

Modeling and Analyzing a Mobile Agent-based Clinical Information System

Junhua Ding, Dianxiang Xu, Xudong He, and Yi Deng

Abstract — This paper presents an approach for modeling and model-checking a mobile agent system specified by LAM, which is a two-layer formal method for characterizing logical agent mobility using Predicate/Transition (PrT) nets. Based on the transformation of PrT nets into input programs of the model checker SPIN, we model check a variety of properties with respect to agents, logical agent mobility, agent environments, and system interaction in a mobile agent system model. We demonstrate our approach through a case study on the modeling and analysis of a mobile agent-based clinical information system.

Index Terms—Mobile Agent, Predicate/Transition Nets, Model Checking, SPIN

1. INTRODUCTION

A well designed software architecture can provide a better understanding of system structure and reduce overall development cost with the help of detecting and eliminating design errors early in the development cycle [19]. Modeling and analyzing mobile agent systems at architectural level is a challenge due to the logical mobility of agents, which are executing programs that migrate from machine to machine in a heterogeneous network [7]. Since mobile agents may execute at different locations during their life spans, most of previous mobile agent models have characterized agent mobility as change of value of the location attribute (refer to [23] for a literature review on the formal modeling of mobile agents). This approach is inadequate for capturing the dynamic nature of mobile agent systems because of the failure in addressing issues related to agent migration and transfer. Agent migration comprises a sequence of activities, such as issuing a request of migration (either by the agent itself or by the environment), suspending the agent execution, transmitting the agent (code and/or state), and resuming the agent execution. The capability of migration reflects the essential difference between logical mobility of software agents and physical mobility of mobile nodes.

To address this problem, we have developed the two-layer approach, LAM (Logic Agent Mobility) [23], to formally modeling logical agent mobility based on Predicate/Transition (PrT) nets. The semantics of a two-layer LAM model is

defined by transforming it into a complete PrT net. System properties are then verified through reachability analysis of PrT nets. While the reachability analysis of LAM models offers an important way of system verification, it is inefficient for complex mobile agent systems because of larger state space.

In order to better understanding the design and behaviors of distributed systems based on mobile agents, providing a practical but rigorous analysis approach for analyzing this type of systems is necessary. In addition, there are few works on real-world application of mobile agents. Therefore, formally modeling and analyzing a real-world mobile agent application would greatly help us to develop and research mobile agent systems.

In this paper, first we use LAM to model a mobile agent-based clinical information system -- MACIS, which was proposed to solve a real-world problem. Then we present a model-checking approach to verifying LAM models. Model checking verifies a system model against correctness criteria by an efficient procedure for characterizing all possible executions and exploring all behaviors exhaustively [11]. In our previous work, we successfully used the model checker SMV [2] found an error in an FMS model [9], [24], [21]. Model checkers can deal with very large state space (up to 10^{100} or more states), although some may suffer from the state-space explosion problem for systems with a large number of parallel processes [5], [8], [15], like large scale mobile agent systems. Therefore, we proposed a hierarchical analysis method to analyze a LAM model in order to avoiding the state space explosion problem. We have chosen SPIN [11], an explicit verification model checker that uses partial order to reduce state-space, as the model checking tool since its modeling language is more convenient and its performance is higher comparing with SMV for checking large software model. The fundamental idea of our approach is model checking PrT nets using SPIN. This is accomplished by transforming PrT nets and properties of interest into SPIN input programs. Therefore, we can check a variety of properties about agents, agent mobility, agent environment, and system interaction against LAM models for mobile agent systems.

The contributions of this research work include: 1) Modeling a mobile agent system using two-layer PrT nets (i.e. LAM), which illustrates the approach to modeling the dynamic cooperation and interaction behaviors between mobile agents and their environment. 2) Model checking the LAM models of a mobile agent system based on a hierarchical analysis method.

Junhua Ding is with the Beckman Coulter Inc., Miami, FL 33196 USA (phone: 305-380-3193; fax: 305-380-3605; e-mail: jding01@cs.fiu.edu).

Dianxiang Xu is with the Computer Science Department, North Dakota State University, Fargo, ND 58105 USA (e-mail: dianxiang.xu@ndsu.nodak.edu).

Xudong He, and Yi Deng are with the School of Computer Science, Florida International University, Miami, FL 33199 USA (e-mail: {hex, deng}@cs.fiu.edu).

This paper not only presents a method to analyze a LAM using model checking tool SPIN, but also discusses how to check the dynamic properties of mobile agent system via analyzing the interaction properties. 3) Designing a real-world application of mobile agents, which helps us to better understanding mobile agent systems, and therefore to develop high quality mobile agent applications.

The rest of this paper is organized as follows. To make the paper self-contained, section 2 is a brief overview of the LAM model for logical agent mobility. Section 3 introduces the key idea of checking LAM models. Section 4 presents the LAM model for MACIS, a mobile agent-based clinical information system. Section 5 discusses how to check the LAM model of MACIS. Section 6 concludes the paper.

2. LAM: AN OVERVIEW

LAM is a two-layer approach for modeling logic agent mobility using PrT nets [6] as the formal foundation. In LAM, a mobile agent system is modeled by a set of agent spaces (components) and external connectors. Each component, as a container or wrapper of a group of mobile agents, is made up of an environmental part and an internal connector. Mobile agents can migrate from one component to another. External connectors model the interaction among components. PrT nets are used to model the behaviors of the environments, mobile agents, as well as internal/external connectors. The nets are called system nets, agent nets, internal/external connector nets, respectively. The internal connector is dynamically configured so that a changing number of agents can be bound to the environment. Agents are allowed to be active at specific places of system nets. These places indicate the running environments that host mobile agents. Transitions in an agent net are disabled whenever the agent is inactive (i.e. in the process of migration). In this case, transitions in the agent net are all disabled. As part of tokens, an agent may move from one place to another and thus change its location when transitions in its hosting system net are fired.

LAM provides a generic way to model agent behaviors in terms of the object-oriented viewpoint of agents. An agent is as an encapsulated entity, consisting of its interface, behavior, and state. By encapsulation, we mean that the state of an agent can be changed only by its own methods. An agent is also an interactive object capable of receiving messages from and sending messages to other agents/objects. In the meantime, it has its own state and some methods for message processing which leads to state change. Formally, the interface, the behavior, and the 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.

A component consists of an environmental part modeled by a system net and an internal connector net for binding agents to the environment. The environment provides facilities for agent mobility (e.g. execution place, activation and deactivation). Components are connected through external

connectors, and arcs of connector nets are supposed to be properly labeled so that a migrating agent is always transferred to a single destination (agent cloning is not considered in LAM). The semantics of a LAM model is defined by transforming the model into a complete PrT net. This lays a theoretical foundation for system analysis.

In the following, we briefly describe the PrT nets used in the LAM model and in this paper. A PrT net is a tuple $(P, T, F, \Sigma, L, \varphi, M_0)$ [23], 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 $\langle \phi \rangle$.
4. φ is a mapping from transitions to a set of inscription formulae. 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 . Variables occurring free in a formula have to occur at an adjacent input arc of the transition.
5. M_0 is the initial or current marking: $M_0 = \bigcup_{p \in P} M_0(p)$.

$M_0(p)$ 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 .

Let $\bullet t = \{p \in P: (p, t) \in F\}$ and $t^\bullet = \{p \in P: (t, p) \in F\}$ be the precondition predicates and the post-condition predicates of transition 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 w.r.t. θ , where l/θ yields a token by substituting all variables in label l with the corresponding bound values w.r.t. θ . The firing of an 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 $p \in t^\bullet$. After the firing of t , we get a new marking M' . Formally, $M_1(p) = M_0(p) - \{l/\theta: l \in L(p, t)\}$ for any $p \in \bullet t$, and $M_1(p) = M_0(p) \cup \{l/\theta: l \in L(t, p)\}$ for any $p \in t^\bullet$. We denote a firing/occurrence sequence as $M_0[t_1\theta_1 > M_1[t_2\theta_2 > M_2 \dots [t_n\theta_n > M_n$, or simply $t_1\theta_1 t_2\theta_2 \dots t_n\theta_n$, 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. A marking

M is said to be reachable from M_0 if there is such a firing sequence that transforms M_0 to M .

Since a PrT net is a structure, and a token in a PrT net may carry structured data, a PrT net (an agent net in LAM) may be packed up as part of a token in another PrT net (a system net in LAM). Additional constraints may be imposed on the enabledness of transitions in a net. For example, to enable the transitions in an agent net, which is part of tokens in a system net, the agent must be located at the special place that models the running environment of mobile agents.

3. MODEL CHECKING LAM MODELS

Model checking is the process of verifying the abstract representation of a system (a model) using state exploration. There are various algorithms used to search the state space generated by the model to best deal with the state explosion problem [11], [12], [22]. Model checking consists of the following tasks: (1) *modeling* – convert the design into a formalism accepted by the model checking tool, (2) *specification* – specify the properties that the design must satisfy, usually written using logical formulas e.g., linear temporal logic (LTL) or computation tree logic (CTL), and (3) *verification* – uses a state exploration algorithm to search the state space for inconsistencies in the specification. The atomic propositions or system properties in this paper are written using LTL formulas. The basic operators of LTL include: $\mathbf{X}p$ – next time p holds, $\Box p$ – p holds globally after any number of steps, and $p \mathbf{U} q$: p holds until q holds, $\Diamond p$ – p will hold in the future. If p, q is LTL formula, then so $\neg p, p \vee q, \mathbf{X}p, \Diamond p, \Box p, p \mathbf{U} q$ are LTL formulae.

3.1 Model Checking a LAM Model

A LAM model can be checked at different levels, such as component level, system level, and interaction level. The common basis is model-checking PrT nets.

System level analysis treats agent nets as regular data or tokens within places of system nets. Since all system nets are statically defined, the composition of system nets is to connect them together with external connectors, which results in a whole net representing the mobile agent system. The analysis is therefore applied to this complete PrT net. Component level analysis is to analyze individual agent models, component models or connectors. When we analyze an agent net, its environment or agent system is abstracted as interfaces, which send messages to or receive messages from agents. When we analyze the interaction between agents and their environment, we use interaction analysis. For interaction analysis, agents in the system net are unfolded into agent nets, and the system net is connected with the agent nets and the internal connectors to form a complete PrT net. The analysis is applied to the composed PrT net. We do not analyze connectors separately; instead they are integrated with component models (internal connectors) or system level models (external connectors).

When we analyze a system net, the external connectors are

reduced to a subnet that receives tokens from and sends tokens to the system net. The internal connector is connected with the system net considering a certain interaction scenario. When we analyze an agent net, we treat the agent net as a regular PrT net. An agent net only makes sense when it is within a system net, so we abstract the system net as interfaces to the agent net of interest.

3.2 Model Checking a PrT Net

The key idea of analyzing PrT nets using model checker SPIN is to transform PrT nets into SPIN inputs (i.e. Promela programs) and define system properties of interest as assertions or LTL [4] claims (e.g. *never* claim, *accept-state* labels or other assertion labels such as *basic assertion*, *end-state* labels, *progress-state* labels, and *trace assertions*) in Promela programs. More information on SPIN and Promela can be found in [11]. Since a Promela program allows for parallel processes, we can convert several individual nets into a single Promela program, where each net is corresponding to a process.

The algorithm for the transformation is as follows:

Step 1: For each place in a PrT net, define a corresponding Promela variable.

We assume that PrT nets in a LAM model are k -bounded (i.e. for any place p , $M(p) \leq k$, $M_0 \geq M$), where k is a constant (k can be predefined according to system requirements). Therefore, each variable is of an enumerate type.

Step 2. Define the initial state.

In the *init* process of the Promela program, each variable is initialized with the corresponding initial marking of the place in the PrT net. Considering that we often need to verify a Petri net with different initial markings, we take advantage of the *init* process of Promela, which can start a process with different inputs automatically.

Step 3. Define transition relations

Each transition in a net is defined as an *atomic* statement within the corresponding process for the net. The *atomic* statement consists of a sequence of *case* statements, and each 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 the *case* statements for each transition is the all possible input of the inscription expressions of the input arcs of the transition. So each *atomic* statement essentially indicates the firing rule of the transition. Global variables and channel variables are used to synchronize and communicate between processes.

3.3 Translating a LAM Model into a Promela Program

Now we describe how to transform a LAM model into a Promela program. If we only analyze an agent net or a system net in the LAM model, we translate the agent net or system net into a Promela program with only one process besides *init*: agent process or system process. For the whole LAM model, each component or agent has a corresponding process in the Promela program. For those components that share a common structure with different identities, we create a generic process,

and instantiate the generic process for each component with its identity in the *init* process.

In addition to the agent nets and system nets in a LAM model, we have to deal with the internal connectors, external connectors, and those predicates that model agent running contexts:

1) *Internal connector*: The internal connectors are translated into channel variables in the Promela program, and the channel variables control the synchronization between multiple agents and the system. A global variable representing p_g is used to decide the connection between an agent net and its system net in the Promela program.

2) *External connector*: Each component connects to all other available components in the network. The tokens or agents from the output place such as $SN_1 p_{ex-out}$ of one component are sent to the destination component input place such as $SN_n p_{ex-in}$. In Promela program, $SN_n p_{ex-in}$ and $SN_1 p_{ex-out}$ are defined as corresponding global variables. When $SN_1 p_{ex-out}$ gets a token or a positive value, then $SN_n p_{ex-in}$ will get the token or value if the agent's destination is SN_n . Each mobile agent has an itinerary, which decides the visiting path of the agent. In Promela program, the itinerary is implemented as a list of locations of the agent process, and agent location is dynamically changed at run time according to pre-defined itinerary and running results.

3) P_w and P_g : In LAM, P_w contains agent identifiers and their nets. In the Promela program, we represent it by an integer array (i.e. we use integer to identify an agent) and the entries of the array represent the mobile agents in P_w . Agent type is defined when it is started with particular agent process. P_g is the global variable indicating whether an agent is active or inactive in the component. In the Promela program, the component process uses component identifier, agent identifier, and agent itinerary to decide values of its P_w and P_g (each component has its own unique global variables p_w and p_g).

A Promela program translated from a simplified agent net is list in Appendix I.

4. MODELING MACIS

In this section, we describe the modeling of a mobile agent-based Clinical Information System (CIS), which processes information on human blood cells. Clients of CIS are researchers who retrieve and process medical data from two different databases: one is for the cytometry analysis [18] data, and the other is for the hematology analysis [17] data. Some researchers need to first process data from one database, and then process data from the other based on previous results.

The CIS clients apply different algorithms and protocols to process huge number of samples data. Each analysis software package or agent includes programs, algorithms and protocols. Protocols are used to select data and analysis parameters, algorithms are used to process data, and programs are used to run algorithms and protocols. Servers are located at the laboratories that provide blood cell analysis data. For each

sample, the servers analyze as many parameters as possible in order to reuse samples and reduce cost. There are two traditional ways to process these data: 1) Copying data from servers to the local site and process data in the local site. 2) Installing algorithms and programs in the sever sides, and then processing data using the client-server approach.

The traditional design for a CIS would suffer from following difficulties. 1) It is impractical for researchers to save all data from different databases to their local hosts and then keep updating. 2) It is hardly feasible for the servers to install all possible analysis software for different clients. But users have to frequently change their protocols and algorithms for processing data according to the intermediate computation results. 3) Users have to manually integrate different databases during the computation. Therefore, it is difficult to automatically process a task.

We propose to apply Mobile Agent technique to the development of a CIS (hence called MACIS). In MACIS, clients send agents with particular algorithms and protocols to servers. These agents run locally on servers, move to different servers to access databases based on their intermediate results, and then deliver results to clients when they finish their tasks. 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. Each server site has an agent system, which is supplied by clients and configured by servers. Each server has an interface that is responsible for database access. Use of mobile agents in MACIS results in a number of advantages, including offline computation, reduction of cost in data transferring and maintenance, synchronization with up-to-date data resources, flexible use of different services, reduction of network traffic, etc.

To facilitate illustration, the LAM model of MACIS used throughout this paper includes three agent systems: one for client, and the other two for cytometry and hematology servers. The static structure of MACIS is $(COMP, CONN)$, where $COMP = \{(CM_1, SN_1, ICN_1), (CM_2, SN_2, ICN_2), (CM_3, SN_3, ICN_3)\}$, CM_i ($1 \leq i \leq 3$) are the agent system for clients, hematology server, and cytometry server, SN_i ($1 \leq i \leq 3$) are the system nets, and ICN_i ($1 \leq i \leq 3$) are the internal connectors for system nets, $CONN = \{CN\}$, and CN is the connector among CM_1 , CM_2 , and CM_3 . Agent AN is created on CM_1 , and then it is sent to CM_2 or CM_3 . The agent roams among servers CM_1 , CM_2 , and CM_3 , and finally returns to CM_1 with results.

4.1 Modeling the Agent

An agent in MACIS only communicates with its system (the host system). It sends retrieving requests to a host according to protocols. Then the host searches its database and sends back data. As soon as the agent gets the data, it processes the data and saves the intermediate results. The agent net is shown in Fig. 1, and the labels of the net are explained in Table 1. Tokens in the agent net have the structure: $\langle dl, da, sl, sa, type, cmd, msg \rangle$, where dl , da , sl , and sa means destination host, destination agent, source host, and source agent, respectively. $type$ means that the message is an agent (AN) or a regular

message (*MSG*). *cmd* is some predefined command, such as *RST*, *MOVE*, and *STOP*. *RSP* means that the message is a result, *MOVE* refers to a request of migration, *STOP* indicates a request to stop running the agent. *cmd* is an agent identifier if message type is *AN*. In Fig. 1, *cl* is the current system location, *p_{in}* and *p_{out}* are places which connect to the internal connector of the current system, *kb* is the knowledge base that includes algorithms and protocols, *rst* is the running results, and *pt* is the itinerary.

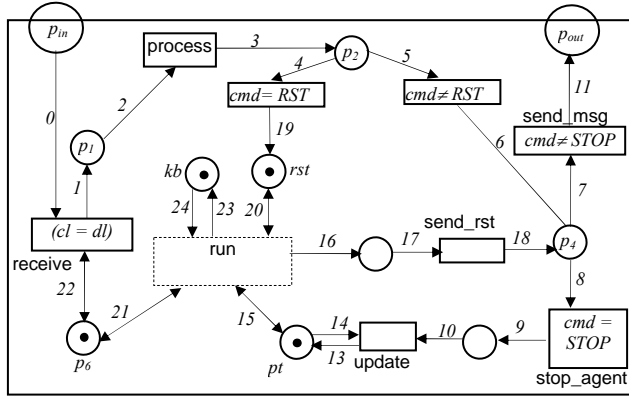


Fig. 1 An agent net in MACIS

Table 1 Legend of Fig. 1

Symbol	Description
0, 1, 2, 3	$\langle dl, DA, dl, \phi, MSG, cmd, msg \rangle$, and $cmd \in \{STOP, RST, MOVE\}$
4, 19, 20	$\langle dl, DA, dl, \phi, MSG, RST, msg \rangle$
5, 6, 7, 11, 16, 17, 18	$\langle dl, DA, dl, \phi, MSG, cmd, msg \rangle$
8	$\langle dl, DA, dl, \phi, MSG, STOP, \phi \rangle$
9, 10, 21, 22	$\langle \phi \rangle$
13, 14	$\langle next \rangle$, $\langle next = next + 1 \rangle$
15	$\langle nl \rangle$, $nl = cur(pt)$, <i>pt</i> is the itinerary, and <i>cur</i> is the reference pointing to the current location or the next destination if it is moving out.
23, 24	$\langle ref, s \rangle$, $\langle ref = (ref + 1)/M, s \rangle$, <i>M</i> is the number of total statements in <i>kb</i> .

The transition *receive* receives messages from place *P_{in}*, and it only receives data from the current system *CM* (its location is *cl*). The transition *process* forwards the data to its output place. If the message is the result data from the server, it is sent to the agent for processing, and the result is saved at the place *rst*. The transition *run* models data processing by the agent 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 *kb* consists of a sequence of statements. It has a property *ref*, which points to the current statement *s*. The statement *s* is structured data, which consists of its type and expression or data obj, i.e. $s = \langle TYPE, obj \rangle$. There are three different types for *s*: *MSG*, *STOP* and *MOVE*. *ref* means moving to next statement when the transition *run* fires. The agent's itinerary is defined with predicate *pt*, which is updated when the transition *update* fires. In predicate *pt*, the *next* attribute points to the next destination that the agent intends to move to. Transition *send_msg* is used to send messages to the current host and

transition *stop_agent* is used to update agent itinerary and stop running.

4.2 Modeling the Environment

The system net for clients is shown in Fig.2 and Table 2. The symbols in Fig.2 and Table 2 have same meaning as that in above section. Predicate *cb* is the set of templates of agent nets. Each token in *cb* is a template of agent nets. We use transition *create* to generate agents, so that it outputs instances of agent templates to place *p_w*. When an agent returns to its home with results, 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 *db*. For simplicity, we do not model agent behaviors after it delivers results.

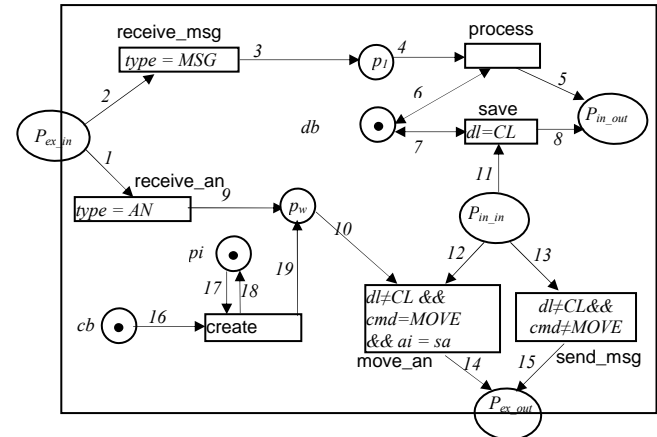


Fig. 2 A system net for clients in MACIS

Table 2, Legend of Fig. 2

Symbol	Description
<i>p_w</i>	The place of mobile agents stay in, $\langle CL, \phi, CL, sa, AN, ai, MN \rangle$, and $sa = ai$.
<i>p₁</i>	The incoming message $\langle CL, da, sl, sa, MSG, cmd, msg \rangle$
<i>P_{ex_in}</i>	The incoming message from other host nets $\langle CL, da, sl, sa, type, cmd, msg \rangle$
<i>P_{ex_out}</i>	The outgoing message to other host nets $\langle dl, da, CL, sa, type, cmd, msg \rangle$
<i>P_{in_out}</i>	The message from the host net to an agent $\langle CL, da, CL, \phi, MSG, cmd, msg \rangle$
<i>P_{in_in}</i>	The message from agents the host net $\langle dl, \phi, CL, sa, type, cmd, msg \rangle$
<i>cb</i>	Agent net templates, $\{MN\}$
<i>pi</i>	Agent identifier, $\{ai\}$
<i>db</i>	Database for save results
1, 2	$\langle CL, \phi, sl, sa, type, cmd, msg \rangle$
3, 4, 11, 12, 13	$\langle CL, \phi, sl, sa, MSG, cmd, msg \rangle$
5, 8	$\langle CL, da, CL, \phi, MSG, cmd, msg \rangle$
6, 7	$\langle cmd, msg \rangle$
9, 19	$\langle CL, \phi, sl, ai, AN, ai, MN \rangle$, $\langle CL, \phi, CL, ai, AN, ai, MN \rangle$
14, 15	$\langle dl, \phi, CL, ai, AN, ai, MN \rangle$, $\langle dl, \phi, CL, \phi, MSG, cmd, msg \rangle$
16	$\langle MN \rangle$, <i>MN</i> is an agent net
10, 17, 18	$\langle ai \rangle$, $\langle ai \rangle$, $\langle ai + 1 \rangle$

The system net for server 1 (for hematology analysis data, location is *LBI*) is shown in Fig.3. The legend is same as that for Fig. 2 except the location in Fig. 3 is *LBI*, other than *CL* in Fig. 3. Predicate *db* represents the database with hematology data. The transition *retrieve* is used to select data from the

database according to statements from agents. The system net for server 2 (for cytometry analysis data) is the same as that for server 1 except that the location is different (the location is *LB1* in server 1, and *LB2* in server 2).

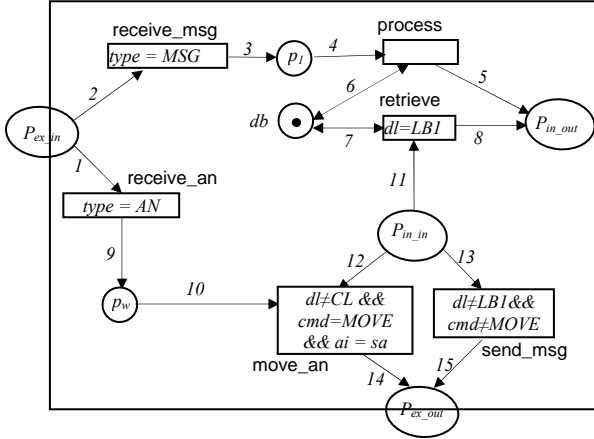


Fig. 3 A system net for servers in MACIS

External connector *CN* is used to establish interaction among *CM₁*, *CM₂* and *CM₃*. Internal connector *ICN_i* is used to facilitate the communication between agents and their system nets. Internal connector nets *ICN* and the external connector *CN* follow the definition of the internal connector in LAM.

5. MODEL CHECKING MACIS

After we modeled a mobile agent system using LAM, analyzing the correctness of the models is necessary. One of the purposes of formally modeling a system is to facilitate the formal analysis, which can guarantee the correctness of the model so that to improve the quality of the system. Model checking is a wide used practical formal analysis method due to its effectiveness, efficiency and automatic procedures. Model checker SPIN is more suitable to analyze large software systems. This section presents the use of SPIN for checking the MACIS models. For simplicity, we consider the scenario with only one client agent: the agent starts from *CM₁* and moves to *CM₂* and *CM₃* to search and process hematology and cytometry information in each server, and returned back to *CM₁* with results as soon it completes its tasks. We checked several important properties about the agent, hosts, as well as the interaction. To any agent or host system, deadlock-free is the basic requirement, therefore, this property has to be checked. In following each subsection, a brief explanation of the checking properties is given.

5.1 Model Checking the Agent Net

For an agent, receiving and processing incoming messages is a basic function, and stopping its running and moving itself to the destination are the essential functions of an agent. Therefore, we checked the properties as deadlock-freedom and reachability. For convenience, the token structure in the agent net is simplified to $\langle dl, da, cmd \rangle$. In the agent net, regular messages are the only type of messages although *cmd* can be

STOP or other commands.

The properties include: 1) If the agent net receives a message from its input place *P_{in}*, eventually, this message will reach *p₂* (processed by the agent); 2) If the agent receive a *STOP* command, it updates the agent's location, which means *dl* become *dl + 1*, and *cl = cl + 1*. The transition *run* updates *cl* to *cl + 1* and sends a message $\langle dl + 1, \phi, MOVE \rangle$ to the current host net to move out this agent. These properties are first translated into LTL formula, and then coded into *never* claims automatically. Formally, the first property is $\diamond(p2.dl != \phi)$, and its negated formula is $\neg(\diamond(p2.dl != \phi))$. Therefore, checking the *never* claim means: $\neg(\neg(\diamond(p2.dl != \phi)))$ or $\diamond(p2.dl != \phi)$. The second property is: $\diamond(p6.dl == p4.dl) \ \&\& \ (p4.dl != \phi)$.

Part of the running results of checking this property is given below:

- 1) (Spin Version 4.0.7 -- 1 August 2003)
- 2) + Partial Order Reduction
- 3) Full statespace search for:
- 4) never claim +
- 5) assertion violations + (if within scope of claim)
- 6) acceptance cycles + (fairness enabled)
- 7) invalid end states - (disabled by never claim)
- 8) State-vector 48 byte, depth reached 33, errors: 0
- 9) 16 states, stored (64 visited)
- 10) 50 states, matched
- 11) 114 transitions (= visited+matched)
- 12) 7 atomic steps

The line numbers are added for the purposes of illustration. Line 2 indicates that partial order reduction algorithm was used. The symbol '+' (or '-') in lines 4 to 7 means that the corresponding property was (or was not) checked. From line 8, we know that the complete description of a single global system state required 48 bytes of memory, the longest depth-first search path contained 33 transitions from the root of the tree, and no error was found (i.e. the property $\diamond(p2.dl != \phi)$ holds). Line 9 says that a total of 16 unique global system states were stored in the state space. Line 10 indicates that in 50 cases the search returned to a previously visited state in the search tree. Line 11 indicates that a total of 114 transitions were explored in the search. Line 12 shows that 7 transitions were part of atomic sequences [11].

5.2 Model Checking the System Net

Each host system needs to correctly receive and process incoming messages or agents, start the running of an incoming agent, and stop the running of an agent and move it out. For simplicity but not lost generality, we only show how to check properties on receiving incoming messages and stopping the running of the agent. For convenience, the token structure is simplified to $\langle dl, type, cmd \rangle$. 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 *MOVE*.

The properties checked are: 1) If the host net receives a

message or an agent from its external input place P_{ex-in} , eventually, this message or agent will reach p_l or p_w . The LTL property for receiving agents is: $((p_0.type == AN) \ \&\& \ (p_0.dl == LB1) \rightarrow \diamond(p_3.type = AN) \ \&\& \ (p_3.dl == LB1))$, where p_0 represents P_{ex-in} , and p_3 is p_w . 2) If the message is a *MOVE* command $\langle dl + 1, MSG, MOVE \rangle$, the token in p_w will be moved to place P_{ex-out} (we only consider one agent in this model, so we do not need to compare agent identifiers). The LTL property is: $((p_0.type == MSG) \ \&\& \ (p_0.cmd == MOV) \rightarrow \diamond(p_8.type = AN) \ \&\& \ (p_8.dl == LB2))$, where p_8 represents P_{ex-out} .

Part of the running results of checking one property is given below:

```
Full statespace search for:
  never claim          +
  assertion violations + (if within scope
of claim)
  acceptance cycles + (fairness enabled)
  State-vector 52 byte, depth reached 48,
errors: 0
```

The interpretation is similar to that in subsection A. It shows that property $((p_0.type == AN) \ \&\& \ (p_0.dl == LB1) \rightarrow \diamond(p_3.type = AN) \ \&\& \ (p_3.dl == LB1))$ holds.

5.3 Model Checking the System Interaction

An agent is running within a host system, and an agent net is wrapped as a token in a host net. One of the contributions of LAM is to facilitate modeling the interaction between agents and their host systems. Checking the interaction between two layer nets is a challenge work, and the checking method was discussed in section III. Therefore, we need to check whether a request from an agent to its host or a host to the agents is realized correctly. In order to check the interaction properties, the agent net, system nets, and internal connectors are composed as a whole model. The token structure is $\langle dl, da, type, cmd \rangle$, where dl is the location of current host and the agent (the first location is *LB1*), da is the agent identifier, $type$ is the message type: agents or regular messages, and the cmd could be *STOP*, *MOVE* or agent identifier *AI* if the message is an agent.

One interaction property was checked: if the host net receives a move out command, and it has an agent in place p_w , then the agent is stopped. The LTL formula is: $((p_4.cmd == MOV) \ \&\& \ (p_4.type == MSG) \ \&\& \ (p_6.type == AN) \ \&\& \ (p_6.cmd == AI) \rightarrow \diamond((p_{12}.da == DA) \ \&\& \ (p_{12}.cmd == STOP)))$, where p_4 is P_{in-in} , p_6 is in p_w the system net, and p_{12} is p_4 in the agent net, *AI* and *DA* is agent ID.

Part of the running results of checking this property is given below:

```
Full statespace search for:
  never claim          +
  assertion violations + (if within scope
of claim)
  acceptance cycles + (fairness enabled)
  State-vector 48 byte, depth reached 15,
errors: 0
```

The interpretation is similar to that in subsection A. It shows that property holds.

5.4 Model Checking the Mobility

The most important character of a mobile agent system is an agent can move from one host to another in a network. Therefore, checking the mobility is necessary. If an agent is moved from one host to another, it has to be stopped in the current host, and then requests its host to move it out to the destination host, and eventually it is started in the destination host after it arrives. We first compose the host nets and external connector into one PrT net. We treat agents as regular tokens in this model since we have already checked host nets, agent nets and the interaction. The token structure in the net: $\langle dl, sl, type, cmd \rangle$, 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.

The properties were checked are: 1) If the host net *LB1* sends a message ($M(p_{in-in}) = \langle LB2, LB1, MSG, msg \rangle$) or an agent ($M(p_{in-in}) = \langle LB2, LB1, AN, ai \rangle$, and $M(p_w) = \langle ai \rangle$) to destination host *LB2*, eventually it arrives at its destination *LB2*, $M(SN_{LB2}, P_{ex-in}) = \langle LB2, LB1, type, msg \rangle$, and then to the p_w in *LB2*. The LTL formula is: $((ps.sl == LB2) \ \&\& \ (ps.type == AN) \ \&\& \ (ps.dl == LB1) \rightarrow \diamond((p_{11}.type == AN) \ \&\& \ (p_{11}.dl == LB2)))$, where p_{11} represents p_w in *LB2*. 2) If the host net *LB2* sends a message or an agent to destination host *LB1*, eventually it will arrive at its destination *LB1*.

Part of the running results of checking one property is given below:

```
Full statespace search for:
  never claim          +
  assertion violations + (if within scope of
claim)
  acceptance cycles + (fairness enabled)
  State-vector 84 byte, depth reached 110,
errors: 0
```

The interpretation is similar to that in subsection A. It shows that property $((ps.sl == LB2) \ \&\& \ (ps.type == AN) \ \&\& \ (ps.dl == LB1) \rightarrow \diamond((p_{11}.type == AN) \ \&\& \ (p_{11}.dl == LB2)))$ holds.

6. CONCLUSIONS

We have presented the approach for checking LAM models of mobile agent systems using SPIN. Based on model-checking PrT nets by transforming them into SPIN input programs, we can verify various properties about agent mobility, agent environments, as well as system interaction. The work in this paper provides an effective method for analyzing mobile agent systems modeled by LAM. Considering that a literature review on the formal modeling of mobile agent systems is already available in [23], we do not intend to duplicate the discussions on the related work. Nevertheless, it is worth mentioning that in the more recent work [16], agent mobility is similarly defined in terms of the paradigm of “nets within nets” [1], [20]. Object nets and system nets are both modeled using reference nets [14], which are extended color Petri nets; and object nets are embedded as

tokens in system nets. The communication between nets is through synchronous channels instead of connectors in LAM. We believe that our model checking approach is applicable to the agent mobility model in [16]. To do so, we can treat communication channels as connectors in LAM.

Concerning the further work, we will investigate compositional model checking methods in order to overcome the state-space explosion issue for very complex mobile agent systems. Another direction is the modeling and verification of agent cooperation.

ACKNOWLEDGEMENT

This research was supported in part by the National Science Foundation of the USA under grant HRD-0317692, and by the National Aeronautics and Space Administration of the USA under grant NAG2-1440.

The authors would like to thank the comments provided by the anonymous reviewers and editor, which help the authors improve this paper significantly. We would also acknowledge the help from Jiacun Wang of Monmouth University.

APPENDIX I

```

/* A Promela program for a */
/* simplified agent net */
/* CL as 1, DA as 111, STOP as 100 */
/* NL 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) ->
        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;
        p0.da = p2.da;
        p0.cmd = p2.cmd;
        resetp(p2)
    }
}
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()
}

```

REFERENCES

- [1] E. Best, H. Fleischhack, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz: A Class of Composable High Level Petri Nets: *ICATPN'1995, LNCS 935*,103-120, 1995
- [2] E.M. Clark, O. Grumberg, and Doron A. Peled: *Model Checking*, Cambridge, MA. MIT Press, 1999
- [3] J. Ding, Z. Dai, J. Wang, and X. He, Formally Modeling and Analyzing a Secure Mobile Agent Finder, *IEEE International Conference on Systems, Man, and Cybernetics*, Hawaii, USA, 2005
- [4] E.A. Emerson, Temporal and modal logic, *Handbook on Theoretical Computer Science*, vol. B, Elsevier Science, pp. 995-1072, 1990
- [5] J. Esparza, S. Romer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *TACAS'96, LNCS 1055*, 87-106, 1996
- [6] H.J. Genrich: Predicate/Transition Nets. *Petri Nets: Central Models and Their Properties*, W. Brauer, W. Reisig and G. Rozenberg, eds., pp. 207-242, 1987
- [7] 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
- [8] 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

- [9] X. He, J. Ding, and Y. Deng: Analyzing SAM Architectural Specifications Using Model Checking. *SEKE2002*, Italy, 2002
- [10] X. He, H. Yu, T. Shi, J. Ding, Y. Deng: Formally analyzing software architecture specification using SAM. *The Journal of Systems and Software*, vol.71 pp.11-29, 2003
- [11] G.J. Holzmann: *The SPIN Model Checker: Primer and reference manual*. Boston, MA. Addison-Wesley, 2003
- [12] M. D. Jeng and X. L. Xie: Analysis of Modularly Composed Nets by Siphons, *IEEE Transactions on Systems, Man, and Cybernetics*, part A, vol. 29, no. 4, pp. 399-406, July 1999
- [13] V. Khomenko and M. Koutny: Towards An Efficient Algorithm for Unfolding Petri Nets. *CONCUR'2001, LNCS 2154*, 366-380, 2001
- [14] O. Kummer, F. Wienberg: *Reference net workshop (Renew)*. <http://www.renew.de>, 1998
- [15] K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. *CAV'92, LNCS 663*, pp.164-174, 1992
- [16] M. Köhler, D. Moldt, H. Rölke: Modelling Mobility and Mobile Agents Using Nets within Nets. *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, pp. 121 – 139, 2003
- [17] B.F. Rodak, W.B. Saunders, *Hematology: Clinical Principles & Applications (2nd Edition)*, Elsevier-Health Sciences Division, January 2002
- [18] H.M. Shapiro: *Practical Flow Cytometry*, Wiley-Liss, March 2003
- [19] M. Shaw: The coming-of-age of software architecture research. *Proceedings of International Conference on Software Engineering*. Toronto, pp. 656-664, 2001
- [20] R.G. Valk: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. *Application and Theory of Petri Nets*, J. Desel and M. Silva (eds.), *LNCS 1420*, pp. 1-25, 1998
- [21] J. Wang: *Timed Petri Nets: Theory and Application*, Kluwer Academic Publishers, 1998
- [22] X. L. Xie and M. D. Jeng: ERCN-merged Nets and Their Analysis Using Siphons, *IEEE Transactions on Robotics and Automation*, vol. 15, no. 4, pp. 692-703, August 1999
- [23] 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, Jan. 2003
- [24] M.C. Zhou, and K. Venkatesh: *Modeling, Simulation and Control of Flexible Manufacturing Systems: A Petri Net Approach*, World Scientific, 1999



Junhua Ding received the MS and PhD degrees in computer science from Florida International University in 2000, 2004, respectively. He is a senior software engineer with Beckman Coulter Inc. in Miami, USA. His research interests are in the areas of distributed software engineering, software specification and design, software verification and validation, and software agents.



Dianxiang Xu received the BS, MS, and PhD degrees in computer science from Nanjing University, China in 1989, 1992, and 1995, respectively. He is assistant professor of computer science at North Dakota State University, USA. From August 2000 to July 2003, he was research assistant professor and engineer in the department of computer science at Texas A&M University. From May 1999 to

August 2000, he was a research associate at the school of computer science, Florida International University. Prior to that, he was an associate professor and an associate head of the department of computer science and technology, Nanjing University. His research interests are in the areas of software engineering, software security, applied formal methods, and software agents. He is a senior member of the IEEE computer society.



Xudong He received the BS and MS degrees in computer science from Nanjing University, China, in 1982 and 1984, respectively. He received the Ph.D. degree in computer science from Virginia Polytechnic Institute & State University (Virginia Tech) in 1989.

He joined the faculty in the School of Computing and Information Sciences (SCIS) at Florida International University (FIU) in 2000, and is a professor of SCIS and the Director of the Center for Advanced Distributed System Engineering. Prior to joining FIU, he was an associate professor in the Department of Computer Science at North Dakota State University since 1989. His research interests include formal methods, especially Petri nets, and software testing techniques. He has published over 80 papers in the above areas. He has been ranked among the top 15 scholars in Software and System Engineering worldwide during 1999 – 2003 by the Journal of System and Software in 2005. He served as the organizing chair of the 26th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency held at FIU in June 2005.

Dr. He is a member of the Association for Computing Machinery, and a senior member the IEEE Computer Society.



Yi Deng received the PhD degree in computer science from the University of Pittsburg, Pennsylvania, in 1992. He is the Dean and Professor of the School of Computing and Information Sciences at the Florida International University (FIU) – the state university of Florida at Miami. Prior to his current position, he had served as the managing director of the Embedded Software

Center and an associate professor of computer science in the School of Engineering and Computer Science, at the University of Texas at Dallas, and as the founding director for the Center for Advanced Distributed Systems Engineering (CADSE)—a university research center and an associate professor of computer science at FIU. His research interests include component-based software engineering, software architecture, intelligent communication systems, formal methods for complex systems, modeling and analysis of software security systems, and middleware. He has published more than 80 papers in various journals and conferences and served as the PI or Co-PI for 15 research awards of over 7 million dollars from various federal agencies and industry. He is an editor of the International Journal of Software Engineering and Knowledge Engineering and has been the program committee chair or member for several conferences. He is a member of the ACM and a senior member of IEEE.