to move out, or asking for data from database.

4. Send out agent *AI* to server S_1. Suppose the itinerary in *pt* is *{CL, LB1, LB2, CL}*, which represents server 1 S_1, server 2 S_2 and client *C*. The reference *cur* points to *CL*. Suppose the statement *s = ref (kb), s.TYPE = MOV, s.obj =* <*LB1*>, then transition *run* sends a token <*¢, CL, ¢, CL, AI, MSG, STOP, CL*> to place p_2. The agent updates its *cur(pt)* to next location, here is 1 now. As soon as the transition *run* fires, agent is stop since $cl \neq nl$. The *run* sends out a token <*false, CL, ¢, CL, AI, MSG, MOV, LB1*> to place p_2. The transition *send_msg* sends this token to the host net *CL* through channel *CL*. When host net *CL* receives this token from channel *CL*, the token <*false, CL, ¢, CL, AI, MSG, MOV, LB1*> is sent to place p_4. Then the transition *send_agent* is enabled since p_w has this agent *AI*. The transition *send_agent* sends token <*false, LB1, ¢, CL, AI, AN, AI, MN*> to place p_5. As soon as channel *LB1* gets the token <*false, LB1, ¢, CL, AI, AN, AI, MN*>, the transition receive in S_1 fires, and send the token into place p_1, since the type is *AN*, the token is then send to p_3 as soon as *receive_agent* fires. The transition *start* sends *LB1* to channel *AI*, and then agent *AI* starts, and <*¢, LB1, ¢, CL, AI, AN, AI, MN*> is put into p_w for working.

5. Run agent *AI* in server S_1. When agent *AI* arrives place p_3 in system net S_1, it is started through sending *LB1* to channel *AI*, and the agent state is resumed from the stop point. The agent net <*AI, MN*> is put into the place p_w of S_1. The agent runs its task according to the statements from *kb*. If the *s = ref (kb), s.TYPE = MSG,* and *s.OBJ = SQL* statements, then the token go through place p_2, transition *send_msg* in *AI* net arrives at S_1. The token is processed by transition *select_data*, and data from *db* is wrapped as token <*true, LB1, AI, LB1, ¢, MSG, RST, msg*> and sent to p_5, where *msg* is the selected data. Then the token with data is sent to agent *AI* through channel *AI*. Finally, the data is processed by the transition *run* in *AI*, and the result is saved in place *rst*.

6. Move agent *AI* to server S_2. The procedure to move agent *AI* out is the same as the procedure in 4, and the only difference is the destination changes from *LB1* to *LB2*. The moving request is from agent *AI*, and the request to move agent out only comes from agents.

7. Run the agent in server S_2. This procedure is same as the procedure in 5.

8. Move agent *AI* back to client *C*. This procedure is same as the procedure in 4.

9. Run agent *AI* in *C*. When agent *AI* returns to home with results, it is started and put into place p_w of S_1. Then agent *AI* sends out a message with results *<false, CL, ¢, CL, AI, MSG, RST, msg>* to system *C*. As soon as *C* gets the token, it is forwarded to place p_4 of *C*, and then the transition *save_rst* processes the token and puts the results into place p_6.

6 Model Checking the CIP Models using SPIN

One of the important goals of building a formal architectural model of mobile agent systems is to help ensure the correct design that meets certain specifications and system requirements. A correct design should meet certain crucial requirements such as liveness, deadlock-free, and concurrency [XS03]. In this section, we use model checker SPIN to analyze and verify the simplified models of the CIP system. We check agent or host properties on agent nets and on host nets, respectively. We check system properties based on system-level nets, and interaction properties based on the connected nets composing from agent nets and host nets.

6.1 Model Checking a Host Net

In this part, we check the deadlock-free and reachability property of a host net for medical information servers in the CIP system, and the model is simplified for this specific analysis. We chose the host net of server 1 to analyze these properties. The following is the transformed net from the original CPrT net model for server 1.

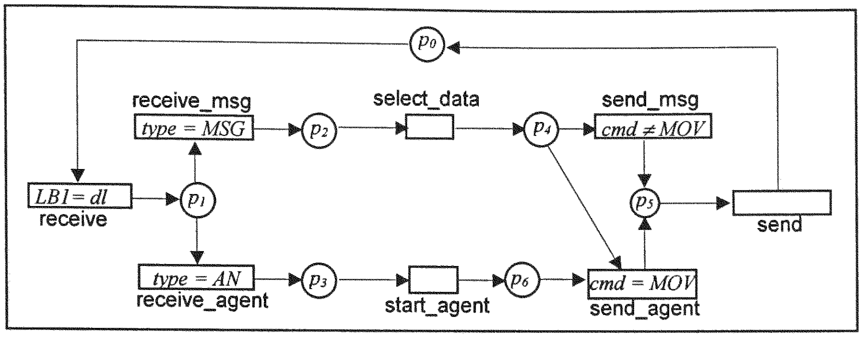

Figure 6.6 A host net in the CIP system

The token structure in this host net: *<dl, type, cmd>*, since we only consider receiving messages in this model without caring where they come from, we only need destination parameter *dl* here, there are two types of messages: agents *AN* and regular message *MSG*. The *cmd* is the agent identifier if the message is an agent, or *cmd* is the command *MOV*. We check the following properties: if the host net receives a message from its channel, eventually, this message will reach p_4 or p_6, and if the message is a *MOV* command, the token in p_6 will be moved to place p_0 (we only consider one agent in this model, so we do not need to compare agent identifiers). The following is the program and its running results (checking the safety and acceptable states).

```
/* we define LB1 as 1, MSG as 0, AN as 1, */
/* MOV as 1, and AI as 0, and ER as 0*/

#define          LB1   1
#define          MSG   0
#define          AN    1
#define          AI    0
#define          MOV   1
#define          ER    0

typedef Place {
      bit   dl;
      bit   type;
      bit   cmd
};
Place p0, p1, p2, p3, p4, p5, p6;
#define resetp(p) p.dl = 0; p.type = 0; p.cmd = 0

proctype hostnet()
{
  do
  /* receive */
```

```
:: atomic { p0.dl == LB1 ->
                        p1.dl = LB1;
                        p1.type = p0.type;
                        p1.cmd = p0.cmd;
                        resetp(p0)
   }
/* receive_msg */
:: atomic { (p1.type == MSG && p1.dl == LB1) ->
                        p2.dl = LB1;
                        p2.type = MSG;
                        p2.cmd = p1.cmd;
                        resetp(p1)
   }
/* select_data */
:: atomic { (p2.type == MSG && p2.dl == LB1) ->
                        p4.dl = LB1;
                        p4.type = MSG;
                        p4.cmd = p2.cmd;
                        resetp(p2)
   }
/* receive_agent */
:: atomic { (p1.type == AN && p1.dl == LB1) ->
                        p3.dl = LB1;
                        p3.type = AN;
                        p3.cmd = p1.cmd;
                        resetp(p1)
   }
/* start_agent */
:: atomic { (p3.type == AN && p3.dl == LB1) ->
                        p6.dl = LB1;
                        p6.type = AN;
                        p6.cmd = p3.cmd;
                        p0.dl = LB1;
                        p0.type = MSG;
                        p0.cmd = MOV;
                        resetp(p3)

   }
/* send message, cmd <> MOV */
:: atomic { (p4.type == MSG && p4.dl == LB1 &&
             p4.cmd != MOV) ->
                        p5.dl = LB1;
                        p5.type = MSG;
                        p5.cmd = p4.cmd;
                        resetp(p4)
   }

/* send_agent */

:: atomic { (p4.type == MSG && p4.dl == LB1 &&
             p4.cmd == MOV && p6.type == AN &&
             p6.dl == LB1 && p6.cmd == AI) ->
                        p5.dl = LB1;
                        p5.type = AN;
```

```
                                        p5.cmd = AI;

                                        resetp(p4);
                                        resetp(p6)
                    }

                /* send */
                :: atomic { if
                            :: (p5.type == MSG && p5.dl == LB1 &&
                                p5.cmd != MOV) ->
                                        p0.dl = LB1;
                                        p0.type = AN;
                                        p0.cmd = AI;
                                        resetp(p5)
                            :: (p5.type == AN && p5.dl == LB1 &&
                                p5.cmd == AI) ->
                                        p0.dl = LB1;
                                        p0.type = AN;
                                        p0.cmd = AI;
                                        resetp(p5)
                            fi
                    }
                accept: p0.type == AN && p0.dl == LB1 && p0.cmd == AI;
                od
        }

        init
        {
          p0.dl = LB1; p0.type = MSG; p0.cmd = ER;
          run hostnet()
        }
```

Figure 6.7 The Promela program for the Figure 6.6

The Results:

```
(Spin Version 4.0.7 -- 1 August 2003)
   + Partial Order Reduction

State-vector 44 byte, depth reached 17, errors: 0
      18 states, stored
       1 states, matched
      19 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
```

6.2 Model Checking an Agent Net

In this part, we check the deadlock-free and reachability property of an agent net in the CIP system, and the agent net is simplified for this specific analysis. The following is the transformed net from the original CPrT net model for the agent in the CIP system.

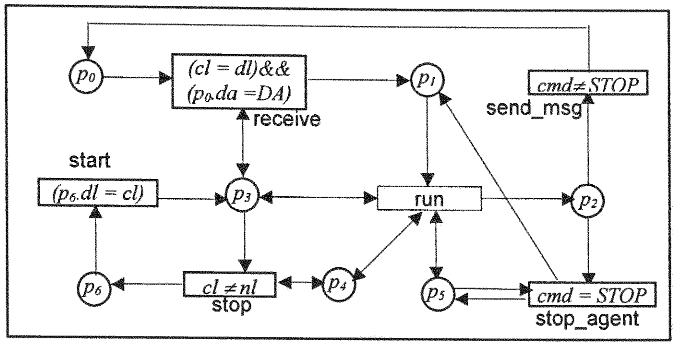

Figure 6.8 A simplified agent net

The token structure in this host net: *<dl, da, cmd>*, each agent has its location, which is its host net identifier *dl (*its first location is *CL)*. Because we only consider receiving messages in this model without caring about where they come from, we only need destination parameter *dl* and *da* here, and regular messages are the only type of messages in agent net, but *cmd* could be *STOP* or other commands. We check the following properties: if the agent net receives a message from its channels, eventually, this message will reach p_2 *(*processed by the agent*)*, and the *STOP* command can update the agent's location, which means *dl* become *dl +1*. As soon as the agent is sent out, the program updates its receiving channel *cl* to *nl*, so that it simulates the dynamic migration property of an agent. The following is the program and its running results (checking the safety and acceptable states).

```
/* we define CL as 1, DA as 111, STOP as 100, */
/* NL is the next destination of CL as 2, ER as 0*/
#define         CL    1
#define         NL    2
#define         DA    111
#define         STOP  100
#define         ER    0

typedef Place {
      byte  dl;
      byte  da;
      byte  cmd
};

Place p0, p1, p2, p3, p4, p5, p6;
byte nl;  /*next location of this agent */
```

```
#define resetp(p) p.dl = 0; p.da = 0; p.cmd = 0

proctype agentnet()
{
  do
  /* receive */
  :: atomic { (p0.dl == p3.dl && p0.da == DA) ->
                        p1.dl = p3.dl;
                        p1.da = p0.da;
                        p1.cmd = p0.cmd;
                        resetp(p0)
     }
  /* run */
  :: atomic { (p1.dl != ER && p3.dl != ER &&
               p4.dl != ER && p5.dl != ER) ->
                        p2.dl = p5.dl;
                        p2.da = p1.da;
                        p2.cmd = p1.cmd;
                        p4.dl = p5.dl;
                   resetp(p1)
     }
  /* stop_agent*/
  :: atomic { (p2.dl != ER && p2.cmd == STOP) ->
                        atomic {
                               nl = (nl + 1) % 10;
                               if
                               ::(nl == 0) -> nl = NL;
                               :: nl = nl;
                               fi
                        };
                        p5.dl = nl;
                        p5.da = ER;
                        p5.cmd = ER;
                        p1.dl = nl;
                        p1.da = DA;
                        p1.cmd = ER;
                        resetp(p2)
     }
  /* stop */
  :: atomic { (p3.dl !=ER && p4.dl != ER &&
               p3.dl != p4.dl) ->
                        p6.dl = p4.dl;
                        p6.da = ER;
                        p6.cmd = ER;
                        resetp(p3)

     }
  /* start */
  :: atomic { (p6.dl == p0.dl && p6.dl != ER) ->
                        p3.dl = p6.dl;
                        p3.da = ER;
                        p3.cmd = ER;
                        resetp(p6)
     }
  /* send_msg */
```

```
    :: atomic { (p2.dl != ER && p2.cmd != STOP) ->
                                  p0.dl = nl; /*to next host*/
                                  p0.da = p2.da;
                                  p0.cmd = p2.cmd;
                                  resetp(p2)
       }

  accept:
        p2.dl != ER && p4.dl != ER && p3.dl != ER;
  od
}

init
{
  nl = 1;
  p0.dl = CL; p0.da = DA; p0.cmd = STOP;
  p6.dl = CL; p6.da = ER; p6.cmd = ER;
  p5.dl = CL; p5.da = ER; p5.cmd = ER;
  p4.dl = CL; p4.da = ER; p4.cmd = ER;
  run agentnet()
}
```

Figure 6.9 The Promela program for the Figure 6.8

The Results:

```
(Spin Version 4.0.7 -- 1 August 2003)
   + Partial Order Reduction

State-vector 40 byte, depth reached 28, errors: 0
      29 states, stored
       1 states, matched
      30 transitions (= stored+matched)
       2 atomic steps
hash conflicts: 0 (resolved)
```

6.3 Model Checking an Interaction

In this part, we check the deadlock-free and reachability property of a two-level mode, which is a simplified model composing from one agent net and its host net. Before the agent is moved to place p_6 in the host net, it is started through message passing from the host net to the agent net. As soon as the agent is started and moved to its place p_6, the host sends a request to move it out. The moving is realized through two steps: first, the host sends a stop command to the agent to stop the execution of the agent; second, the agent updates its itinerary to the next destination and sends a message to current host to migrate the agent to the next destination. As soon as the agent is moved out, the locations of host net and agent net are updated as the next destination so that the program can simulate the dynamic migration of mobile agents. The moving (such as stopping an agent, updating its location) and receiving agents (such as starting an agent) are completed with the cooperation between agents and their hosts. The following is the transformed net from the original CPrT net model for the agent and one host of CIP system.

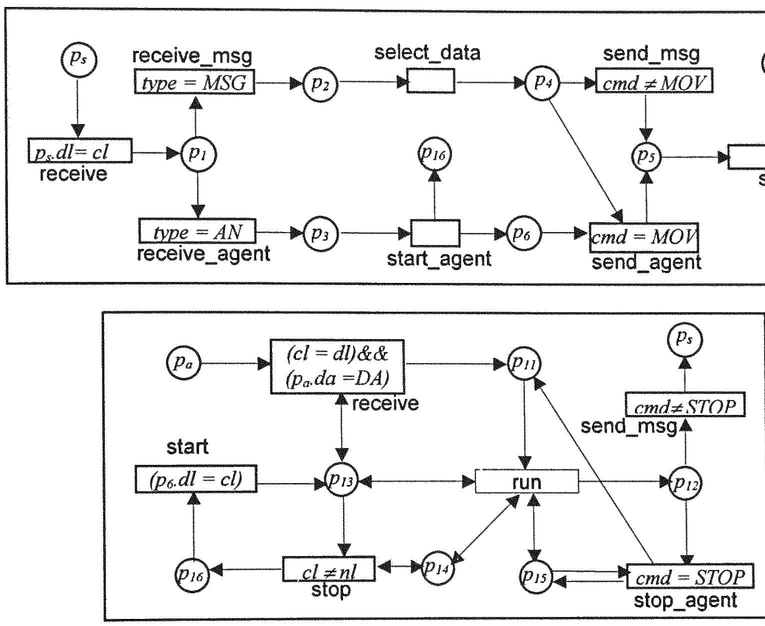

Figure 6.10 An interaction model of the CIP system

The token structure in this host net: *<dl, da, type cmd>*, where *dl* is the location of current host and the agent (the first location is *LB1* or *CL)*, *da* is the agent identifier, *type* is the message type: agents or regular messages, and *cmd* could be *STOP*, *MOVE* or agent identifier *AI* if the message is an agent. Because we only consider receiving messages in this model without caring about where they come from, we only need destination parameter *dl* and *da* here. We check the following properties: if the host net receives an agent, the agent will be started and put into place p_6, and the agent is sent to its destination if the host requests to move out the agent. As soon as the agent is sent out, the program updates the receiving channel *cl* and *dl* to *nl (nl* is dynamically updated by the agent*)*, so that it simulates the dynamic migration and interaction property of an agent and its host. The following is the program and its running results (checking the safety and acceptable states).

```
/* we define LB1, CL as 1, MSG as 66, AN as 88 */
/* AI, DA for agent ID is 10, */
/* MOV as 111, and STOP as 222 */
/* NL is the initialize next destination as 2 */
/* ER is empty as 0 */

#define         LB1   1
#define         MSG   66
#define         AN    88
#define         AI    10
#define         MOV   111
#define         STOP  222
#define         CL    1
#define         NL    2
#define         DA    10
#define         ER    0

typedef Place {
      byte  dl;
      byte  da;
      byte  type;
      byte  cmd;
};

Place ps, p1, p2, p3, p4, p5, p6;
Place pa, p11, p12, p13, p14, p15, p16;
byte cl, dl, nl;

#define resetp(p) p.dl = 0; p.da = 0; p.type = 0; p.cmd = 0

proctype hostnet()
{
  do
  /* receive */
  :: atomic { ps.dl == dl ->
                           p1.dl = ps.dl;
                           p1.da = ps.da;
                           p1.type = ps.type;
                           p1.cmd = ps.cmd;
                           resetp(ps)
     }
  /* receive_msg */
  :: atomic { (p1.type == MSG) ->
                           p2.dl = p1.dl;
                           p2.da = p1.da;
                           p2.type = MSG;
                           p2.cmd = p1.cmd;
                           resetp(p1)
     }
  /* select_data */
```

```
:: atomic { (p2.type == MSG) ->
                         p4.dl = p2.dl;
                         p4.da = p2.da;
                         p4.type = MSG;
                         p4.cmd = p2.cmd;
                         resetp(p2)
   }
/* receive_agent */
:: atomic { (p1.type == AN && p1.da != ER) ->
                         p3.dl = p1.dl;
                         p3.da = p1.da;
                         p3.type = AN;
                         p3.cmd = p1.cmd;
                         resetp(p1)
   }
/* start_agent */
:: atomic { (p3.type == AN && p3.dl == dl) ->
                         p6.dl = p3.dl;
                         p6.da = p3.da;
                         p6.type = AN;
                         p6.cmd = p3.cmd;
                         p16.dl = dl;
                         p16.da = ER;
                         p16.type = ER;
                         p16.cmd = ER;
                         resetp(p3)

   }
/* send message, cmd <> MOV, do not simulate sending out
/* other kinds of messages, we did that in host models */
:: atomic { (p4.cmd != MOV) ->
                         resetp(p4)
   }
/* send_agent */
:: atomic { (p4.type == MSG && p4.cmd == MOV
          && p6.type == AN && p6.cmd == AI) ->
                         dl = (dl + 1) % 10;
                         if
                         :: (dl == LB1) -> dl = NL;
                         :: else -> dl = dl
                         fi;
                         p5.dl = dl;
                         p5.da = DA;
                         p5.type = AN;
                         p5.cmd = AI;
                         resetp(p4);
                         resetp(p6)
   }

/* send  */
:: atomic { (p5.type == AN ) ->
                         pa.dl = p5.dl;
                         pa.da = p5.da;
                         pa.type = MSG;
                         pa.cmd = STOP;
```

```
                          ps.dl = p5.dl;
                          ps.da = p5.da;
                          ps.type = AN;
                          ps.cmd = AI;
                          resetp(p5)
      }
   accept: (p6.dl != ER) && (p6.type == AN)
  od
}

proctype agentnet()
{
  do
  /* receive */
  :: atomic { (pa.dl == cl && pa.da == DA &&
             p13.dl == cl) ->
                          p11.dl = cl;
                          p11.da = DA;
                          p11.type = MSG;
                          p11.cmd = pa.cmd;
                          resetp(pa)
     }
  /* run */
  :: atomic { (p11.dl != ER && p11.da == DA &&
             p13.dl != ER && p14.dl != ER &&
             p15.dl != ER) ->
                          p12.dl = p11.dl;
                          p12.da = DA;
                          p12.type = MSG;
                          p12.cmd = p11.cmd;
                          p14.dl = p15.dl;
                          resetp(p11)
     }
  /* stop_agent*/
  :: atomic { (p12.da == DA && p12.cmd == STOP) ->
                          nl = (nl + 1) % 10; /*10 hosts*/
                          if
                          :: (nl == CL) -> nl = NL;
                          :: else -> nl = nl
                          fi;
                          p15.dl = nl;
                          p15.da = ER;
                          p15.type = ER;
                          p15.cmd = ER;
                          p11.dl = cl;
                          p11.da = DA;
                          p11.type = MSG;
                          p11.cmd = MOV;
                          resetp(p12)
     }
  /* stop */
  :: atomic { (p13.dl == cl && p14.dl != ER &&
             p13.dl != p14.dl) ->
                          p16.dl = p14.dl;
                          p16.da = ER;
```

```
                                    p16.type = ER;
                                    p16.cmd = ER;
                                    resetp(p13)
               }
             /* start */
             :: atomic { (p16.dl == cl) ->
                                    p13.dl = p16.dl;
                                    p13.da = p16.da;
                                    p13.type = p16.type;
                                    p13.cmd = p16.cmd;
                                    cl = nl;
                                    resetp(p16)
               }
             /* send_msg */
             :: atomic { (p12.dl != ER && p12.cmd != STOP) ->
                                    ps.dl = p12.dl;
                                    ps.da = p12.da;
                                    ps.type = p12.type;
                                    ps.cmd = p12.cmd;
               }
             accept:
                    (pa.dl != ER) && (p14.dl == p15.dl);
             od
           }

           init
           {
             dl = LB1; cl = CL; nl = CL;

             ps.dl = LB1; ps.da = DA; ps.type = AN; ps.cmd = AI;

             pa.dl = CL; pa.da = DA; pa.type = MSG; pa.cmd = STOP;
             p15.dl = CL; p15.da = ER; p15.type = ER; p15.cmd = ER;
             p14.dl = CL; p14.da = ER; p14.type = ER; p14.cmd = ER;

             atomic {run hostnet(); run agentnet()}
           }
```

Figure 6.11 The Promela program for the Figure 6.10

The Results:

```
(Spin Version 4.0.7 -- 1 August 2003)
   + Partial Order Reduction

State-vector 80 byte, depth reached 41, errors: 0
      49 states, stored
      22 states, matched
      71 transitions (= stored+matched)
       3 atomic steps
hash conflicts: 0 (resolved)
```

6.4 Model Checking the Mobility

In this part, we check the deadlock-free and reachability property of a system-level model, which is the model composing from two host nets. When an agent or message is sent out from one host to another, eventually the destination host will receive the agent or message. We abstract channels as common places within two nets. We treat agents as regular tokens in this model since we already checked host nets, agent nets and the interaction model. The following is the transformed net from the original CPrT net model for two host nets of the CIP system.

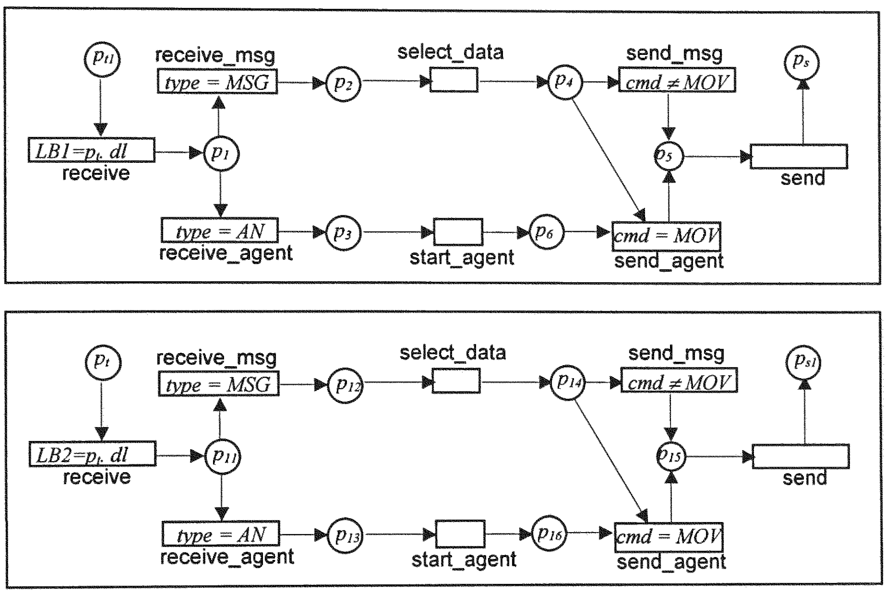

Figure 6.12 A system model of the CIP system

The token structure in this host net: *<dl, sl, type cmd>*, where *dl* is the destination location, sl is the source location (only two locations: *LB1* and *LB2*), *type* is the message type: agents or regular messages, and *cmd* could be *STOP*, *MOVE* or agent identifier *AI* if the message is an agent. We check the following properties: if the host net *LB1* sends a message or an agent to destination host *LB2*, eventually it will arrive at its destination *LB2*, and if the host net *LB2* sends a message or an agent to destination host *LB1*, eventually it will arrive at its destination *LB1*. The following is the program and its running results (checking the safety and acceptable states).

```
/* we define LB1 as 11, LB2 as 22 */
/* MSG as 66, AN as 88 */
/* MOV as 111, and AI as 100, ER as 0 */

#define         LB1   11
#define         LB2   22
#define         MSG   66
#define         AN    88
#define         AI    100
#define         MOV   111
#define         ER    0

typedef Place {
      byte  dl;
      byte  sl;
      byte  type;
      byte  cmd
};

Place ps, p1, p2, p3, p4, p5, p6;
Place pt, p11, p12, p13, p14, p15, p16;

#define resetp(p) p.dl = 0; p.sl = 0; p.type = 0; p.cmd = 0

proctype hostnet1()
{
  do
  /* receive */
  :: atomic { ps.dl == LB1 ->
                           p1.dl = LB1;
                           p1.sl = ps.sl;
                           p1.type = ps.type;
                           p1.cmd = ps.cmd;
                           resetp(ps)
     }
  /* receive_msg */
  :: atomic { (p1.type == MSG) ->
                           p2.dl = LB1;
                           p2.sl = p1.sl;
                           p2.type = MSG;
                           p2.cmd = p1.cmd;
                           resetp(p1)
     }
  /* select_data */
  :: atomic { (p2.type == MSG) ->
                           p4.dl = LB1;
                           p4.sl = p2.sl;
                           p4.type = MSG;
                           p4.cmd = p2.cmd;
                           resetp(p2)
     }
  /* receive_agent */
  :: atomic { (p1.type == AN ) ->
                           p3.dl = LB1;
```

```
                              p3.sl = p1.sl;
                              p3.type = AN;
                              p3.cmd = p1.cmd;
                              resetp(p1)
     }
  /* start_agent */
  :: atomic { (p3.type == AN) ->
                              p6.dl = LB1;
                              p6.sl = p3.sl;
                              p6.type = AN;
                              p6.cmd = p3.cmd;
                              ps.dl = LB1;
                              ps.sl = LB1;
                              ps.type = MSG;
                              ps.cmd = MOV;
                              resetp(p3)

     }
  /* send message, cmd <> MOV */
  :: atomic { (p4.type == MSG && p4.cmd != MOV) ->
                              p5.dl = LB2;
                              p5.sl = LB1;
                              p5.type = MSG;
                              p5.cmd = ER;
                              resetp(p4)
     }
  /* send_agent */
  :: atomic { (p4.type == MSG && p4.cmd == MOV
              && p6.type == AN && p6.cmd == AI) ->
                              p5.dl = LB2;
                              p5.sl = LB1;
                              p5.type = AN;
                              p5.cmd = AI;
                              resetp(p4);
                              resetp(p6)
     }

  /* moving out the agent  */
  :: atomic { (p5.dl == LB2) ->
                              pt.dl = LB2;
                              pt.sl = LB1;
                              pt.type = p5.type;
                              pt.cmd = p5.cmd;
                              resetp(p5)
     }
   accept: pt.dl == LB2 /*message is sent to destination*/
 od
}

proctype hostnet2()
{
do
  /* receive */
  :: atomic { pt.dl == LB2 ->
                              p11.dl = LB2;
```

```
                        p11.sl = pt.sl;
                        p11.type = pt.type;
                        p11.cmd = pt.cmd;
                        resetp(pt)
    }
/* receive_msg */
:: atomic { (p11.type == MSG) ->
                        p12.dl = LB2;
                        p12.sl = p11.sl;
                        p12.type = MSG;
                        p12.cmd = p11.cmd;
                        resetp(p11)
    }
/* select_data */
:: atomic { (p12.type == MSG) ->
                        p14.dl = LB2;
                        p14.sl = p12.sl;
                        p14.type = MSG;
                        p14.cmd = p12.cmd;
                        resetp(p12)
    }
/* receive_agent */
:: atomic { (p11.type == AN ) ->
                        p13.dl = LB2;
                        p13.sl = p11.sl;
                        p13.type = AN;
                        p13.cmd = p11.cmd;
                        resetp(p11)
    }
/* start_agent, and request to move out */
:: atomic { (p13.type == AN) ->
                        p16.dl = LB2;
                        p16.sl = p13.sl;
                        p16.type = AN;
                        p16.cmd = p13.cmd;
                        pt.dl = LB2;
                        pt.sl = LB2;
                        pt.type = MSG;
                        pt.cmd = MOV;
                        resetp(p13)

    }
/* send out message, cmd <> MOV */
:: atomic { ( p14.type == MSG && p14.cmd != MOV ) ->
                        p15.dl = LB1;
                        p15.sl = LB2;
                        p15.type = MSG;
                        p15.cmd = p14.cmd;
                        resetp(p14)
    }
/* send_agent */
:: atomic { (p14.type == MSG && p14.cmd == MOV &&
                p16.type == AN && p16.cmd == AI) ->
                        p15.dl = LB1;
                        p15.sl = LB2;
```

```
                                  p15.type = AN;
                                  p15.cmd = AI;
                                  resetp(p14);
                                  resetp(p16)
       }

    /* moving out the agent  */
    :: atomic { (p15.type != ER && p15.dl == LB1) ->
                                  ps.dl = LB1;
                                  ps.sl = LB2;
                                  ps.type = MSG;
                                  ps.cmd = ER;
                                  resetp(p15)
       }
    accept2: (ps.dl != ER) /*message is sent out*/
  od
  }

  init
  {
    ps.dl = LB1; ps.sl = LB2; ps.type = AN; ps.cmd = AI;
    atomic { run hostnet1(); run hostnet2() }
  }
```

Figure 6.13 The Promela program for the Figure 6.12

The results:

```
(Spin Version 4.0.7 -- 1 August 2003)
   + Partial Order Reduction

State-vector 76 byte, depth reached 34, errors: 0
      53 states, stored
       3 states, matched
      56 transitions (= stored+matched)
       1 atomic steps
hash conflicts: 0 (resolved)
```

7 Concluding Remarks

In this chapter, we propose an architectural model for a medical information processing system based on mobile agents. It demonstrates advantages such as the flexibility, high efficiency, less cost of mobile agent technology. The CIP system includes one agent and three servers. The agent migrates, retrieves and processes medical information in different servers, and delivers results back to its users. There are two different servers; one is used in client sides, which has the functionality to create agents for specific tasks; and another is used in server sides, which provides basic functionalities to support the execution of agents. We model the agent net, host nets (for servers or clients) using CPrT nets and these nets communicate and interact through dynamic channels. We analyze the reachability property of the CPrT models and the cooperation between host nets and agent nets. From the success of modeling and analyzing models of the CIP system, we demonstrate the expressive power of CPrT nets, especially the advantage of dynamical channels, which naturally capture the dynamic property of mobile agent systems. We chose model checking tool SPIN to verify some properties such as reachability, deadlock free and safety of CIP system based on hierarchical analysis method. The results show that model checking is an effective way to verify software architectures. It is almost impossible to manually verify or prove a complex software system, so that the automation of model checking is an obvious advantage. When model checking method is integrated with hierarchical analysis method, it is possible to automatically verify much larger and more complex systems.

CHAPTER VII

Conclusion

Formally modeling and analyzing software architecture of mobile agent systems is a challenging work because of their complexity and dynamic reconfiguration of their architectures. We address this issue from two ways: a formalism to define the system architecture and an analysis method to formally verify system properties. The formalism is a PrT net extended with dynamic channels, and the analysis is a hierarchical method for model checking. We borrow the multi-layer modeling paradigm from EOS to CPrT nets so that the formalism is suitable to model mobile agent systems. From successful modeling and analysis of mobile agent systems and other systems with code mobility, we conclude that CPrT net is a powerful tool to model mobile computing systems. The two-layer modeling paradigm smoothly transforms physical models of mobile agent systems to their formal architectural models. Since agents and agent systems are two relative independent systems, this framework brings us convenience to focus on a particular system without involving the complexity of its environments. Moreover, it is also useful to analyze models since we analyze them on a particular level and conside models on other levels as interfaces. The channel naturally captures the dynamic configuration property of mobile systems, and it facilitates the synchronous communication between different nets. Communicative objects change their communication topologies with the changes of their environments at run time since channel values are dynamically assigned during the execution. The dynamic channel provides a mechanism to construct easier-to-understand and more compact models because each dynamic channel is a finite set of static channels. In addition, the software architecture of mobile agent systems essentially has a hierarchical structure, so we introduce a hierarchical analysis method to verify the software architecture. We verify component properties based on transformed individual components, and then system properties are checked based on simplified system models. Only properties involving two different models are analyzed on connected models. The hierarchical analysis method provides a solid foundation for the software architecture of mobile agent systems. It not only reduces the analysis complexity, but also expands the application scope of model checking technology. From successful modeling and analysis of these systems, we can deeply understand mobile agent systems especially the mobility and cooperation properties. It is helpful to model and analyze other complex concurrency systems as well. We propose an architectural model for a medical information processing system based on mobile agents. It shows high level flexibility, high efficiency, low cost of mobile agent technology. It provides a practical and convincing case for the application of mobile agents. We chose model checking tool SPIN to verify properties such as reachability, concurrency and safety of CIP system based on hierarchical analysis method. The results show that model checking is an effective and efficient way to verify software architectures. Integrating hierarchical analysis method with model checking technique brings the possibility to automatically verify much larger and more complex systems.

In this dissertation, we only address the synchronous communication between components, and channels in CPrT nets are introduced for this purpose. It is enough to model mobile systems in this dissertation; however, asynchronous communication between nets is also an important research topic especially for real time systems, which is one of our future research topics. We translate CPrT net models into Promela programs manually, but it is a tedious work and it is difficult to guarantee the consistency between the net models and their Promela programs. We are developing a system to translate CPrT net models into Promela programs, but it still requires users to input initial markings and define some variable types. Although we propose a hierarchical analysis method to verify the software architecture of mobile agent systems using model checking, the method still is the complete model checking, i.e. we first verify the correctness of individual components and then verify the correctness of a composition by connecting the individual components into a single composition level model. This approach works in most situations due to the high-level abstraction of software architectures. However, the connected composition level model can be quite large in some situations to prevent the effective use of model checking techniques. Compositional model checking techniques are potential methods to solve this problem.

LIST OF REFERENCES

[AB96] A. Asperti, N. Busi, "Mobile Petri Nets," *Technical Report UBLCS-96-10*, Laboratory for Computer Science, University of Bologna, Italy, 1996

[ACD90] R. Alur, , C.Courcoubetis, , and D. Dill, "Model-checking for real-time systems", In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science,* New York, 1990

[BCC98] Sergey Berezin, Sergio Campos, Edmund M. Clarke, "Compositional Reasoning in Model Checking", *Technical Report, CMU-CS-98-106*, Carnegie Mellon University, February 1998

[BCL90] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic Model Checking with Partitioned transition relations", In *VLSI 91*, Edinburgh, Scotland, 1990

[BCM+a90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond." In *Proceeding of the Fifth Annual Symposium on Logic in Computer Science*, 1990

[BCM+b90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking." *In 27th ACM/IEEE Design Automation Conference*, 1990

[BD91] Bernard Berthomieu and Michel Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets" *IEEE Transactions on Software Engineering*, vol. 17, no. 3, March 1991

[BFF95] E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel, and E.Pelz, "A Class of Composable High Level Petri Nets," *ICATPN'1995, LNCS,* vol. 935, pp.103-120, 1995

[BM96] F. Buschmann and R. Menuier, *A System of Patterns*, Wiley, 1996

[BO98] E. Badouel and J. Oliver, "Reconfigurable Nets: A Class of High-Level Petri Nets Supporting Dynamic Changes with Workflow Systems," *INRIA Research Report*, PI-1163, 1998.

[Bry86] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, C-35(8) 677-691, 1986

[Cam93] Sergio V. Campos, "The priority Inversion Problem and Real-Time Sysmbolic Model Checking", *CMU-CS-93-125*, SCS, Carnegie Mellon University, 1993

[Cam96] S. V. Campos, "A Quantitative Approach to the Formal Verification of Real-Time System", *PhD thesis, SCS*, Carnegie Mellon University, 1996

[CBG+91] Clarke, E. M., Burch, J., Grumberg, O., Long, D., and McMillan, K., "Automatic verification of sequential circuit designs", In *Conference Proceedings of the Royal Society of London,* 1991

[CBM89] O. Coudert, C. Berthet, and J. Madre, "Verification of synchronous sequential machines based on symbolic execution", In *Proceedings of the 1st Workshop on Computer-Aided Verification. LNCS*, vol. 407, Springer-Verlag, Berlin, 1989

[CC94] S. Campos, E. M. Clarke, "Real-Time Symbolic Model Checking for Discrete Time Models", *CMU-CS-94-146*, Carnegie Mellon University, May 1994

[CCM+94] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitative Characteristics of Finite-State Real-Time Systems", *Technical report, CMU-CS-94-147*, Carnegie Mellon university, May 1994

[CCM+96] S. Campos, E. Clarke, W. Marrero and M. Minea, “Verus: A Tool for Quantitative Finite-State Real-Time Systems”, *Technical report, CMU-CS-96-159*, Carnegie Mellon university, August 1996

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specification.” *ACM Trans. on Programming Language and System*, vol. 8, no.2, pp. 244-263, 1986

[CFM00] P. Ciancarini, F. Franze, and C. Mascolo, “Using a Coordination Language to Specify and Analyze Systems Containing Mobile Components.” *ACM Trans. On Software Engineering and Methodology*, vol. 9, no.2, pp. 167-198, 2000

[CGL93] E. M. Clarke, O. Grumberg, and D. E. Long, Verification tools for finite-state concurrent systems. *In REX' 93 School/Workshop: A decade of Concurrency*, Noordwijkerhout, The Netherlands, June 1993

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model Checking,* The MIT Press, Cambridge, 1999

[CH94] S. Christensen, N.D. Hansen, “Coloured Petri Nets Extended with Channels for Synchronous Communication.” In *Application and Theory of Petri Nets*, pp. 159-178, 1994

[CK96] S. C. Cheung, J. Kramer, “Context Constraints for Compositional Reachability Analysis”, *ACM Transactions on Software Engineering and Methodology*, vol5, no. 4, pp. 334-377, October 1996

[CKL+95] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, “Synthesizing Petri nets from state-based models”, *Technical Report RR 95/09 UPC/DAC*, Universitat Politecnica de Catalunya, April 1995

[CLM89] E. M. Clarke, D. E. Long and K. L. McMillan, “Compositional Model Checking”, *Technical Report, CMU-CS-89-145*, Carnegie Mellon University, April 19, 1989

[CM88] K. M. Chandy and J. Misra, “*Parallel Program Design: A Foundation.*” Addison-Wesley, 1988.

[DDA96] Y. Deng, W. Du, P. C. Attie and M. Evangelist, “A Formal Approach for Architectural Modeling and Decomposition of Distributed Real-time Systems.” In *Proc. of 8th International Conference on Software Engineering and Knowledge*, Nevada, 408-417, 1996

[DDX99] Y. Deng, J. Ding, and D. Xu, "Formalizing MASIF interoperable architecture of mobile agent systems," *Technical Report*. FIU-CS-CADSE, December 1999

[Dim99] Dimitra Giannakopoulou, “Model Checking for Concurrent Software Architecture”, *Ph.D. thesis*, Imperial College of Science, Technology and Medicine, University of London, January 1999.

[Din00] J. Ding, “An Approach for Model Checking Petri Nets-Based Software Architecture,” *Master Thesis,* School of Computer Science, Florida International University, May 2000

[DXG03] J. Ding, X. He, D. Xu, S. Gao, Y. Deng, “Analyzing LAM Architecture using Model Checking”, *preparing paper,* December 2003

[Emr90] E.A. Emerson, Temporal and modal logic, *Handbook on Theoretical Computer Science*, vol. B, Elsevier Science, pp. 995-1072, 1990.

[EP02] C. Eisner, and D. Peled, "Comparing Symbolic and Explicit Model Checking of a Software System," *SPIN2002*, April 2002

[ERV96] J. Esparza, S. Romer and W. Vogler, "An Improvement of McMillan's Unfolding Algorithm," In *TACAS'96, LNCS,* vol. 1055, pp. 87-106, 1996.

[FB98] J.M. Fernandes, and O. Belo, "Modeling Multi-Agent Systems Activities Through Colored Petri Nets", In 16th IASTED International Conference on Applied Informatics (AI'98), pp. 17-20, Germany, February 1998

[FIP00] Foundation for Intelligent Physical Agents, *FIPA Agent Management Support for Mobility Specification,* http://www. Fipa.org/specs/fipa00087, June 2000

[FPV98] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. on Software Engineering*, vol. 24, pp. 342-361, May 1998

[Gay96] R.S. Gray, "Agent Tcl: A Flexible and Secure Mobile Agent System," In *Proceedings of the Fouth Annual Tcl/Tk Workshop* (TCL '96), July 1996

[GCK01] R.S. Gray, G. Cybenko, D. Kotz, and D. Rus, "Mobile Agents: Motivations and States of the Art," *Handbook of Agent Technology*, J. Bradshaw, ed., 2001

[Gen81] H.J. Genrich, and K. Lautenbach, "System Modeling with High-level Petri Nets," Theoretical Computer Science, vol. 13, pp.109-136, 1981

[Gen87] H.J. Genrich, "Predicate/Transition Nets," *Petri Nets: Central Models and Their Properties*, W. Brauer, W. Resig, and G. Rozenberg, eds., pp. 207-247, 1987

[Gei95] K. Geiger, *Inside ODBC*, Microsoft Press, 1995

[CGL99] E.M. Clark, O. Grumberg, and D. E. Long, *Model Checking*, Cambridge, MA. MIT Press, 1999

[GL94] O. Grumberg, and D. E. Long, "Model Checking and Modular Verification", *ACM Trans. on Programming Languages and Systems*, vol. 16, no. 3, pp. 843-871, May 1994

[GH02] P.R. Gluck, and G.J. Holzmann, "Using SPIN model checking for flight software verification," *IEEE Aerospace Conference Proceedings*, vol. 1, pp. 105-113 March 2002

[GL94] Orna Grumberg and David Long, "Model Checking and Modular Verification", *ACM Transaction on Programming Language and Systems*, 16(3): 843-871, May 1994

[GP98] B. Grahlmann and C. Pohl, "Profiting from Spin in PEP." *SPIN Workshop*, Paris 1998.

[Gra96] R.S. Gray, "Agent Tcl: A Flexible and Secure Mobile Agent System," In *Proceedings of the Fouth Annual Tcl/Tk Workshop* (TCL '96), July 1996

[GS03] D. Garlan, and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola, and G. Tortora eds, World Scientific Publishing Company, Singapore, pp. 1-39, 1993

[HD02] X. He, and Y. Deng, "A Framework for Developing and Analyzing Software Architecture Specifications in SAM". *The Computer Journal*, vol. 45, no. 1, pp. 111—128, 2002

[HD01] J. Hulass, and D. Buchs, "An Experiment with Coordinated Algebraic Petri Nets as Formalism for Modeling Mobile Agents", In *Workshop on Modeling of Objects, Components, and Agents (MOCA'01)*, pp. 73-84, DAIMI PB-553, Aarhus University, August 2001

[HDD02] X. He, J. Ding, and Y. Deng, "Analyzing SAM Architectural Specifications Using Model Checking", *SEKE2002*, Italy, 2002

[HM95] C. Heitmeyer and D. Mandrioli, "Formal Methods for Real-time Computing: An Overview", *Formal Methods for Real-time Computing*, pp.1-29, 1995

[HNS91] Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S, "Symbolic model checking for real-time systems" In *Proceedings of the 7th Symposium on Logics in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif, 1991

[Hoa85] C.A.R. Hoare, "*Communicating Sequential Processes.*" Prentic-Hall International, UK 1985

[Hol97] G.J. Holzmann, "The Model Checker Spin." *IEEE Trans. on Software Engineering.* vol.23, no. 5, pp. 279-295, May 1997

[Hol03] G.J. Holzmann, *The Spin Model Checker: Primer and reference manual*, Boston, MA. Addison-Wesley, 2003

[HS99] G.J. Holzmann and M.H. Smith, "Software Model Checking", *FORTE*, pp.481-497, 1999

[HYS03] X. He, H. Yu, T. Shi, J. Ding, Y. Deng, "Formally Analyzing Software Architecture Specifications using SAM", *The Journal of Systems and Software*, vol.71 pp.11-29, 2003

[Jon94] B. Jonsson, "Compositional Specification and Verification of Distributed Systems", *ACM Transactions on Programming Languages and Systems*, vol.16, no. 2, pp. 917-979, March 1994

[JTM98] E. Y.T. Juan, J. J.P. Tsai, T. Murata, "Compositional Verification of Concurrent Systems Using Petri-Net-Based Condensation Rules", *ACM Transactions on Programming Languages and Systems*, vol20, no. 5, pp. 917-979, September 1998,

[KK01] V. Khomenko and M. Koutny, "Towards an Efficient Algorithm for Unfolding Petri Nets." *CONCUR'2001, LNCS 2154*, 366-380, 2001

[KLO97] G. Karjoth, D. Lange, and M. Oshima, "A Security Model for Aglets," *IEEE Internet Computing*, pp.68-77, July-August 1997

[KMR01] M. Kohler, D. Moldt, and H. Rolke, "Modeling the behavior of Petri net agents," In J.M. Colom and M. Koutny, editors, *Proceedings of the 22nd Conference on Application and Theory of Petri Nets, LNCS*, vol. 2705, pp. 224-241, June 2001

[KMR03] M. Kohler, D. Moldt, and H. Rolke, "Modeling mobility and mobile agents using nets within nets." In W.M.P. van der Aslst and E. Best, editors, *Proc. of Int. Conf. on Applications and Theory of Petri Nets, LNCS*, vol. 2769, pp. 121-139, June 2003

[KR01] M. Kohler, H. Rolke, "Towards a Unified Approach for Modeling and Verification of Multi Agent Systems," *Workshop on Modelling of Objects, Components, and Agents (MOCA'01)*, Daniel Moldt, Ed., pp. 85-104. DAIMI PB-553, Aarhus University, August 2001

[Kum98] O. Kummer, "Simmulating synchronous channels and net instance." In J. Desel, P. Kemper, E. Kindler, and A. Oberweis, Eds., 5. *Workshop Algorithmen und Werkzeuge für Petrinetze*, Forschungsbericht Nr. 694, pp. 73-78. Fachbereich Informatik, Universität Dortmund, October 1998

[Kum99] O. Kummer, and F. Wienberg, "Renew – the reference net workshop," http://www.renew.de, 1999

[LL99] X. Li and J. Lilius, "Verifying Time Petri Nets by Linear Programming", *Research report, Number TUCS-TR-259*, Turku Centre for Computer Science, Finland, March 31, 1999

[Mas99] C. Mascolo, "Mobis: A Specification Language for Mobile Systems," *Proc. Third Int'l Conf. Coordination Models and Languages*, pp. 37-52, April 1999

[McM92] K. L. McMillan, "Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits," *CAV'92, LNCS*, vol. 663, pp. 164-174, 1992

[Mcm93] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, 1993

[Mil93] R. Milner, "The Polyadic π-Calculus: a Tutorial," In *Logic and Algebra of Specification*, Hamer, Brauer, and Schwichtenberg, Eds., Spring-Verlag, Berlin, pp. 1-49, 1993

[Mil99] R. Milner, "*Communicating and Mobile Systems: The π-Calculus*." Cambridge University Press, New York, 1999

[MK96] T. Miyamoto, and S. Kumagai, "A Multi Agent Net Model of Autonomous Distributed Systems", In *Proceedings of CESA'96, Symposium on Discrete Events and Manufacturing Systems*, pp. 619-623, 1996

[MKG98] J. Magee, J. Kramer, and D. Giannakopoulou, "Software Architecture Directed Behaviour Analysis," in *Proc. of the Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9)*, pp. 144-146, Ise-shima, Japan, April 1998

[MR98] P. J. McCann, G.C. Roman, "Compositional Programming Abstractions for Mobile Computing." *IEEE Trans. on Software Engineering*, vol. 24, no. 2, pp. 97-110, 1998

[MW97] D. Moldt, and F. Wienberg, "Multi-Agent-System based on Colored Petri Nets", *LNCS*, vol. 1248 1997

[Mur89] T. Murata, "Petri Nets: Properties, Analysis and Applications." *Proceedings of the IEEE*, vol.77, no.4, pp. 541-580, 1989

[Obj97] ObjectSpace, Inc. "ObjectSpace Voyager Core Package Technical Overview", Technical Report, ObjectSpace, Inc., July 1997

[OMG98] OMG, MASIF—Mobile Agent System Interoperability Facility, *Technical Report*, OMG, 1998

[OMG02] OMG, *CORBA™/IIOP™ Specification*, http://www.omg.org/technology/documents/formal/corba_iiop.htm, 2002

[PMH02]P.J. Pingree, E. Mikk, G.J. Holzmann, M.H. Smith, and D. Dams, "Validation of mission critical software design and implementation using model checking," *Proceedings of Digital Avionics Systems Conference*, vol. 1, pp.1-12, October 2002

[RB02] B.F. Rodak, W.B. Saunders, *Hematology: Clinical Principles & Applications (2nd Edition)*, Elsevier-Health Sciences Division, January 2002

[RMP97] G.C. Roman, P. J. McCann, and J.Y. Plun, "Mobile UNITY: Reasoning and Specification in Mobile Computing," In *ACM Trans. Software Engineering and Methodology*, vol. 6, no.3, pp. 250-282, 1997

[SG96] M. Shaw and D. Garlan, *Software Architecture: Perspectives on Emerging Discipline*, Prentice Hall, 1996.

[Sha01] M. Shaw, "The coming-of-age of software architecture research." In *Proceedings of International Conference on Software Engineering*. Toronto, pp. 656-664, 2001

[Sha03] H.M. Shapiro, *Practical Flow Cytometry*, Wiley-Liss, March 2003

[SM98] G.D. Serugendo, M. Muhugusa, et al. "A Survey of Theories for Mobile Agents," In *World Wide Web Journal*, special issue on distributed World Wide Web processing, applications and techniques of web agents, 1998

[Tsc96] C.F. Tschudin, "The Messenger Environment M0 – A Condensed Description," *LNCS*, J. Vitek and C. Tschudin, Eds., Vol. 1222, pp. 149-156, 1996

[Val78] R.G. Valk, "Self-Modifying Nets: A Natural Extension of Petri Nets," *Proc. Int'l Colloquium Automata, Languages, and Programming (ICALP '78)*, pp. 464-476, 1978.

[Val98] R. Valk, "Petri Nets as Toekn Objects, An Introduction to Elementary Object Nets, " In Jorg Desel, Eds., *19th International Conference on Application and Theory of Petri Nets, LNCS*, vol. 1420, Berlin, 1998

[Val99] R. Valk, "Concurrency in Communication Object Petri Nets," In F. DeCindio, G.A. Agha, and G. Rozenberg, eds., *Concurrent Object-Oriented Programming and Petri Nets. LNCS*, 1999

[Wan98] J. Wang, *Timed Petri Nets: Theory and Application*, Kluwer Academic Publishers, 1998.

[WD99] J. Wang and Y. Deng, "Incremental Modeling and Verification of Flexible Manufacturing System", *Journal of intelligent Manufacturing*, no 4, 1999

[WHD99] J. Wang, X. He, and Y. Deng, "Introducing Software Architecture Specification and Analysis in SAM through an Example," *Information and Software Technology*, 41 (7), pp. 451–467, 1999

[Whi95] J.E. White, "Mobile Agents." *Technical Report*, General Magic, Inc., October 1995

[XD00] D. Xu, and Y. Deng, "Modeling Mobile Agent Systems with High Level Petri Nets", In *Proc. Of IEEE International Conference on Systems, Man, and Cybernetics (SMC'00)*, pp. 3177-3182, Nashville, October 2000.

[XS03] H. Xu, S.M. Shatz, "A Framework for Model-Based Design of Agent-Oriented Software", *IEEE Trans. on Software Engineering*, vol. 29, no. 1, pp. 15-30, January 2003

[XYD03] D. Xu, J. Yin, Y. Deng and J. Ding, "A Formal Architecture Model for Logical Agent Mobility," *IEEE Trans. on Software Engineering*. vol.29, no. 1, pp. 31-45, January 2003

[YM97] J. Yang, and A. K. Mok, "Symbolic Model Checking for Event-Driven Real-Time Systems", *ACM Transitions on Programming Languages and Systems*, vol. 19, no. 2, March 1997

[YMW93] J. Yang, A. K. Mok, and F. Wang, "Symbolic model checking for event-driven real-time systems", In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, New York, 1993

VITA

JUNHUA DING

October 20, 1971	Born, Anhui, China
1986 – 1990	Geochemistry Nanjing School of Geosciences Nanjing, China
1990 – 1994	B.E., Computer Engineering China University of Geosciences Wuhan, China
1994 – 1997	M.E., Computer Science Nanjing University Nanjing, China
1997 – 1998	Software Engineer State Key Laboratory for Novel Software Technology Nanjing, China
1998 – 2000	M.S., Computer Science Florida International University Miami, Florida
2000 – Present	Doctorate in Computer Science Florida International University Miami, Florida Software Engineer Beckman Coulter Inc. Miami, Florida

PUBLICATIONS

J. Ding, S. Sun, D. Yang, and J. Lv, "Consistency of Multi-view Requirement Definitions and Their Verifications", *Computer Research & Development*, March 1998

J. Ding, J. Lv, "Researches on Software Interoperability", *Computer Research & Development*, October 1998

J. Ding, H. Dong, J. Lv, "Researches on the Models and Languages of Application Framework-based Interoperability", In *Proceedings of the 7th National Conference of Young Computer Scientists (NCYCS '98)*, Shanghai, China, October 1998

J. Ding, H. Dong, D. Wu, and J. Lv, "Software Interoperability: A Comparison Study of CORBA and Other Approaches", *Computer Research and Development*, vol. 35(7), pp.577-583, December 1998

H. Dong, J. Ding, J. Lv, "Researches on Open Communication Frameworks for Software Agents", *Proceedings of the 7th National Conference of Young Computer Scientists (NCYCS '98)*, Shanghai, China, October 1998

X. He, J. Ding, and Y. Deng: "Analyzing SAM Architectural Specifications Using Model Checking", *SEKE2002*, Italy, July 15-19, 2002

X. He, H. Yu, T. Shi, J. Ding, and Y. Deng: "Formally Specifying and Analyzing Software Architectural Specifications Using SAM", *Journal of Systems and Software*, vol. (71), pp. 11-29, 2004

X. Li, H. Dong, J. Ding, J. Lv, "Security of Software Agents", *Computer Science*, May 1998

D. Yang, J. Ding, J. Lv, "The Researches on Dictionary Management Methods in Software Requirement Analysis Automation Systems", *Computer Software & Application*, December 1998

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, pp. 31-45, January 2003