Formal Description Language and Middleware for Managing Communication among Mobile Terminals

Takaaki Umedu

December, 2004

Abstract

Recently, since parsonal mobile terminals are becoming powerful and popular, requirements for distributed cooperative applications on such devices are also becoming larger. New kinds of applications such as network meeting systems, data sharing systems on ad hoc networks and support tools for forming mobile communities have been proposed. However designing and implementing such mobile distributed applications cause various problems that cannot be solved easily by using existing techniques.

In this thesis, first, we have proposed a formal description language for specifying distributed cooperative applications in mobile ad-hoc environments and a middleware based on this language. We define a new language called LOTOS/M which enables dynamic establishment of multi-way synchronization channels among multiple agents (processes running on mobile hosts) on ad hoc networks, and show how it can be applied for designing wireless mobile applications. In LOTOS/M, a system specification is given by a set of independent agents. When a pair of agents is in a state capable of communicating with each other, a synchronization relation on a given gate (channel) list can dynamically be assigned to them by a new facility of LOTOS/M: (i) advertisement for a synchronization peer on a gate list and (ii) participation in the advertised synchronization. The synchronization relation on the same gate list can also be assigned to multiple agents to establish a multi-way synchronization channel incrementally so that the agents can exchange data through the channel. When an agent goes in a state incapable of communication, a synchronization relation assigned to the agent is canceled and it can run independently of the others. By describing some examples, we have confirmed that typical wireless mobile systems can easily be specified in LOTOS/M, and that they can be implemented efficiently with our LOTOS/M to Java compiler.

We also propose a Java-based middleware based on LOTOS/M. The proposed middleware provides facilities for establishing multi-way synchronization channels among multiple agents (processes executed in mobile hosts) based on their locations. So, we can easily handle multicast data distribution and mutual exclusion in communication among agents by using multi-way synchronization. Using the middleware, it is shown that we can easily develop a simple video conferencing application on a wireless network where only one of participants can transmit his/her video data to the others exclusively. Through some experiments on IEEE 802.11b wireless LAN, we have confirmed that the proposed

middleware provides practically good performance in each channel establishment, time for each synchronization execution, and data transfer rate.

Next, we have proposed a decomposition technique of Java programs for mobile terminals that cannot handle learger applications because of their limited resources. For a given application program that exceeds their resource limits, we propose a method for partitioning it into two module sets where we assume that only a part of modules of the given application is assigned to a mobile terminal and the rest of modules are running on its proxy server, and that the mobile terminal invokes the modules on the server using remote method invocation. It is desirable that we can minimize the total amount of communication, delay time and power consumption between the mobile terminal and its server (here, we call the total amount as the *total cost*).

In this research, we propose a technique for dividing a given Java program with multiple modules into the two modules using Simulated Annealing (SA) considering the minimization of the total cost. In the proposed technique, first, a given Java program is repeatedly simulated on a single machine, and we collect the statistics information about (1) the communication amount between each pair of two modules and (2) the consumed CPU time of each module when the given program is executed. We have developed a tool for this information gathering. Then, we give the resource limitation of the mobile terminal such as the memory size and an objective function that shows what total cost should be minimized. Under those constraints, our tool divides a given Java program into the two module sets where the value of the objective function is minimized using SA. We have applied our technique to some application programs and examined its usefulness by evaluating their total costs.

Lastly, we have studied about an efficient deadlock detection method for distributed cooperative applications executed on unspecified number of terminals. We introduce a formal model for designing distributed cooperative systems (concurrent systems) with symmetries and propose an efficient deadlock detection method on this model. In our method, we describe a specification of a system by a set of coloured Petri-nets and synchronization among them. Each coloured Petri-net is either the specification of a participant's behavior or the constraint about the temporal ordering of multiple participants' behavior. For this specification, if given constraints are inconsistent with each other, the total system enters a deadlock state. In general, the reachability analysis for such systems may cause the state explosion problem depending on the size of the system. However, there are a lot of cases that multiple participants carry out the same behavior in distributed cooperative systems such as network meeting. In such symmetric specifications, by merging equivalent states in a given specification, we can reduce the cost necessary for the reachability analysis. Here, we propose an efficient reachability analysis method using symmetries. We have also developed a verification tool based on the method and shown the usefulness of the method using some examples.

List of Major Publications

Papers Corresponding to Thesis

1. Takaaki Umedu, Hirozumi Yamaguch, Keiichi Yasumoto, Akio Nakata, Teruo Higashino : Constraint-Oriented Model for Specifying Distributed Cooperative Systems and Efficient Deadlock Detection Using Symmetries, *Journal of Information Processing Society of Japan*, Vol. 42, No.12, pp.3054-3062 (Dec. 2001). (In Japanese).

2. Takaaki Umedu, Hirozumi Yamaguchi, Keiichi Yasumoto, Akio Nakata and Teruo Higashino : Constraint-Oriented Model for Describing Distributed Cooperative Systems and Efficient Verification Using Symmetries, *International Journal of Computer and Information Science*, Vol. 3, No. 2, pp. 125-136 (Jun. 2002).

3. Keiichi Yasumoto, Takaaki Umedu, Hirozumi Yamaguchi, Akio Nakata and Teruo Higashino : Protocol Animation based on Event-driven Visualization Scenarios in Real-time LOTOS, *Computer Networks*, Vol.40, No.5, pp. 639-663 (Dec. 2002).

4. Takaaki Umedu, Keiichi Yasumoto, Akio Nakata and Teruo Higashino : Middleware for Supporting Dynamic Establishment of Multi-way Synchronization Channels, *Journal of Information Processing Society of Japan*, Vol.45, No.11 (Nov. 2004) (In Japanese) (to appear).

International Conferences Corresponding to Thesis

1. Takaaki Umedu, Hirozumi Yamaguchi, Keiichi Yasumoto and Teruo Higashino : Protocol Synthesis from SMIL-Based Scenarios and Its Implementation in Distributed Environment, *Proceedings of IEEE 15th International Conference on Information Networking (ICOIN-15)*, pp 163-170, (Jan. 2001).

2. Takaaki Umedu, Hirozumi Yamaguchi, Keiichi Yasumoto, Akio Nakata and Teruo Higashino : A Constraint-Oriented Design Method for Dis-

tributed Cooperative Systems and Efficient Deadlock Detection Using Symmetries, *Proceedings of 2001 Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing (SNPD'01)*, pp 584-591, (Aug. 2001).

3. Takaaki Umedu, Yoshiki Terashima, Keiichi Yasumoto, Akio Nakata, Teruo Higashino and Kenichi Taniguchi: A Language for Describing Wireless Mobile Applications with Dynamic Establishment of Multi-way Synchronization Channels, *Proceedings of International Symposium of Formal Methods Europe(FME2002)*, pp.607-624 (Jul. 2002).

4. Takaaki Umedu, Keiichi Yasumoto, Akio Nakata, Hirozumi Yamaguchi and Teruo Higashino: Middleware for Synchronous Group Communication in Wireless Ad Hoc Networks, *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN2002)*, pp. 48-53 (Nov. 2002).

5. Takaaki Umedu, Shigeharu Urata, Akio Nakata and Teruo Higashino : Automatic Decomposition of Java Program for Implementation on Mobile Terminals, *Proceedings of 19th IEEE International Conference on Advanced Information Networking and Applications (AINA2005)* (Jun. 2005) (to appear).

Other Related Papers

1. Keiichi Yasumoto, Akio Nakata, Yoshiki Terashima, Takaaki Umedu, Teruo Higashino and Kennichi Taniguchi : A Language for Wireless Mobile Applications with Dynamic Establishment of Multi-way Synchronization Channels, *Computer Software - Japan Society for Software Science and Technology (JSSST) Journal* , Vol.19, No.2, pp.35-46 (Mar. 2002) (In Japanese).

2. Kouji Nishigaki, Keiichi Yasumoto, Takaaki Umedu, Teruo Higashino and Minoru Ito : Middleware for Cellular Phones Providing Group Formation Based on Context and Group Communication Facility, *International Journal of Computer and Information Science*, Vol.45, No.12 (Dec. 2004) (In Japanese) (to appear).

3. Thilmee M. Baduge, Akihito Hiromori, Takaaki Umedu, Hirozumi Yamaguchi and Teruo Higashino : A Decentralized Protocol MODE for Minimum Delay Spanning Trees on Overlay Networks, *Journal of Information Processing Society of Japan*, Vol.46, No.2 (Feb. 2005) (In Japanese) (to appear).

Other Related International Conferences

1. Kouji Nishigaki, Keiichi Yasumoto, Takaaki Umedu, Teruo Higashino and

Minoru Ito: Middleware Providing Dynamic Group Communication Facility for Cellular Phone Applications, *Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM 2004)*, p.170 (Jan. 2004).

2. Kouji Nishigaki, Keiichi Yasumoto, Takaaki Umedu, Teruo Higashino, Minoru Ito: Middleware Providing Dynamic Group Communication Facility for Cellular Phone Applications, *Proceedings of 24th International Conference on Distributed Computing Systems Workshops - W3: IWSAWC (ICDCSW'04)*, pp. 434-437 (Jan. 2004).

3. Masashi Saito, Mayuko Funai, Takaaki Umedu and Teruo Higashino : Inter-vehicle Ad-hoc Communication Protocol for Acquiring Local Traffic Information, *Proceedings of 11th World Congress on ITS*, CD-ROM, 4066.pdf (Oct. 2004).

4. Masashi Saito, Jun Tsukamoto, Takaaki Umedu, Teruo Higashino : Evaluation of Inter-Vehicle Ad-hoc Communication Protocol, *Proceedings of 19th IEEE International Conference on Advanced Information Networking and Applications (AINA2005)* (Jun. 2005) (to appear).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, since parsonal mobile terminals are becoming powerful and popular, requirements for distributed cooperative applications on such devices are also becoming larger. It is hoped that new kinds of applications such as network meeting systems, data sharing systems on ad hoc networks and support tools for forming mobile communities will change our world. However designing and implementing such mobile distributed applications cause various problems that cannot be solved easily by using existing techniques. At first, for such applications, (1) we cannot fix the number of terminals where the application will execute and how those mobile terminals will connect with each other before execution. In such applications, terminals also connect with and disconnect from each other more often than in the applications for wired environments. In an application such that unspecified number of people who happen to come together will cooperate with each other in ad-hoc wireless environment, the number of users (i.e. the number of terminals) will be changed dynamically according to movements of users. Also the communication channels may often be disconnected by communication errors in wireless environments. Next, (2) available resources in handheld devices are limited and the charges for cellular phones often depends on the amount of communication. So we must implement applications for these environments with considering such conditions. Lastly, (3) since a large number of terminals cooperate with eath others, we must take care to avoid deadlocks caused by inconsistent specification.

So, we have proposed (1) a formal description language for specifying distributed cooperative applications in mobile ad-hoc environments [1] and a middleware based on this language [2]. Also we have proposed (2) a decomposition technique of Java programs that derives optimized assignments of modules for servers and mobile terminals as good as possible in a sense of the given metrics under the given restrictions [3]. Finally, we have studied about (4) an efficient deadlock detection method for distributed cooperative applications executed on unspecified number of terminals [4].

1.1 A Formal Description Language for Mobile Ad-Hoc Environments

LOTOS [5] is one of the formal specification languages for communication protocols. In LOTOS, we can specify *multi-way synchronization* which enables several parallel processes to execute events synchronously. With multi-way synchronization, we can easily handle complicated mechanisms such as broadcast/multi-cast communication and mutual exclusion for accessing resources in distributed systems. It also allows us to describe systems incrementally as a main behavior and a set of behavioral constraints (called the *constraint oriented style* [6, 7]). So, multi-way synchronization seems useful to design and develop wireless mobile systems. However, standard LOTOS does not have the facility for dynamic channel establishment among processes.

On the other hand, in π calculus [8] which is a process algebra including a dynamic channel allocation mechanism between processes, and in M-LOTOS [9, 10] which is an extension of LOTOS introducing the above mechanism of π calculus, dynamic channel establishment among processes can be specified. However, multi-way synchronization cannot be specified among processes.

We propose a new language called LOTOS/M which enables dynamic establishment of multi-way synchronization channels among multiple agents (processes running on mobile hosts) on ad hoc networks, and show how it can be applied to designing wireless mobile applications. LOTOS/M enables dynamic establishment of channels for multi-way synchronization among multiple mobile processes.

In LOTOS/M, a system specification is given by a set of independent agents. When a pair of agents is in a state capable of communicating with each other, a synchronization relation on a given gate (channel) list can dynamically be assigned to them by a new facility of LOTOS/M: (i) advertisement for a synchronization peer on a gate list and (ii) participation in the advertised synchronization. The synchronization relation on the same gate list can also be assigned to multiple agents to establish a multi-way synchronization channel incrementally so that the agents can exchange data through the channel. When an agent goes in a state incapable of communication, a synchronization relation assigned to the agent is canceled and it can run independently of the others. By describing some examples, we have confirmed that typical wireless mobile systems can easily be specified in LOTOS/M, and that they can be implemented efficiently with our LOTOS/M to Java compiler.

This research is detailed in chapter 2.

1.2 A Middleware Providing Multi-way Synchronization Method in Ad-Hoc Environments

We propose a Java-based middleware for group communication in wireless ad hoc networks. The proposed middleware provides the facilities defined in LO-

TOS/M for establishing multi-way synchronization channels among multiple agents (processes executed in mobile hosts) based on their locations. Through multi-way synchronization channels, agents can execute events on the same channel synchronously to exchange data. So, we can easily handle multicast data distribution and mutual exclusion in communication among agents.

To implement multi-way synchronization mechanism, it is important to manage the information about member agents in each agent group and the synchronization relations assigned among the member agents to calculate what events can be executed synchronously among those agents. For the purpose, we represent the synchronization relations assigned to an agent group as a binary tree where each intermediate node corresponds to a binary synchronization relation and each leaf node does an agent, and let the agents keeps the latest tree information in a distributed manner so that events executed synchronously among the agents can be calculated based on the tree. When an agent (or a sub-agent group) goes in a state incapable of communicating with the other agents of an agent group, the synchronization tree is reconstructed so that the remaining agent group can proceed without the isolated agent. We have implemented a polling mechanism to detect whether each agent has gone into such a state incapable of communication or not.

Using the middleware, it is shown that we can easily develop a simple video conferencing application on a wireless network where only one of participants can transmit his/her video data to the others exclusively. Through some experiments on IEEE 802.11b wireless LAN, we have confirmed that the proposed middleware provides practically good performance in each channel establishment, time for each synchronization execution, and data transfer rate.

This research is detailed in chapter 3.

1.3 Automatic Decomposition of Java Program for Mobile Terminals

To implement distributed application on mobile terminals, there exist other problems such as resource limitation. Here, for a given application program that exceeds their resource limits, we propose a method for partitioning it into two module sets where we assume that only a part of modules of a given application is assigned on a mobile terminal and the rest of modules are running on its proxy server, and that the mobile terminal invokes the modules on the server using remote method invocation. It is desirable that we can minimize the total amount of communication, delay time and power consumption between the mobile terminal and its server (here, we call the total amount as the *total cost*). In this research, we propose a technique for dividing a given Java program with multiple modules into the two modules using Simulated Annealing (SA) considering the minimization of the total cost.

To derive such a division, the statistical data about the amount and number of communication among modules are needed. For statistical performance eval-

uation, there are many studies based on analysis of source codes such as studies of slicing of parallel Java programs [11, 12]. However, here for simplicity of discussion, we use a simulation based performance evaluation technique. In our technique, additional codes for performance evaluation are inserted to the given source code automatically. The codes are inserted to be called before all the method invocations and record the amount and number of communication between two modules (classes). The statistical performance data can be collected by executing the modified code repeatedly with considering various situations.

In this way, we collect the statistics information about (1) the communication amount between each pair of two modules and (2) the consumed CPU time of each module when the give program is executed. We have developed a tool for this information gathering. Then, we give the resource limitation of the mobile terminal such as the memory size and an objective function that shows what total cost should be minimized. Under those constraints, our tool divides a given Java program into the two module sets where the value of the objective function is minimized. Here, we have proven that this division problem is NP-hard. Since in general the optimized solution cannot be derived in practical time, we use an SA algorithm that is one of representative heuristic algorithms, to get an approximated solution.

We have applied our technique to some application programs and examined its usefulness by evaluating their total costs.

This research is detailed in chapter 4.

1.4 An Efficient Deadlock Detection Method Using Symmetries for Distributed Applications

We introduce a formal model for designing distributed cooperative systems (concurrent systems) with symmetries and propose an efficient deadlock detection method on this model. In our method, we describe a specification of a system by a set of coloured Petri-nets and synchronization among them based on the constraint oriented style [7]. Each coloured Petri-net is either the specification of a participant's behavior or the constraint about the temporal ordering of multiple participants' behavior. For this specification, if given constraints are inconsistent with each other, the total system enters a deadlock state. In general, the reachability analysis for such systems may cause the state explosion problem depending on the size of the system.

In order to reduce the verification costs, Ref. [13] proposes techniques to merge equivalent states into one and make a reduced size's reachability graph called *OS graph* [14] from the original reachability graph. Ref. [15, 16] propose another kind of efficient reachability analysis techniques using *symbolic reachability graph*. Ref. [17, 18] use invariants for reducing the verification costs. Ref. [19, 20] use stochastic Petri-nets, and Ref. [21] uses compositional high level Petri-nets for efficient reachability analysis.

However, in distributed cooperative systems, there are a lot of cases that

multiple participants carry out essentially the same behavior and they do not cause different results for reachability analysis. For example, in a simple network meeting system where only one of multiple participants can be a speaker at each moment, the number of the participants does not affect the reachability analysis of the system specification, since the behavior of those participants can be regarded as the same, i.e., the specification has symmetries.

Here, we propose an efficient reachability analysis method using symmetries. We have also developed a verification tool based on the method and shown the usefulness of the method using some examples.

This research is detailed in chapter 5.

Chapter 2

A Formal Description Language for Mobile Ad-Hoc Environments

2.1 Introduction

Owing to recent maturity of wireless transmission technologies and popularity of personal mobile devices (e.g., cellular phones, PDA, etc), wireless mobile applications are becoming more and more important. Various applications have been proposed, for example, location aware systems [22] in ubiquitous networks, virtual meeting on ad hoc networks [23], and so on.

Such wireless mobile applications need dynamic communication facilities with which channels are dynamically allocated between mobile hosts and they can communicate via the channels when they happen to meet in a common radio range (communication area). Therefore, languages and tools for design and implementation of mobile applications which have dynamic communication facilities are desired.

LOTOS [5] is one of the formal specification languages for communication protocols, which has the powerful operators such as choice, parallel and interruption among multiple processes. With the parallel operators, we can specify *multi-way synchronization* which enables several parallel processes to execute the specified events synchronously to exchange data. With multi-way synchronization, we can easily handle complicated mechanisms such as broadcast/multicast communication and mutual exclusion for accessing resources in distributed systems. It also allows us to describe systems incrementally as a main behavior and a set of behavioral constraints (called the *constraint oriented style* [6, 7]). So, multi-way synchronization seems useful to design and develop wireless mobile systems.

However, standard LOTOS does not have the facility for dynamic channel

establishment among processes when those processes are in a state capable of communicating with each other (e.g., by approaching in a common radio range). Since mobile systems require such dynamic communication, it is difficult to apply LOTOS to design and implement such systems.

On the other hand, in π calculus [8] which is a process algebra including a dynamic channel allocation mechanism between processes, and in M-LOTOS [9, 10] which is an extension of LOTOS introducing the above mechanism of π calculus, dynamic channel establishment among processes can be specified. However, multi-way synchronization cannot be specified among processes.

In this paper, we propose a new language called LOTOS/M which enables dynamic establishment of channels for multi-way synchronization among multiple mobile processes. In LOTOS/M, a system specification is given by a set of independent multiple agents (processes running on mobile hosts).

When a pair of agents is in a state capable of communicating with each other, a synchronization relation on a given gate (channel) list can dynamically be assigned to them by a new facility of LOTOS/M: (i) advertisement for a synchronization peer with a gate list and (ii) participation in the advertised synchronization. The pair of combined agents (agents with synchronization relation) is regarded as a single agent, and thus can combine with another agent on a gate list. The synchronization relation on the same gate list can also be assigned to multiple agents to enable multi-way synchronization. The group of combined agents is called the *agent group*, and its member agents can communicate with each other by multi-way synchronization until the synchronization relation is canceled. When an agent (or a sub agent group) goes in a state incapable of communication, the synchronization relation assigned to the agent is canceled and it can run independently of the others.

In LOTOS/M, the agents in an agent group form a binary tree (called the *synchronization tree*) where each node corresponds to the synchronization operator of LOTOS or an agent itself. This property makes it easy to implement specifications based on the existing techniques of our standard LOTOS compiler [24]. The current version of our LOTOS/M compiler generates from a given LOTOS/M specification, multiple Java programs which run on corresponding mobile hosts. We assume that data types and functions specified in LOTOS/M specifications are available as the corresponding Java methods. So, our compiler provides only a mechanism to invoke those methods. Thus, our compiler can be used as a tool for developing concurrent Java programs with a multi-way synchronization mechanism.

In the following Sect. 2.2, we outline the LOTOS language and problems to describe mobile systems in it. In Sect. 2.3, we define LOTOS/M language and give its formal semantics. In Sect. 2.4, we describe typical mobile applications in the proposed language to show applicability of LOTOS/M. Sect. 2.5 outlines our implementation technique. Finally, Sect. 2.6 concludes the this chapter.

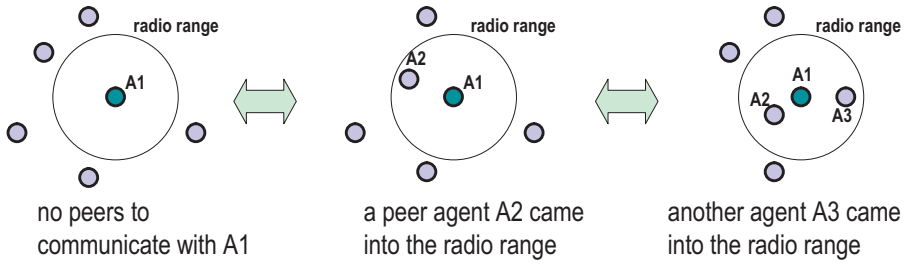


Figure 2.1: Dynamic change of agent combinations capable of communication

2.2 LOTOS and its Applicability to Mobile Systems

2.2.1 Outline of LOTOS

LOTOS [5] is a formal description language for communication protocols and distributed systems, which has been standardized by ISO. In LOTOS, a system specification can be described by a parallel composition of several (sequential) processes. The behavior of each process is described by a *behavior expression*, which specifies execution sequences of *events* (actions observable to the external environment), *internal events* (unobservable actions), and process invocations. Here, an event is an interaction (input/output of data) between a process and an external environment, which occurs at an interaction point called a *gate*.

To specify the ordering of execution, the operators such as action prefix $(a; B)$, choice $(B1[]B2)$, parallel $(B1|[G]|B2)$ [1], interleaving $(B1|||B2)$, disabling $(B1[> B2)$ ($B2$ can interrupt $B1$) and sequential composition $(B1 >> B2)$ ($B2$ starts when $B1$ successfully terminates) are specified at each pair of sub-expressions. Especially, using the parallel operator, we can specify *multi-way synchronization*, that is, multiple (possibly more than two) processes executing events on the same gate simultaneously and exchanging data at the gates. In addition, we can restrict the execution of B by a boolean expression *guard* by denoting "$[guard] -> B$". Also, we can create new gates gl used only in B by denoting "**hide** gl **in** B".

By specifying multi-way synchronization among multiple nodes of a distributed system, we can easily describe systems with complicated mechanism such as broadcasting/multicasting and/or mutual exclusion [2] for accessing resources [24]. Moreover, it is known that using multi-way synchronization, some facilities such as step-by-step addition of behavioral constraints among nodes by the constraint-oriented specification style [6, 7] are available. Therefore, it is also desirable to use multi-way synchronization among agents of mobile systems.

[1] Hereafter, we say that in $B1|[G]|B2$, a *synchronization relation* on gate list G is assigned between $B1$ and $B2$.

[2] Sometimes two-way synchronization may be enough, e.g., $R|[G]|(C1|||C2|||...|||Cn)$

2.2.2 Problems for Describing Mobile Systems in LOTOS

In a wireless mobile system, as shown in Fig.2.1, combinations of agents capable of communicating with each other dynamically change since they move around. Thus, to describe such a system in LOTOS, the following problems arise.

(1) There is no facility to allocate channels to a combination of agents only when they are in a state capable of direct communication (e.g., when they are in a common radio range).

(2) There exist a lot of possible combinations of agents, depending on their locations, on the number of agents participating in the same multi-way synchronization, or so on. In LOTOS, basically we must describe all of such combinations of agents statically in a behavior expression.

In [25], a mobile telephony system including roaming services is described in LOTOS. However, the system is restricted to use only one-to-one communications between mobile hosts and base stations, and solve the above problem partly by dynamically exchanging IDs of each mobile host and each base station.

To solve the problem essentially, we think that we need a language support to dynamically assign among any combination of agents a synchronization relation for multi-way synchronization though which the agents can communicate by executing events synchronously.

2.3 Proposal of LOTOS/M

In order to solve the problems in the previous section, we propose a new language called *LOTOS/M* suitable to describe wireless mobile applications.

2.3.1 Definition of LOTOS/M

We show new constructs and their informal semantics of LOTOS/M in Table 2.1. In LOTOS/M, the entire mobile system is given as a set of independent agents $A := A_1 \mid A_2 \mid ... \mid A_n$ where each A_i does not know how to communicate with other agents. We define the operational semantics of $A_1 \mid A_2 \mid ... \mid A_n$ (also denoted by $|\{A_1, ..., A_n\}$) as follows (Act is the set of all events used in $A_1, ..., A_n$).

$$\frac{A_i \xrightarrow{a} A_i', \quad a \in Act \cup \{sync, disc\}}{|\{A_1, ..., A_n\} \xrightarrow{a} |\{A_1, ..., A_i', ..., A_n\}}$$

The behavior expression of each agent A_i is specified with a new construct "**agent** ... **endagent**" where the expression includes only operators of standard LOTOS and "**sync** ... **endsync**" (i.e., "|" cannot be used).

Channel establishment In order to allocate channels (assign a synchronization relation) between a pair of agents only when they can physically communicate with each other, LOTOS/M provides the following special actions:

Table 2.1: Extended syntax and informal semantics

Syntax	Semantics
$A1 \mid A2 \mid ... \mid An$	Parallel execution of n agents $A1, ..., An$ independently of each other.
agent $A[G](E) := B$ **endagent**	Definition of the behavior expression of agent A.
sync $!G : sid\ IO\ Guard$ **in** B **endsync**	Advertisement for a synchronization peer with gatelist G and definition of behavior expression B to be executed after the peer agent has been found.
sync $?H : sid\ IO\ Guard$ **in** B **endsync**	Acceptance for an advertisement and definition of behavior expression B to be executed after the acceptance has been approved.
$disc!sid; B$	Disconnection of a channel (cancellation of a synchronization relation) established with $ID = sid$ and definition of behavior expression B to be executed after the disconnection.
$g!P; ... \ \|[g]\|_{sid}\ g?Q : process; ...$	Exchange of a process name among agents.

(i) an advertisement for a synchronization peer (**sync** $!G : sid\ IO\ Guard$ **in** $B1$ **endsync**) and (ii) participation in a synchronization advertisement (**sync** $?H : sid\ IO\ Guard$ **in** $B2$ **endsync**).

Here, G and H denote gate lists, "$: sid$" represents a variable for keeping the ID for the synchronization relation between agents and is used for active cancellation of the synchronization relation with $disc!sid$ action. "IO" represents a list of input and output parameters (e.g., !E1?x1?x2), and "$Guard$" is the boolean expression denoted by $[f(c1, c2, ...x1, x2, ...)]$ where constants $c1, c2, ...$ and parameters $x1, x2, ...$ in IO may be used. IO and $Guard$ are used to restrict only specific agents to be combined. "IO" and "$Guard$" may be omitted.

If the following conditions hold for a pair of agents, then a synchronization relation on a given gate list G is assigned between the agents (we also say that the two agents are *combined* (or *joined*) on gate list G).

- one agent $\mathcal{A}_1$ is ready to execute the *sync* action "$B1$:=**sync** $!G :\ IO1\ Guard1$ **in** $B1'$ **endsync**" (called *host agent*) and the other agent $\mathcal{A}_2$ can execute "$B2$:=**sync** $?H : sid\ IO2\ Guard2$ **in** $B2'$ **endsync**" (called *participant agent*).
- the numbers of gates in G and H are the same.
- the numbers of parameters in $IO1$ and $IO2$ are the same, and each pair

of the corresponding parameters consist of an input (?x) and an output (!E) where their types must match.

- both of $Guard1$ and $Guard2$ hold after assigning the value of each output parameter to the corresponding variable of the input parameter (e.g., when the parameters are $?x$ and $!E$, the value of expression E is assigned to variable x).

- the two agents are in a state capable of communicating with each other physically[3].

The succeeding behavior is equivalent to $\mathcal{A}_1[B1'/B1]$ $|[G]|_{sid}$ $\mathcal{A}_2[B2'[G/H]/B2]$. Here, $|[G]|_{sid}$ is the new operator called *ad hoc parallel* operator which is equivalent to $|[G]|$ except that its operands can be separated by $disc!sid$ action. $\mathcal{A}[B'/B]$ is the entire behavior expression of agent $\mathcal{A}$ obtained by replacing sub-behavior expression B with B', and $B2[G/H]$ is a behavior expression obtained by replacing every gate in H appearing in $B2$ with the corresponding gate in G.

Also, note that G must be created with *hide* operator of LOTOS (or G may be the gate list received from another agent by $sync?G$) before used in $sync!G$. Also, in $sync?H$, H must not be included in environment gates of the agent (e.g., interaction points to the user).

The combined agents are treated as one agent and called the *agent group*. Each agent group can be combined incrementally with another agent by executing $sync$ $!G$ (or $sync$ $?H$) action. As an example, Fig. 2.2 illustrates that the following three agents are combined in a step-by-step manner.

```
A1 | A2 | A3
where
  A1:= sync !{g} in g; stop endsync
  A2:= sync ?{h} in
         sync !{h} in h; stop endsync
       endsync
  A3:= sync ?{f} in f; stop endsync
```

(Here, *disc* action is omitted).

Channel disconnection Each agent group can be separated into several agents/agent groups by executing active/passive disconnection. We define the effect of disconnection to internal behavior of each agent as follows.

(1) active disconnection By executing $disc!sid$ action, each agent can disconnect the specified channel (synchronization relation) spontaneously. This is called *active disconnection*. When an agent goes into a state incapable of communication, e.g., by moving out of a common radio range, we think that active disconnection is executed by the agent.

[3]In LOTOS/M, whether each agent is currently able to communicate with another agent is treated as an implementation matter. As a process algebra treating location information to check capability of direct communication, for example, [26] is proposed.

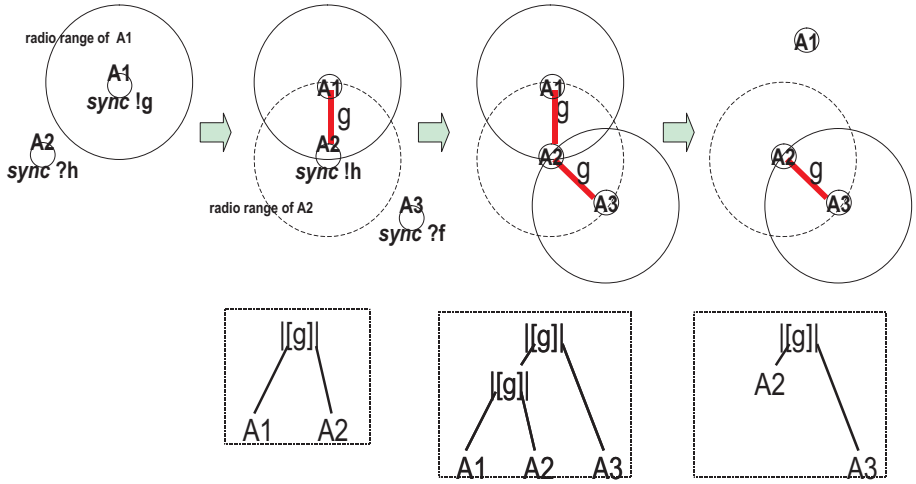


Figure 2.2: Assignment/cancellation of the synchronization relation among agents

(2) passive disconnection When an agent executes $disc!sid$, $pdisc!sid$ is executed at each member agent in its agent group to inform about the channel disconnection. This is called *passive disconnection*.

(3) synchronization of channel disconnection $disc!sid$ and $pdisc!sid$ must be executed synchronously among agents.

In each agent's behavior expression B, we can describe an appropriate processing after each channel disconnection, as an exception handling process when the corresponding $disc!sid$ or $pdisc!sid$ is executed. Below, we show an example where the behavior stops by $disc!sid$ but continues by $pdisc!sid$.

```
sync ?G:sid in
(   B'
||| pdisc!sid; exit
)
[>
  disc!sid; exit
endsync
```

(Here, we omit exception description by passive disconnections with IDs other than *sid*).

Exchange of processes as data values As shown in Table 2.1, in LOTOS/M, process names can also be treated as data-type like higher-order π calculus [27]. And thus it is possible to describe systems where mobile hosts dynamically download programs. In LOTOS/M, only process names are exchanged between agents (therefore semantics extension is not needed). We assume that the agent which has received a process name can invoke the process

(that is, each agent can obtain the behavior expression of the received process). How to exchange the behavior expression of a process is left as an implementation matter (it can easily be implemented in Java language). The example applications using these features are shown in Sect. 2.4.

2.3.2 Semantics of LOTOS/M

Here, we define the semantics of extended constructs, *sync* action and *disc* (*pdisc*) action as follows. We also define a structured operational semantics in Table 2.2 to provide a precise formal definition of the semantics.

- Identifier *sid* is issued for each execution of a *sync* action so that *disc* action with the issued *sid* can cancel the assigned synchronization relation.
- When an agent A executes $disc!sid$ action, the operand of the ad hoc parallel operator with $ID = sid$ (the sub-agent group including A) leaves from the entire agent group and the separated two agents run independently. At the time, the corresponding synchronization relation is canceled and the cancellation is informed to all the agents in the agent group.

Combining When there exist a host agent executing *sync* $!G$ and a participant agent executing *sync* $?X$, these two agents are combined and a synchronization relation on G is assigned by the inference rule **Agent-Join** in Table 2.2. X is a list of formal gate parameters, which are replaced with the actual gate list G when combined. *id* is issued for this synchronization relation, which can be disconnected by the *disc* action with the same *id*. Table 2.3 shows how inference rules are applied when three agents A_1, A_2 and A_3 are combined into one agent group.

Isolation When an agent executes $disc!id$, it must be separated from its agent group (this is called *isolation*).

Inference rules for agent isolation in Table 2.2 have the following meaning.

- **Agent-Leave-1** is the rule for the case when *sid* matches the ID of the current ad hoc parallel operator. This rule removes the ad hoc parallel operator with $ID = sid$ after registering it as an auxiliary term, and makes the peer agent execute $pdisc!sid$.
- **Agent-Leave-2** is the rule for the case when *sid* does not match the ID of the current ad hoc parallel operator. This rule makes the disconnection request work outside of agent group $A_1|[G]|_{sid'}A_2$. This also makes the peer agent execute $pdisc!sid$.
- **Agent-Leave-3** is the rule which brings the disconnected agent in the auxiliary term to the upper node in the syntax tree and makes the peer agent execute $pdisc!sid$.
- **Agent-Leave-4** is the rule which makes the disconnected agent in an auxiliary term work as an independent agent.

Table 2.2: Structured operational semantics for LOTOS/M

Agent-Join

$$\frac{A_i \overset{sync!G}{\longrightarrow} A_i', \quad A_j \overset{sync?X}{\longrightarrow} A_j', \quad G = \{g_1, ..., g_k\}, \quad X = \{x_1, ..., x_k\}}{|\{A_1, ..., A_k\} \overset{sync!G}{\longrightarrow} (A_i' \; |[G]|_{sid} \; A_j'[G/X] \quad | \quad |\{A_1, ..., A_k\} - \{A_i, A_j\})}$$

Here, A_i (also, A_j) is the behavior expression of each agent/agent group. $sync!G$ and $sync?H$ can be replaced each other in A_i and A_j. $B[G/X]$ is obtained from B by replacing each free occurrence of formal gate parameters in $X = \{x_1, ..., x_k\}$ used in the B with the corresponding actual gate parameters in $G = \{g_1, ..., g_k\}$. Since the semantics of the guard expressions specified in *sync* action is the same as LOTOS, it is omitted here.

Agent-Leave-1

$$\frac{A_1 \overset{disc!sid}{\longrightarrow} A_1', \; A_2 \overset{pdisc!sid}{\longrightarrow} A_2'}{[A_1|[G]|_{sid} A_2 \overset{disc!sid}{\longrightarrow} A_2', \; A_1']}$$

Agent-Leave-2

$$\frac{A_1 \overset{disc!sid}{\longrightarrow} A_1', \; A_2 \overset{pdisc!sid}{\longrightarrow} A_2', \; sid <> sid'}{A_1|[G]|_{sid'} A_2 \overset{disc!sid}{\longrightarrow} A_1'|[G]|_{sid'} A_2'}$$

Agent-Leave-3

$$\frac{[A_1 \overset{disc!sid}{\longrightarrow} A_1', A_3], \; A_2 \overset{pdisc!sid}{\longrightarrow} A_2', \; sid <> sid'}{[A_1|[G]|_{sid'} A_2 \overset{disc!sid}{\longrightarrow} A_1'|[G]|_{sid'} A_2', \; A_3]}$$

Agent-Leave-4

$$\frac{[A \overset{disc!sid}{\longrightarrow} A', \; A'']}{A \overset{disc!sid}{\longrightarrow} A'|A''}$$

Agent-Leave-5

$$\frac{A_1 \overset{pdisc!sid}{\longrightarrow} A_1', \; A_2 \overset{pdisc!sid}{\longrightarrow} A_2', \; sid <> sid'}{A_1|[G]|_{sid'} A2 \overset{pdisc!sid}{\longrightarrow} A_1'|[G]|_{sid'} A_2'}$$

Here, $A, A', A_1, A_2, ...$ represent the behavior expressions of agents/agent groups. The auxiliary term $[A \overset{disc!sid}{\longrightarrow} A', A'']$ is the same as the transition relation $A \overset{disc!sid}{\longrightarrow} A'$ except that the extra information A'' (agent to be isolated) are attached. In **Agent-Leave-1–5**, *disc* and *pdisc* can be replaced each other in A_1 and A_2.

- **Agent-Leave-5** is the rule which represents that $A_1|[G]|_{sid'}A_2$ can execute $pdisc!sid$ only if A_1 and A_2 can execute $pdisc!sid$.

Suppose that an agent group $(A_1|[g]|_1A_2)|[h]|_2A_3$ is the result after A_1 and A_2 have combined on $g(sid = 1)$ and then A_1 and A_3 have combined on $h(sid = 2)$. In Table 2.4, we show how inference rules are applied when each agent leaves from the above agent group with different sid.

(1) When A_2 is isolated with $sid = 1$, that is, $A_2 \stackrel{disc!1}{\longrightarrow} A_2'$:

Inference rules are applied as shown in Example1 of Table 2.4. As a result, A_2 leaves from the agent group, and $pdisc!1$ is executed in A_1 and A_3 so that they know the active disconnection from A_2 with $sid = 1$.

(2) When A_2 is isolated with $sid = 2$, that is, $A_2 \stackrel{disc!2}{\longrightarrow} A_2'$:

Inference rules are applied as shown in Example2 of Table 2.4. As a result, A_2 is separated as a sub-agent group $(A_1|[g]|_1A_2)$ from A_3, and A_1 and A_3 know the fact by execution of $pdisc!2$.

(3) When A_3 is isolated with $sid = 2$, that is, $A_3 \stackrel{disc!2}{\longrightarrow} A_3'$:

Inference rules are applied as shown in Example3 of Table 2.4. As a result, A_3 is separated from $(A_1|[g]|_1A_2)$, and A_1 and A_2 know the fact by execution of $pdisc!2$.

Note that the semantics in Table 2.2 enables us to construct an LTS from a given LOTOS/M specification where each node of the LTS corresponds to a tuple of the current agent behaviors (e.g., $A1 \mid (A2' \mid[F]|_1\ A3')$) and each label corresponds to an event, $sync$ or $disc(pdisc)$ action.

Reason why the above semantics were chosen

In [28] Groote proved that if the inference rules of a given operational semantics satisfy one of the following conditions, the semantics preserves the congruence relation.

$$\frac{t_1 \stackrel{a_1}{\longrightarrow} y_1, t_2 \stackrel{a_2}{\longrightarrow} y_2, ...}{f(t_1, t_2, ...) \stackrel{a}{\longrightarrow} t'} \qquad \frac{t_1 \stackrel{a_1}{\longrightarrow} y_1, t_2 \stackrel{a_2}{\longrightarrow} y_2, ...}{x \stackrel{a}{\longrightarrow} t'}$$

Although rule **Agent-Join** satisfies the above sufficient condition, rule **Agent-Leave** does not. This is because we think that any agent should be able to leave from its agent group to whatever ad hoc parallel operator $(|[G]|_{id})$ it connects. If we modify rule **Agent-Leave** so that it can be applied only to the root operator of the syntax tree of the agent group (i.e., the agents can be separated only in reverse order when they combined), we can construct the semantics which preserves the congruence relation. However, in actual mobile applications, we cannot expect in what order agents are leaving from the agent group, we think that such modification makes no sense. That's a reason why the semantics in Table 2.2 were chosen.

Table 2.3: Applying inference rules when agents combine

When $A_1 \overset{sync!G}{\longrightarrow} A_1'$, $A_2 \overset{sync?F}{\longrightarrow} A_2'$, $A_3 \overset{sync!F}{\longrightarrow} A_3' \overset{sync?G}{\longrightarrow} A_3''$,
$A_1 \mid A_2 \mid A_3$
$\overset{sync!F:1}{\longrightarrow} A_1 \mid (A_2' \mid [F]\mid_1 A_3')$
$\overset{sync!G:2}{\longrightarrow} A_1' \mid [G]\mid_2 (A_2' \mid [F]\mid_1 A_3'')$

Table 2.4: Applying inference rules when agents are isolated

Example1

$$\dfrac{\dfrac{\dfrac{A_2 \overset{disc!1}{\longrightarrow} A_2' \quad A_1 \overset{pdisc!1}{\longrightarrow} A_1'}{[A_1|[g]|_1A_2 \overset{disc!1}{\longrightarrow} A_1', A_2']}\ (\ \textbf{Agent-Leave-1}\ ) \qquad A_3 \overset{pdisc!1}{\longrightarrow} A_3' \quad 1 <> 2}{[(A_1|[g]|_1A_2)|[h]|_2A_3 \overset{disc!1}{\longrightarrow} (A_1'|[h]|_2A_3'), A_2']}\ (\ \textbf{Agent-Leave-3}\ )}{(A_1|[g]|_1A_2)|[h]|_2A_3 \overset{disc!1}{\longrightarrow} (A_1'|[h]|_2A_3')|A_2'}\ \textbf{(Agent-Leave-4)}$$

Example2

$$\dfrac{\dfrac{\dfrac{A_2 \overset{disc!2}{\longrightarrow} A_2' \quad A_1 \overset{pdisc!2}{\longrightarrow} A_1' \quad 2 <> 1}{A_1|[g]|_1A_2 \overset{disc!2}{\longrightarrow} A_1'|[g]|_1A_2'}\ (\ \textbf{Agent-Leave-2}\ ) \qquad A_3 \overset{pdisc!2}{\longrightarrow} A_3'}{[(A_1|[g]|_1A_2)|[h]|_2A_3 \overset{disc!2}{\longrightarrow} A_3', A_1'|[g]|_1A_2']}\ (\ \textbf{Agent-Leave-1}\ )}{(A_1|[g]|_1A_2)|[h]|_2A_3 \overset{disc!2}{\longrightarrow} A_3'|(A_1'|[g]|_1A_2')}\ \textbf{(Agent-Leave-4)}$$

Example3

$$\dfrac{\dfrac{\dfrac{\dfrac{A_1 \overset{pdisc!2}{\longrightarrow} A_1' \quad A_2 \overset{pdisc!2}{\longrightarrow} A_2' \quad 2 <> 1}{A_3 \overset{disc!2}{\longrightarrow} A_3' \qquad A_1|[g]|_1A_2 \overset{pdisc!2}{\longrightarrow} A_1'|[g]|_1A_2'}\ \textbf{(Agent-Leave-5)}}{[(A_1|[g]|_1A_2)|[h]|_2A_3 \overset{disc!2}{\longrightarrow} A_1'|[g]|_1A_2', A_3']}\ \textbf{(Agent-Leave-1)}}{(A_1|[g]|_1A_2)|[h]|_2A_3 \overset{disc!2}{\longrightarrow} (A_1'|[g]|_1A_2')|A_3'}\ \textbf{(Agent-Leave-4)}$$

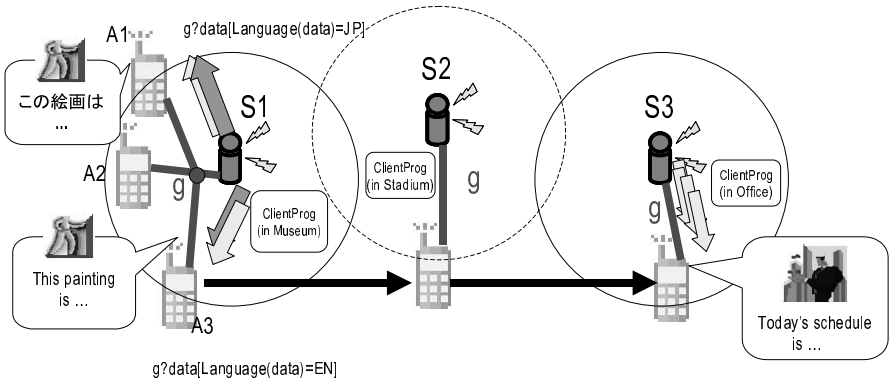


Figure 2.3: Example of a location aware system

2.4 Describing Wireless Mobile Systems in LOTOS/M

2.4.1 Location Aware System

In order to develop a location aware system [22], mechanisms (i) for detecting each agent's location and (ii) for providing a different service depending on each location are required. In LOTOS/M, (i) is realized by the dynamic channel establishment mechanism that a base station (host agent) advertises for a synchronization peer in its radio range and a mobile host responds it. Similarly, (ii) is easily realized by the process name exchange mechanism so that a mobile host can download a specific behavior expression from a base station.

An example of a location aware system is depicted in Fig. 2.3. We also show an example specification in LOTOS/M in Table 2.5.

In the example, there are three base stations $S1, S2$ and $S3$ and multiple mobile hosts $A1, A2, ..., An$. Gates f and g are used as interaction points between each base station and each agent. In the system, $S1$ advertises for synchronization peers in its radio range for a while (it is described in sub-process $Service$ by iterative execution of $sync\ !\{f, g\}$ action). After some time elapses ($[StartTime()] - > ...$), $S1$ sends through gate f a program (f!ClientProg) to the agents which participated in the advertised synchronization ($sync\ ?\{h, g\}$ action in agent $A1$). Then $S1$ starts transmitting through gate g information in several languages in parallel ($Transmit[g](JapaneseInfo)\ |||\ ...$). Data is transmitted to all participated agents at the same time with the property of multiway synchronization. Since each user selects his/her mother language ($u?lang$), only information in the specified language are displayed ($[Language(data) = lang] - > ...$). When some of the participated agents move out of the radio range or execute $disc$, only the agents are isolated (the behavior of agents are initialized by $...[> (disc!sid; A1[u][]pdisc!sid; A1[u]))$, and the other agents can

Table 2.5: Example specification of a location aware system

```
specification LocationAware:  noexit
behavior
 A1 | A2 | ...  | S1 | S2 | S3
where
 agent A1[u] :  exit:= (* A2, A3 are similar *)
  sync ?{h,j}:sid in
   (h?ClientProg:process; u?lang; ClientProg[j](lang))
   [> (disc!sid; A1[u] [] pdisc!sid; A1[u])
  endsync
 endagent
 agent S1 :exit:=
  hide f,g in
   Service[f,g]
 where
  process Service[f,g] :  exit:=
   sync !{f,g}:sid in
     [not(StartTime())]- >
      (Service[f,g] ||| (disc!sid; exit [] pdisc!sid; exit))
   [] [StartTime()] - > f!ClientProg; ProvideInfo[g])
   endsync
  endproc
  process ProvideInfo[g] :noexit:=
   Transmit[g](JapaneseInfo)
  ||| Transmit[g](EnglishInfo)
  ||| ...
  endproc
  process ClientProg[g](lang) :noexit:=
   g?data;
     ([Language(data) = lang]; ...  (* display information *)
     [] [Language(data) <> lang]; exit (* skip information *)
     ) >> ClientProg[g](lang)
  endproc
  ...
 endagent
 agent S2 :noexit:=
 ...
 agent S3 :noexit:=
 ...
endspec
```

(Here, behavior expressions of processes $S2, S3$ and $Transmit$ are omitted. $StartTime()$ is an ADT function which becomes true when the service starting time has come)

keep receiving service.

Like this example, by sending/receiving process names, programs can be down-loaded on demand to each mobile host. It contributes efficient use of memory in mobile hosts with poor resources. Also, using properties of multi-way synchronization, we can develop an application where the multiple users interact with a base station at the same time cooperating with each other (e.g., interactive games, quiz competition, and so on).

2.4.2 Routing in Wireless Ad Hoc Networks

In wireless ad hoc networks, each mobile agent can communicate with another distant agent via some intermediate agents by repeating broadcasting to their radio ranges. Here, based on Dynamic Source Routing [23], we describe a routing protocol to find a path from a source agent to a destination agent in wireless ad hoc networks.

In this protocol, as shown in Fig. 2.6, when an agent (source agent) wants to obtain a path to a particular agent, it broadcasts a request message to its radio range. Each agent which has received the request message re-broadcasts it. In such a way, message flooding is carried out until reaching the destination agent. The request message includes the path from the source to the current node (each intermediate agent adds its ID as the last entry of list variable $route\_record$). Since the message includes the complete path from the source to the destination when it has reached to the destination agent, then it returns the path information to the source along the reverse direction of the path.

At each stage, each agent executes one of the following action sequences.

(1) at a source node, it inputs the destination agent's ID ($u?dest\_id$) and broadcasts a request message asking for a path to the destination.

(2) at an intermediate node, it receives the request message and forwards it by re-broadcasting.

(3) at a destination node, it receives the request message and returns the complete path information to the source node.

(4) at an intermediate node, it receives the return message and forwards it towards the source node according to the return path.

(5) at a source node, it receives the return message and provides the obtained path to its user ($u!route$).

A message broadcast by an agent (sender) can be described by a multi-way synchronization between the sender and other agents in its radio range. So, here, we describe the sender agent to advertise for other agents by executing $sync\,!\{b\}$ repeatedly during a time interval, and to broadcast a message by multi-way synchronization among agents which has participated in the advertisement. The above broadcast mechanism can be described as the following LOTOS/M process.

```
process Broadcast[b](msgtype, dest_id, data): exit:=
  sync !{b}:sid!msgtype in
    [not(TimerExpired())]− >
      Broadcast(dest_id, msgtype, data) >> disc!sid; exit
  [] [TimerExpired()]− > b!msgtype!dest_id!data; disc!sid; exit
  endsync
endproc
```

(here, $TimerExpired()$ is an ADT function which becomes true when the preset timer has expired)

Among the above behavior, in (2), to avoid the message replication, each agent must forward the message only when the current path attached to the message does not include itself ($not(included(route\_record, my\_id))$).

In (3), (4) and (5), each intermediate agent must send the path information only to the next agent in the return path towards the source agent. So, the ID of the next agent is used as the I/O parameter in *sync* action so that only the intended pair of agents can be combined ($sync\ !\{b\}!Return!last(route\_record)$ and $sync\ ?\{a\}?msg?id[msg = Return\ and\ id = my\_id]$).

The whole description of the agent behavior in LOTOS/M is shown in Table 2.6.

2.5 Implementation of LOTOS/M Specifications and Experimental Results

2.5.1 LOTOS/M Compiler

We have developed a LOTOS/M compiler where a given LOTOS/M specification is implemented as a set of Java programs executed at the corresponding nodes, respectively. Since we aim at modeling behavior of wireless mobile systems, only the control part of a given specification is automatically implemented. We assume that functions used in the specification (which are supposed to be described in abstract data types in LOTOS [5]) are available as the corresponding "methods" in the given Java class libraries and Java programs generated by our compiler just invoke the methods appropriately. Here we outline the basic ideas on how to establish multi-way synchronization channels and execute events synchronously among mobile agents.

(1) Dynamic establishment of channels As explained in Sect. 2.3, a multi-way synchronization channel among multiple agents is established incrementally. So, we adopt the following procedure: (i) the host agent broadcasts a message to advertise for a synchronization peer in its radio range periodically until receiving a participation message from at least one agent; and (ii) if it has received the participation messages from multiple agents, it selects one agent among them and sends the acceptance message only to the selected agent.

(2) Execution of events by multi-way synchronization In order to calculate what event tuples can be executed synchronously among agents when those agents request execution of events, we construct the *synchronization tree*

Table 2.6: Example specification of a path finding protocol based on Dynamic Source Routing

```
agent DSR[u](my_id):  noexit:=
hide b in
(
 (* (1) sending route request with destination node ID *)
 u?dest_id:int;
 Broadcast(Search,dest_id,{my_id})
[] (* (2) forwarding route request *)
 sync?{a}:sid?msg[msg=Search] in
 a!Search?dest_id?route_record
  [(dest_id<>my_id) and not(included(route_record,my_id))]);
 disc!sid;
 Broadcast(Search, dest_id, route_record+{my_id})
[] (* (3) when route request reaches the destination node *)
 a!Search?dest_id?route_record[dest_id = my_id];
 disc!sid;
 (sync !{b}:sid2!Return!last(route_record) in
  b!Return!route_record+{my_id}!route_record; disc!sid2; exit
 endsync
 )
endsync
[] (* (4) forwarding of route information *)
 sync?{a}:sid?msg?id[msg=Return and id=my_id] in
  a!Return?route?return_path
   [included(return_path,my_id) and return_path <> my_id];
  disc!sid;
  (sync !b:sid2!Return!last(return_path-{my_id}) in
   b!Return!route!return_path-{my_id}; disc!sid2; exit
   endsync
  )
 [] (* (5) when the source node receives route info.  *)
  a!Return?route?return_path[return_path={my_id}];
  u!route;
endsync
) >> DSR[u](my_id)
endagent
```

($included(list, item)$ and $last(list)$ are ADT functions calculating whether $item$ is included in $list$ or not and returning the last item of $list$, respectively. Exception behavior by executing $pdisc$ is omitted.

for each agent group. Here the synchronization tree corresponds to the syntax tree where each intermediate node corresponds to an ad hoc parallel operator and each leaf node does an agent. To enable communication along the syntax tree, we let the host agent to be the *responsible node* where it receives request messages from the participant agent and evaluates the synchronization condition for each pair of events requested from the host agent and the participant agent on given gate list G. Since a responsible node is assigned to each operator node in the tree, according to standard LOTOS semantics, the executable events can be calculated by examining conditions at each intermediate node along the path from the leaves to the root. In [24], we have proposed an implementation method of standard LOTOS and a compiler where a similar algorithm is used to check executability of multi-way synchronization. So, the above algorithm could easily be implemented by extending the algorithm used in our existing compiler.

(3) Isolation of agents To handle this issue, the mechanisms (1) for detecting an agent (or a sub-agent group) isolated from the agent group and for (2) reconstructing the synchronization tree are required. For (1), we have implemented a mechanism to periodically send a polling signal to members of the agent group. In (2), there are some complicated cases for reconstructing the tree: e.g., when an intermediate responsible node has been isolated. For this case, we have implemented a detection mechanism to save some of children agents which are still alive (capable of communicating with at least one member of the agent group).

2.5.2 Experimental Results

With the Java programs generated by our compiler, we have measured (1) time for a channel establishment, (2) time for executing each event on the established channel, and (3) data transmission rate on the channel (we used four note PCs with MMX Pentium 233MHz to Celeron 333MHz on IEEE 802.11b wireless LAN, 11Mbps).

For (1), it took about 2.6ms for a channel establishment between a pair of agents. For (2), when the numbers of combined agents is 2 to 4, 60 to 140 events were executed among agents per second. For (3), the achieved rate was 0.9 to 2.3Mbps when the number of agents is two to four. Since data transmission rates between two agents when using http and ftp protocols on the same environment were 692.9 Kbps and 3471.9 Kbps, respectively, we think our compiler generate efficient code enough for practical use.

2.6 Conclusion

In this chapter, we have proposed a new language called LOTOS/M suitable for description and implementation of wireless mobile applications.

In LOTOS/M, we can describe dynamic establishment of multi-way synchronization channels among agents so that the agents which happen to meet in a

common radio range can dynamically communicate by multi-way synchronization. Also, LOTOS/M can naturally handle the case that some of the combined agents are dynamically isolated (e.g., by leaving from a radio range).

Through some experiments, we have confirmed that our proposed technique is enough applicable to describe and implement wireless mobile applications.

Since our LOTOS/M compiler generates Java programs, it will be easy to implement LOTOS/M specifications on cellular phones and PDAs which can execute Java programs with IEEE802.11 or Bluetooth interfaces (such devices are already available). As part of future work, we would like to develop more practical applications such as video conferences on ad hoc networks consisting of those devices.

Chapter 3

A Middleware Providing Multi-way Synchronization Method in Ad-Hoc Environments

3.1 Introduction

In recent years, technologies used in portable mobile devices such as cellular phones and PDAs have made remarkable progress. Nowadays, these devices can execute Java programs, communicate with other devices via short distance wireless channels (e.g., Bluetooth[29] and IEEE802.11), and know its geographical location by GPS. This background has been encouraging people to focus on ad hoc networks [23] which consist of multiple mobile hosts where each host can forward packets to enable long distance communication.

In an ad hoc network, unlike an ordinary communication facility based on the client-server model in the Internet, we need other kind of communication facilities such that a mobile host can dynamically search its communication peers and establish channels with them. In order to share information efficiently among a number of mobile hosts, it is important to be able to use multi-point channels which enable multicast distribution of data. Also, for easy handling of interactions among mobile hosts, mechanisms for synchronization and mutual exclusion among them are essential.

Among a lot of languages to describe communication protocols and distributed systems, ISO LOTOS [5] has a multi-point communication facility called *multi-way synchronization*. Multi-way synchronization enables several parallel processes to execute the specified events synchronously so as to exchange data values. With multi-way synchronization, we can specify not only synchronous execution of events but also mutual exclusion such that a speci-

fied process synchronizes with only one of the other processes exclusively at each synchronization time. So, we can easily handle complicated mechanisms such as broadcast/multi-cast communication and mutual exclusion in distributed systems with this facility. Moreover, multi-way synchronization also allows us to describe systems incrementally as a main behavior and a set of behavioral constraints (called the *constraint oriented style* [6, 7]). From these reasons, multi-way synchronization seems useful to design and develop wireless mobile systems. However, multi-way synchronization in LOTOS does not support dynamism in wireless mobile applications such as dynamic join and leave to/from the current communication channels.

In this chapter, we propose a Java-based middleware that provides facilities for dynamic establishment of multi-way synchronization channels among multiple mobile hosts in ad hoc networks.

The program of each agent can be described in Java where communication among agents is specified with the methods of the proposed middleware. In order to establish of multi-way synchronization channels dynamically, the middleware provides the following methods which assign a specified synchronization relation to a pair of agents when they are in a state capable of communicating with each other: (i) advertisement for a synchronization peer on a channel list (a list of channel names) and (ii) participation in the advertised synchronization. The pair of combined agents (agents with synchronization relation) is regarded as a single agent, and can combine with another agent on another channel list (the same channels can be used to be shared among more than two agents). The group of combined agents is called the *agent group*, and its member agents can communicate with each other by synchronously executing events on specified channels until the synchronization relation is canceled. When an agent (or a sub-agent group) goes in a state incapable of communication with the other agents of the same agent group, the synchronization relation assigned to the agent is canceled and it can run independently of the others.

When implementing the above mechanism, it is important to manage the information about member agents in each agent group and the synchronization relations assigned among the member agents to calculate what events can be executed synchronously among those agents. For the purpose, we represent the synchronization relations assigned to an agent group as a binary tree where each intermediate node corresponds to a binary synchronization relation and each leaf node does an agent, and let the agents keeps the latest tree information in a distributed manner so that events executed synchronously among the agents can be calculated based on the tree. When an agent (or a sub-agent group) goes in a state incapable of communicating with the other agents of an agent group, the synchronization tree is reconstructed so that the remaining agent group can proceed without the isolated agent. We have implemented a polling mechanism to detect whether each agent has gone into such a state incapable of communication or not.

Using the proposed middleware, we have developed a simple video conferencing application on a wireless network where only one of participants can transmit his/her video data to the others exclusively. As a result, we have confirmed that

such a mobile application with data distribution and mutual exclusion can easily be developed using the middleware and achieve practical performance. Also, some experiments on IEEE 802.11b wireless LAN have shown that the overhead of channel establishment is small enough for practical use. The average data transfer rate and the time to execute each synchronization are also reasonable under general conditions.

3.1.1 Related Work

A lot of middleware for mobile applications have been proposed in recent years. They can be classified into four categories: (1) middleware using peer-to-peer (P2P) communication facility [30, 31] (2) extensions of existing middleware/ protocols [32, 33, 34, 35], (3) context-aware middleware [19, 36, 37, 38, 39] and (4) mobile agent based middleware [40, 41, 42].

For category (1), Proem [30] provides components for instant messaging services, file sharing and P2P communication. In order to achieve good interconnectivity among different platforms, Proem provides (i) one transport protocol for connection-less asynchronous communication on ad hoc networks, and (ii) three higher level protocols for presence announcement of each mobile host, for file/data sharing, and for community construction.

With respect to multi-casting of data, the proposed communication facility is similar to P2P. However, our communication facility enables data synchronization and mutual exclusion among agents. It is the main difference.

For category (2), a middleware based on publish/subscribe paradigm (i.e., autonomous components interact with event notifications) has been proposed [32].

The proposed middleware has some similarities in the sense that each agent subscribes advertisement/participation requests for establishing synchronization channels, and that the disconnection of the established channel is notified by the corresponding event. However, the proposed middleware provides not only subscription/notification but also synchronization, data transfer and mutual exclusion in a consistent framework.

For category (3), [38] has proposed middleware services for information dissemination in mobile wireless networks where each mobile host has its own preference and the selective information is retrieved from data bases based on the preference and the location information. Also [37] provides several components to be used in mobile application developments which provide location-aware information retrieved from distributed nodes in the Internet.

Our middleware aims at providing communication channels based on multiway synchronization among agents on ad hoc networks. We think that information retrieval based on location information can be described as applications of the proposed middleware.

For category (4), SOMA [40] has been proposed as Java-based platform based on the mobile agent technology. SOMA architecture consists of four layers. The highest layer is called “mobility middleware” which provides highly abstract services such as virtual resource management like in CORBA. The next

layer called "core services" provides services such as communication, migration, security and QoS adaptation. The lower layers correspond to JVM and physical devices, respectively.

SOMA aims at providing high level services, while our middleware aims at providing primitive communication facilities. This is the main difference. In our middleware, similarly to mobile agent based middleware, we can let object code to be transferred between agents using multi-way synchronization. With the proposed middleware, we think we can develop more functional middleware like SOMA.

3.2 Proposed Middleware for Mobile Applications

In this chapter, we propose a communication middleware for mobile applications in Java that is based on multi-way synchronization of a formal description language LOTOS [5].

3.2.1 Multi-way Synchronization

In LOTOS, system specifications are described by a parallel composition of several processes. The processes are constructed with external (input/output) events that occur at interaction points called *gates* and internal events. For example, an output event at gate g is described as $g!E$ where E is the output data, and an input event at gate g is described as $g?x : int$ where $x : int$ is an integer variable for input data .

The interaction of processes are specified by synchronization operators $P|[G]|Q$ where P and Q are processes and G is a list of gates. The processes combined by synchronization operators synchronously execute the events at the specified gates where each output value from a process is assigned to all the input variables in the other processes. Here, the type of the output value must match the type of the input variables.

When each process includes several alternative events, multiple combinations of synchronizing events may become executable. In this case, one of them must be selected non-deterministically. For example, in $P|[a]|Q$ where process P can execute $a!1$ or $a?x : string$ and process Q can execute $a!"hello"$ or $a?y : int$, combinations of synchronizing events are (i) $(a!1, a?y)$, (ii) $(a?x, a!"hello")$. Here, (i) or (ii) is selected nondeterministically. Other combinations such as $(a!1, a!"hello")$ are excluded since their types do not match.

Also, in LOTOS, the guard expression can be specified to events like $g?x : int[x > 0]$. In this case, this event can synchronize only with the events which output an integer value greater than zero.

If a pair of two processes is combined by a synchronization operator, the pair can be regarded as a new process. Such pairs of combined processes can be combined hierarchically with other processes. Since the hierarchical combination

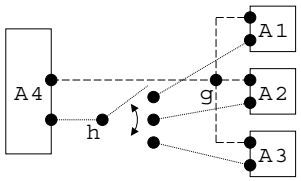


Figure 3.1: Multi-cast communication and exclusive control

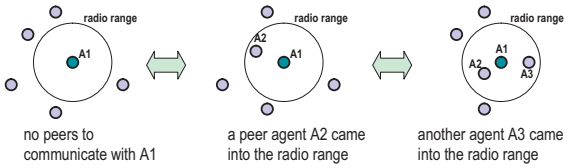


Figure 3.2: Dynamic change of agent tuples capable of communication

of processes enable the synchronization among more than two processes, this facility is called *multi-way synchronization*.

For example, assume that four processes $A1, ..., A4$ are combined by synchronization operators as $((A1|[g]|A2)|[g]|A3)|[g,h]|A4$. In this case, since the events at gate g of these processes are executed synchronously, we can specify multi-cast data transfer to all processes by the events at gate g. In this specification, gate g can be regarded as the common gate among these processes (Fig.3.1). On the other hand, for gate h, process $A4$ and the entire process $((A1|[g]|A2)|[g]|A3)$ is specified to synchronize by the events at gate h. Since the processes $A1$, $A2$ and $A3$ execute the events at gate h independently, only one of them is selected to synchronize with the process $A4$ by the events at gate h as shown in Fig.3.1.

3.2.2 Proposed Middleware and its Facility

In LOTOS, multi-way synchronization is specified with synchronization operators among pre-defined processes. However, in a wireless mobile environment, quite numerous agents are running and they move around towards different directions. Thus, peer agents with which each agent can communicate may change since its communication area as well as other agents' areas changes as shown in Fig. 3.2. In order to apply multi-way synchronization to such a system, we propose facilities to dynamically assign synchronization relations among agents (which correspond to establishing multi-way synchronization channels) step by step.

We propose a middleware library for Java language such that Java programs (agents) running on the corresponding mobile hosts can communicate with each

other by multi-way synchronization on dynamically established communication channels. This library provides two methods for channel establishment (**Advertise** and **Participate**), and one method (**Disc**) for channel disconnection. Also we have introduced a mechanism which enables exchange of Java object codes among agents so that each agent can download an object code and execute it without compiling. We have implemented a base class called **Function** which simplifies the procedure consisting of (i) sending object codes as data, (ii) extracting Java codes from the received data, and (iii) execution of the extracted codes.

Below, we show the facilities of the above methods in the proposed middleware library.

(A) Channel Establishment

The methods for channel establishment are used in the following form:

sid:= **Advertise**(G, IO, Guard)
sid:= **Participate**(H, IO, Guard)

Here, G and H denote a list of channel (gate) names (such as {a,b,c}), "IO" represents a list of input and output parameters, and "$Guard$" represents a boolean expression denoted by $[f(c1, c2, ...x1, x2, ...)]$ where constants $c1, c2, ...$ and parameters $x1, x2, ...$ in IO may be used. IO and $Guard$ restrict agents which can participate in a given multi-way synchronization channel. For example, if we can use the location information by GPS, we can specify a guard expression so that only agents whose distances are less than $10m$ can establish a channel. "IO" and "$Guard$" can be empty. These methods return the ID of the established communication channel. The same value is assigned as IDs to the pair of agents which have established a channel.

If the following conditions hold for a pair of agents, then a synchronization relation on gate list G is assigned between the agents (we also say that the two agents are *combined* (or *joined*) on gate list G).

- one agent $\mathcal{A}_1$ calls the synchronization method **Advertise** (G,IO1, Guard1) (called *host agent*) and the other agent $\mathcal{A}_2$ calls **Participate** (H, IO2,Guard2) (called *participant agent*).
- the numbers of gates in G and H are the same.
- the numbers of parameters in $IO1$ and $IO2$ are the same and each pair of the corresponding parameters consists of an input and an output where their types match.
- both of $Guard1$ and $Guard2$ hold after assigning each output value to the corresponding variable of the input parameter (e.g., when the input and output parameters are x and E, respectively, the value of expression E is assigned to variable x).

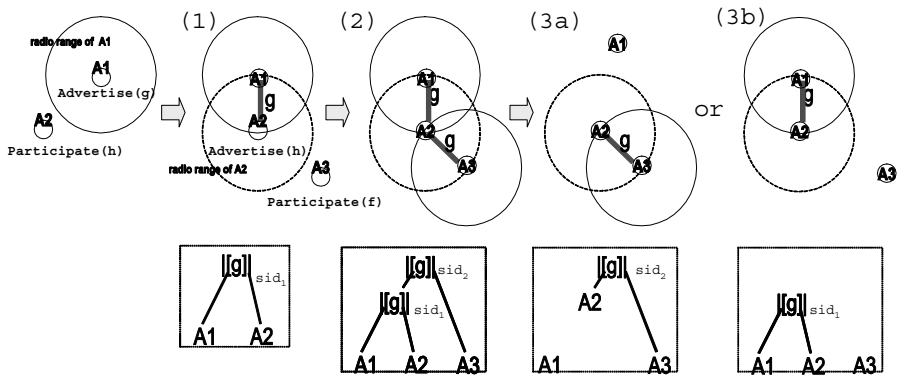


Figure 3.3: Assignment/cancellation of the synchronization relation among agents when they approach/leave into/from a radio range

- the two agents are in states capable of communicating with each other directly (i.e., in a common radio range).

If all of the above conditions hold, a synchronization relation on G is assigned to $\mathcal{A}_1$ and $\mathcal{A}_2$ (we say $\mathcal{A}_1$ and $\mathcal{A}_2$ are *combined*) and they behave as a single agent represented by $\mathcal{A}_1 |[G]|_{sid}\ \mathcal{A}_2[G/H]$. Here, $|[G]|_{sid}$ is the new operator called *ad hoc parallel* operator which is equivalent to the synchronization operator of LOTOS except that its operands can be separated by $Disc(sid)$ method. $A2[G/H]$ represents that gate list H is replaced by G, and that the program code in $A2$ referring each gate in H must refer the corresponding gate in G.

The combined agents are treated as a single agent called an *agent group*. Each agent group can be combined incrementally with another agent by executing **Advertise** (or **Participate**) method. As an example, Fig. 3.3 illustrates that three agents are combined in a step-by-step manner. First, (1) two agents $A1$ and $A2$ are combined on gate g with ID= sid_1. Then (2) another agent $A3$ is approaching to $A2$ and establishes a channel with ID= sid_2. In that situation, if A1 executes Disc(sid_1), (3a) the channel with sid_1 will be disconnected and the agent $A1$ will be isolated. On the other hand, if A2 executes Disc(sid_2), (3b) the sub-agent group $A1|[g]|_{sid_1} A2$ will leave from the agent group.

(B) Channel Disconnection

Each agent group can be separated into several agents/agent groups by active/passive disconnection events.

When an agent calls **Disc(sid)** method, the channel established with ID=sid is disconnected (called *active disconnection*). When a channel is disconnected (including cases that an agent has gone in a state incapable of communicating with any other members of the agent group), an exception **PDisc(sid)** is thrown

to all member agents of the agent group (called *passive disconnection*). This exception indicates which channel has been disconnected by *sid*.

(C) Exchange of Processes as Data Values

In Java, since Java codes such as classes and their instances can be regarded as data, they can be exchanged among different Java programs running on different hosts. Based on this mechanism, we have implemented a facility to specify such exchange of object codes simply. With this facility, when a client requires some functions, it can dynamically download them from servers.

If Java program A wants to execute some object code in a different Java program B, the following steps are required: (1) retrieve the file including the specified class (object code); (2) read the file in the internal buffer and transfer it; (3) receive data and extract the object code from it; and (4) execute the object code. Here, steps (1) and (2) should be done by Java program A, and (3) and (4) by B.

To avoid describing the above steps for every object code transfer, in our middleware library, we have defined a base class **Function** for exchange of object codes. We have implemented a mechanism to automatically apply the above steps to instances of subclasses of **Function**. Using this mechanism, we can transfer classes to multiple agents by method **Synchronization** explained later.

(D) Execution of Events by Multi-way Synchronization

Class **Event** is used to transfer data through dynamically established channels. We set each instance of class **Event** to hold a gate name, a list of I/O parameters and guards. When an agent gives an instance of class **Event** with parameters to method **Synchronize**, it is informed of whether these events can be executed according to the synchronization relations and given guard expressions in the agent group where the agent belongs. We show an example of a client-server application below. Here, assignments of parameters to the instances of **Event** are omitted.

```
void Server(){
    while(true){
        Advertise(G,IO,Guard);              //Channel Open
        Event = Synchronize(Events);        //Sending Data
    }
}

void Client(){
    if(Participate(G,IO,Guard)){            //Channel Open
        while(true){
            Event = Synchronize(Events);    //Receiving Data
        }
    }
}
```

In the above example, suppose that one server computer executes method **Server** and several client computers execute method **Client**. The server executes method **Advertise** repeatedly to find clients in a radio range which want to participate in the agent group. Each client tries to find a server in a radio range by executing method **Participate** when it wants to receive some data. When the server has been found, a channel is established. Since gate list G is used to establish channels among multiple clients in method $Server$, when the server executes an event by method **Synchronize**, data included in the event is transferred to all clients at the same time.

A parameter list is represented as an array of type **Object** (the superclass of all the classes in Java). For an output parameter, the value itself is stored in the element. For an input parameter, an instance of class **Class** (the class for representing classes) is stored. A guard is stored in an element with class **Function**. For each guard, a subclass of class **Function** is defined by overriding its method **Execute**. Guard expressions are transferred to other agents when synchronization condition is evaluated in **Synchronize** method.

3.3 An Example Mobile Application

In this section, we show applicability of our middleware by describing a simple video conferencing application on an ad hoc network.

We assume the following features in the application.

(1) Each user can join the current conference session or leave from the session anytime it wants.

(2) Due to bandwidth restriction, only one participant can talk at a time. If several participants want to talk at the same time, one of them must be selected under the consensus of all participants.

(3) While a participant is talking, his/her live video is transferred and played back on all of the participant hosts.

For the above (1), each agent executed on a participant host can establish a multi-way synchronization channel with other agent to form an agent group by repeatedly executing methods **Advertise** and **Participate** of our middleware. Two gates g and h are used for channel names where 'g' is used for exchanging the control data and gate 'h' for the media data.

For (2), we let each participant select a behavior mode **Talk** or **Listen**. To permit only one participant to talk, we use multi-way synchronization on gate g as follows. If a participant has selected **Listen**, the agent of the participant executes **Synchronize** with input event $g?id$. If a participant has selected **Talk**, the agent executes **Synchronize** with two alternatives consisting of input event $g?id$ and output event $g!ID$ where ID is the identifier given to the participant. By definition of multi-way synchronization in Sect. 3.2.1, only one tuple of synchronizing events such that one is $g!ID$ and the others are $g?id$ is selected even

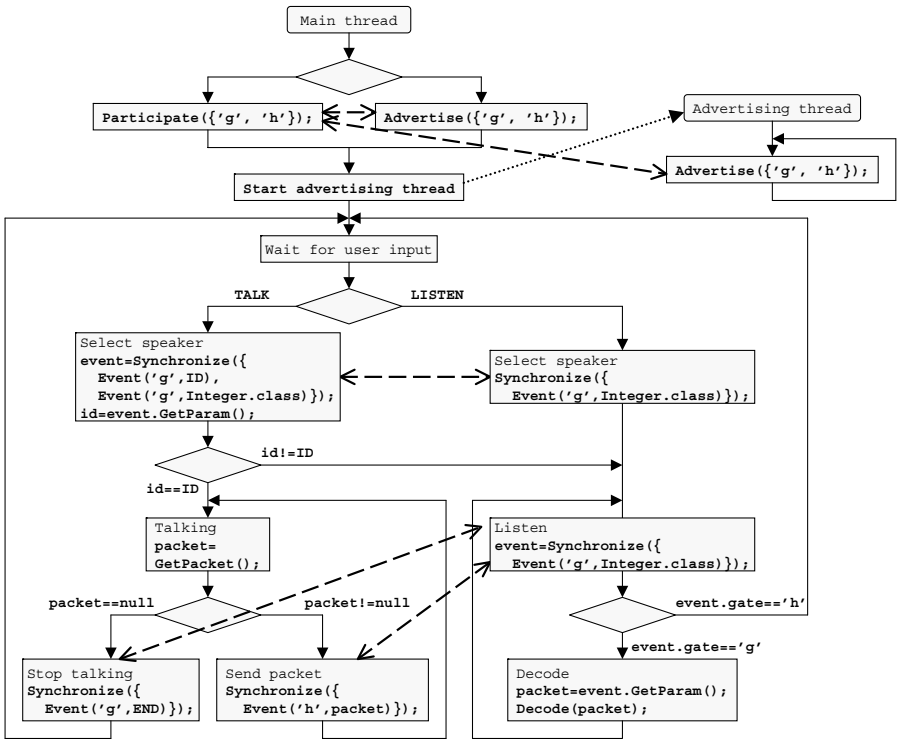


Figure 3.4: Example Java program

when several participants have selected **Talk**. Each participant knows whether his/her request to talk is permitted or not by the return value of **Synchronize**.

For (3), we use multi-way synchronization on gate h so that the live video data of the talking participant is transferred to the other participants by executing **Synchronize** with event $h!packet$ at the talking participant and **Synchronize** with event $h?packet$ at the listening participants.

In Fig. 3.4, we show the flow chart of the example Java program for the above application. Also, in Fig. 3.5, we show a snapshot when executing this program. Here, we introduce two new methods for multimedia processing operations: (i) a method **GetPacket** that captures the live video of a user as MPEG movie and packs it into packets (byte arrays) and (ii) a method **Decode** that decodes given packets and plays back the movie in a window. In this figure, the code for manipulating GUI such as construction of windows and buttons is omitted.

First, this program invokes two threads. One thread executes method **Advertise** continuously to advertise for participants of the conference session in its radio range. Another thread executes method **Participate** to establish a channel with other agents in the radio range to participate in the session. When

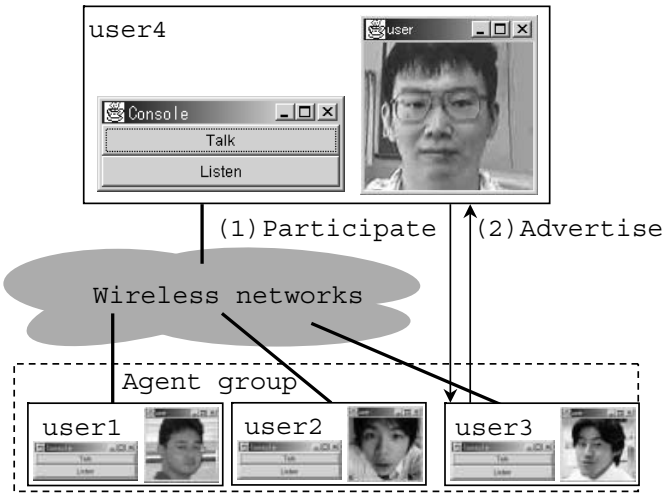


Figure 3.5: Snapshot of a sample application

channels are established, a synchronization relation of two gates g and h is assigned to the pair of agents. After the channel establishment, the operations for the above (2) and (3) are followed.

As explained above, using facilities of our middleware based on multi-way synchronization, we can simply describe applications including complicated interactions such as multi-cast with mutual exclusion.

3.4 Implementation

In this section, we focus on how to implement multi-way synchronization among agents.

3.4.1 How to Implement Multi-way Synchronization Among Agents

Since agents are combined incrementally, the combined agents (agent group) and the synchronization relation assigned among them are represented by a binary tree as shown in Fig. 7 where each intermediate node and each leaf node correspond to an ad hoc parallel operator and an agent, respectively. We call such a tree as the *synchronization tree*, hereafter. Since the synchronization tree is equivalent to the behavior expression in LOTOS, we can derive what events can be executed synchronously among agents by evaluating synchronization condition at each operator node in the tree from the leaf to the root node based on the technique in [24].

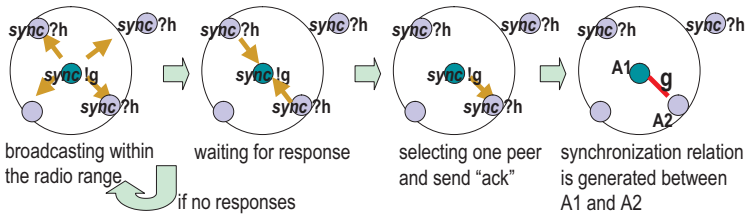


Figure 3.6: How to implement the channel establishment

How to Implement Method Advertise/Participate

By definition, if a host agent is advertising for a synchronization peer by Advertise (G,IO1,Guard1), a participant agent wants to participate in it by Participate (G,IO2,Guard2), and they can directly communicate with each other, a synchronization relation on G is assigned between them.

As shown in Fig. 3.6, we adopt the following procedure to check whether two agents can directly communicate with each other:

(1) the host agent broadcasts a message to advertise for a synchronization peer in its radio range.

(2) it repeats the same message broadcast periodically until at least one agent responds to the message for participation.

(3) if it has received the responses from multiple agents, it selects one agent among them (guard expressions must hold) and sends the acceptance message only to the selected agent. After that, we suppose that a synchronization relation has been assigned between those two agents.

(4) the host agent and the selected participant agent keep the information to set up the synchronization relation between them.

(5) the unselected agents wait another advertisement broadcast from the host agent (or from the selected participant agent) to participate in the same agent group.

For the above (4), we let the host agent to be the *responsible node* where it receives request messages from the participant agent and evaluates the synchronization condition for each pair of events requested from the host agent and the participant agent on given gate list G. Here, the responsible node (the host agent) keeps the gate list whose events must be synchronized and the other node (the participant agent) keeps the host agent to which it should send request messages. As shown in Fig. 3.7, when multiple agents are combined hierarchically, we similarly assign a responsible node to each ad hoc parallel operator generated by executing method Advertise/Participate. Like this way, the combined agents keep the synchronization relation specified among them.

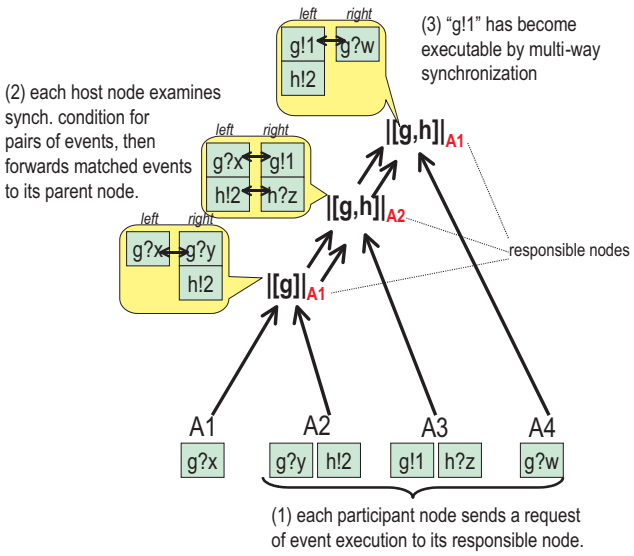


Figure 3.7: How to evaluate executable events in multi-way synchronization

How to Implement Method Synchronize

Suppose that agents $A1$, $A2$, $A3$ and $A4$ have been combined on gate g and the responsible nodes have been assigned as shown in Fig. 3.7. In the figure, $|[g,h]|_{A2}$ shows that agent $A2$ is the responsible node of the operator. Note that ID of each operator is omitted here. Here, we explain how an executable event of multi-way synchronization on gate g is calculated using Fig. 3.7. First, when agent $A2$ wants to execute event $g?y$ or $h!2$ by method Synchronize, it sends to its responsible node ($A1$) the request message for the events ($req(\{g?y, h!2\})$). $A1$, in turn, evaluates the synchronization condition between the events in the request message and event $g?x$ which $A1$ wants to execute. In this case, the synchronization condition holds between $g?x$ and $g?y$, and $h!2$ does not need synchronization in this node. Thus, $A1$ sends the request message for those events ($req(\{g?x, h!2\})$) to the parent responsible node (i.e., $|[g,h]|_{A2}$).

$A2$ receives the request messages from $A1$ and $A3$, and it similarly evaluates the synchronization condition for the messages. In this case, since $req(\{g?x, h!2\})$ and $req(\{g!1, h?z\})$ are received from $A1$ and $A3$, respectively, a new request message $req(\{g!1, h!2\})$ is generated and sent to the parent node ($|[g,h]|_{A1}$). The responsible node of $|[g,h]|_{A1}$ receives the request messages $req(\{g!1, h!2\})$ and $req(\{g?w\})$ from $A2$ and $A4$, respectively. Since the root responsible node ($A1$) does not have a parent node in the synchronization tree, $A1$ concludes that event $g!1$ is executable by multi-way synchronization, and the calculation result is propagated to the related agents which sent request messages along the synchronization tree. This result is returned to method

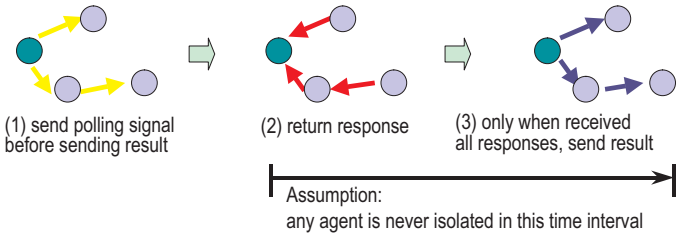


Figure 3.8: Check of physical connection among agents by polling signals

Synchronize.

In Fig. 3.7, the responsible nodes are $A1$, $A2$ and $A1$, respectively. If we can change the responsible node A2 to A1, the message exchanges between responsible nodes are reduced.

How to Handle Cases When an Agent is Isolated from its Agent Group

An agent is isolated from the agent group when it leaves from the radio range of the group (*passive isolation*) or when an agent intentionally executes method Disc (*active isolation*). We define passive isolation of agents as follows.

- when two directly combined agents cannot communicate with each other.
- at a synchronization operator node in the synchronization tree, any pair of agents from its left child subtree and right child subtree cannot communicate.

In this case, we suppose that the agent group whose members cannot communicate with its parent node has been isolated (then a corresponding method Disc is executed).

(1) How to handle agent isolation while calculating an executable event of multi-way synchronization In general, the combined agents of an agent group send their request messages at different time from each other. Therefore, there is a time lag to obtain the result of whether the requested events can be executed or not after sending the request. In our implementation method, as shown in Fig. 3.8, we adopt the following procedure to handle agent isolation while calculating multi-way synchronization:

(1) after calculation of an executable event of multi-way synchronization at the root node, the root node sends the polling signals to the member agents (i.e., agents which sent the request messages for the enabled synchronization).

(2) each member agent responds to the polling signal if it receives the signal.

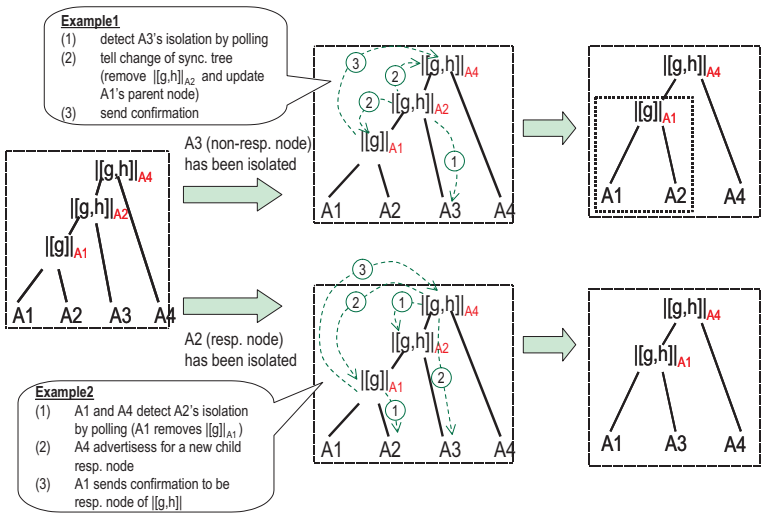


Figure 3.9: Reconstruction of the synchronization tree

(3) only if the root node receives the responses from all of the member agents, it sends the result. Otherwise, it removes the request messages from the agents which did not respond. The calculation of an executable event is carried out from the scratch for the remaining messages and new messages to be received.

If we assume that agents are never isolated while each agent is waiting to receive the result after responding to the polling signal, we can guarantee that multi-way synchronization among agents is correctly implemented even when agents move around. As shown in Sect. 3.5, the time necessary for the synchronization is less than one second. So we think this assumption could be reasonable.

(2) re-construction of the synchronization tree caused by agent isolation When an agent is isolated, the synchronization tree must be re-constructed.

(i) the case when a non-responsible node is isolated Let us suppose that agent $A3$ is isolated in Example1 of Fig. 3.9. Agent $A2$ (responsible node of $|[g,h]|_{A2}$) detects $A3$'s isolation by sending the polling signal. At that time, (1) $A2$ need not be responsible to $A3$ any more, so it should remove the work space for $|[g,h]|_{A2}$ in Fig. 3.7, and (2) the sub-tree whose root is $|[g]|_{A1}$ should be re-connected to the new parent node ($|[g,h]|_{A4}$ in Fig. 3.7).

In our implementation, for the above (2), agent $A2$ ($|[g,h]|_{A2}$) sends a message to $A1$ ($|[g]|_{A1}$) in order to change the parent node of $|[g]|_{A1}$ to agent $A4$ ($|[g,h]|_{A4}$) as shown in Example1 of Fig. 3.9.

Table 3.1: Main classes of our middleware library

Name	Method	Function
Agent	Agent(Communicator[] communicators);	base class of all the agents an agent class is instantiated with some Communication objects
	int Advertise(char[] channels);	advertise for an agent joining an agent group
	int Participate(char[] channels);	participate in an agent group
	Event Synchronize(Event[] events);	synchronize with other agents in the agent group with a possible event in the event list
Event	Event(char channel, Object[] parameters);	class for defining an event an event is defined by a set of channels and formal/actual parameters
Communicator		the abstract of communication functions used by our agents
Communicator-UDP		an implementation of Communicator class using UDP/IP
Function	Object Execute(Object [] parameters);	base class of the functions defining a class to send Java code overriding this method

If $A4$ and $A1$ cannot physically communicate with each other, we suppose that agents of the sub-tree whose root node is $|[g]|_{A1}$ have been isolated.
(ii) the case when a responsible node is isolated When agent $A2$ is isolated in Example2 of Fig. 3.9, $A4$ and $A1$ detect $A2$'s isolation by sending the polling signal. In this case, $|[g]|_{A1}$ should be removed and the responsible node of $|[g,h]|_{A2}$ should be changed to the one which can communicate with $A4$.

To do so, in our implementation method, we have each responsible node keep two sets: one set keeps agent names included in the left-hand sub-tree and another set does in the right-hand sub-tree. Based on the sets, when a responsible node is isolated, its parent node broadcasts a message to the agents included in the corresponding set in order to advertise for a new child node with the gate list for synchronization (we assume that each node knows the gate list for synchronization of its child node). In Example2 of Fig. 3.9, $A4$ ($|[g,h]|_{A4}$) broadcasts a message to $\{A1, A3\}$ for changing the responsible node of $|[g,h]|_{A2}$, and $A1$ responds to be a new responsible node. If no agents respond to the message, we suppose that the whole sub-tree has been isolated.

3.4.2 Implementation of Communication in Underlying Networks

We have defined class **Agent** that is the base class of all the agents based on our middleware (Table 3.1). We can design mobile agents simply by defining a new class extending class Agent and adding required functions to the new class.

The implementation of multi-way synchronization is hidden in class Agent and designers need not understand how the facility works. We have defined class **Communicator** and class **Message** in our middleware to abstract the facility of practical communication protocols such as broadcast of messages or transfer of messages to specified hosts. These abstracted classes construct virtual networks where message exchanges are carried out based on virtual host IDs. Class Agent works by using such abstracted communication protocols. We can use our middleware on various networks by implementing a subclass of class Comunicator using available protocols in the environments.

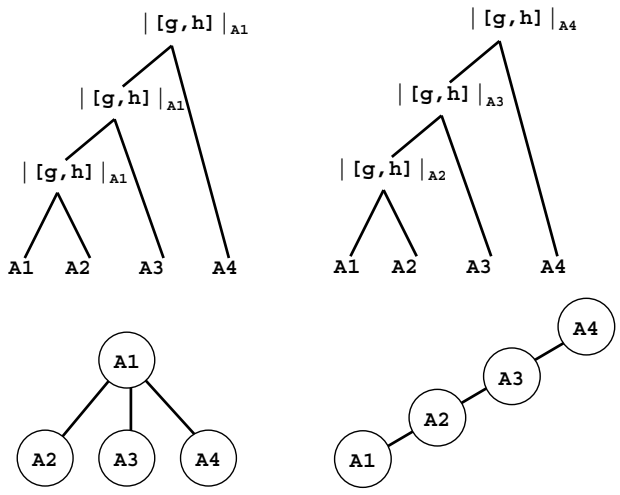
Here, we have implemented class **CommunicatorUDP** that is a subclass of class Communicator in order to examining our library. Class CommunicatiorUDP is based on UDP/IP to exchange the messages among hosts. This class assigns the virtual host IDs to actual IP addresses of the hosts and transfers messages via UDP.

At the present time, the facility of cellular phones are often restricted such that only http protocol can be used for data communication. However, we can implement the required facility of our class Communicator in such environments by using server side programs such as **CGIs** or **Java Servlets** that emulate the broadcast of messages and the transfer of messages to specified hosts. So we can implement our middleware on such cellular phones. Also we can implement them on **Bluetooth**.

We have implemented our middleware such that class Agent of our middleware can manage more than two instances of class Communicator. By using this facility, we can make gateway servers between different communication protocols by implementing subclasses of class Communicator on these protocols and giving them to the servers. For example, the agent program on wireless LAN environments in Sect. 3.3 can communicate only with the other hosts on the same environment. However if we implement a subclass of class Communicator for cellular phones based on http and make some server use them with the Communicator for the wireless LAN environments, the server enables the seamless participation to the network meeting for the cellular phones.

3.5 Experimental Results

We have examined the performance of our middleware. Since the time necessary for calculating executable synchronizations depends on the structure of the synchronization tree. If relatively large size of data is transferred by synchronization, the time also depends on the hop count (longest path length). Here, we have measured the time for two cases : (a) the case of most efficient tree and



(a)Most efficient tree (b)Least efficient tree

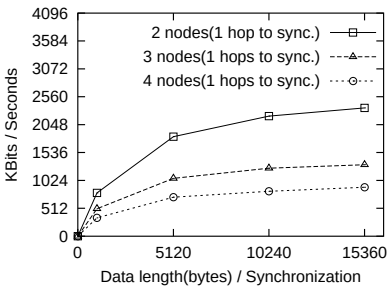
Figure 3.10: Structure of trees

(b) the case of least efficient tree (Fig.3.10). In our experiment, we used 4 nodes of mobile hosts corresponding to PCs with MMX Pentium 230MHz to Celeron 333MHz on a wireless LAN (IEEE 802.11b) environment where the maximum data transfer rate is 11Mbps.

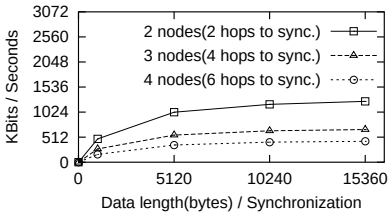
At first, we have measured the time for each channel establishment. The consumed time was less than 300 ms for each channel establishment, independently of the number of hosts in the agent group. Since each two nodes on a wireless LAN can communicate directly with each other, the time required for each channel establishment is converged to a fixed value even when the number of agents increases. In an environment where each pair of nodes cannot communicate directly with each other, the time for a channel establishment will increase as the number of agents increases. However, this overhead seems small and can be ignored for practical applications.

Next, we have examined the performance of synchronization on the established communication channels. For this purpose, we have measured the average data transfer rate and the average number of synchronizations executed in one second for some different data sizes (since other information such as gate names and guards is transferred with given data parameters, the indicated data size is smaller than the actual size of transferred data).

The results are shown in Fig.3.11 and Table 3.2. We have measured the average data transfer rate on the same environment by other protocols. The data transfer rate was 692.9Kbps by *http* and 3471.9Kbps by *ftp*. From these results, our middleware can transfer data at a reasonable speed when we select appropriate data size to be transferred at once.



(a)Most efficient tree



(b)Least efficient tree

Figure 3.11: Average transfer rate(bps)

Table 3.2: Average transfer rate for each data length per synchronization

Data length per sync. (Bytes)	Average data transfer rate(Kbps)					
	(a)Most efficient tree			(b)Least efficient tree		
	2 nodes	3 nodes	4 nodes	2 nodes	3 nodes	4 nodes
0†	0.0	0.0	0.0	0.0	0.0	0.0
1024	794.8	504.4	336.8	476.6	271.4	161.2
5120	1826.7	1062.6	716.7	1022.5	554.1	350.7
10240	2205.9	1249.2	824.4	1182.0	641.5	411.4
15360	2355.3	1312.4	899.9	1242.5	668.4	427.9

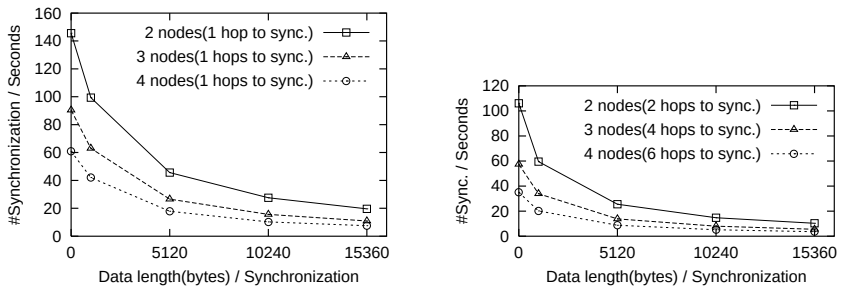
† : synchronize without data

Next, we have measured how many times synchronization can be executed in a second. The result is shown in Fig. 3.12 and Table 3.3. We think this result is practical enough for developing mobile applications on ad hoc networks.

3.6 Conclusion

In this chapter, we have proposed a Java-based middleware for supporting interactions among agents in wireless ad hoc networks based on multi-way synchronization of LOTOS. With our middleware, as explained in Sect. 3.3, we can easily handle group communication such as multicast data distribution and mutual exclusion among multiple agents by the methods of the middleware for dynamically establishing multi-way synchronization channels. Also, we can treat the case that some agents in an agent group are isolated by leaving from the common radio range.

In our implementation method, we manage membership information of each agent group as a binary tree based on LOTOS semantics, we could easily implement multi-way synchronization among agents by keeping the latest tree and collecting/processing request messages from agents along the tree based on the



(a)Most efficient tree (b)Least efficient tree

Figure 3.12: Average number of synchronization per second

Table 3.3: Average number of synchronization per second for each data length per synchronization

Data length per sync. (Bytes)	Average number of sync. per sec.					
	Most efficient tree			Least efficient tree		
	2 nodes	3 nodes	4 nodes	2 nodes	3 nodes	4 nodes
0†	145.6	90.4	60.8	106.1	57.5	35.0
1024	99.4	63.1	42.1	59.6	33.9	20.2
5120	45.7	26.6	17.9	25.6	13.9	8.8
10240	27.6	15.6	10.3	14.8	8.0	5.1
15360	19.6	10.9	7.5	10.4	5.6	3.6

† : synchronize without data

technique in [24]. Through some experiments, we have confirmed that our proposed technique is enough applicable to describe and implement wireless mobile applications.

Chapter 4

Automatic Decomposition of Java Program for Mobile Terminals

4.1 Introduction

Recently, handheld terminals such as mobile phones or PDAs have become very popular. And many additional functions have been given to such devices. Especially, since many mobile phones that equip Java Virtual Machine have appeared, the requirement for the applications running on mobile phones has been becoming larger. However, because of the limitations such as available memory size and processing power, not so many applications can be executed on such handheld devices. On the other hand, remote method invocation facilities such as Java RMI [43] and HORB [44] and distribution methods to execute Java programs in distributed environments by using such facilities have been studied.

In this chapter, we propose a distribution method where given applications will be executed in mobile terminals virtually by using remote method invocation. In case that a given application is too large to be executed on mobile terminals, only a part of the application (e.g. the user interface and frequently invoked modules), can be executed. So in our approach, only a part of the application that includes its user interface is executed on a mobile terminal and large sized modules that cannot be executed in the mobile terminal are executed on its proxy server. The two parts communicate each other by using remote method invocation and the whole system is executed as the same as the original application running on a single machine. In this case, the division is required to satisfy the limitation of mobile terminals and moreover it is better that the division satisfies the requirements from its user's environment. For example, if we use a cellular phone and if we must pay money for communication between the cellular phone and its proxy server depending on the amount of communica-

tion, the user would require to reduce the amount of communication in order to save money. On the other hand, the decrease of communication delay might be most important for real-time applications if we use mobile terminals with free wireless LAN services. Keeping the consumption of battery as low as possible is also important. So in this chapter, we propose a method for obtaining the division that is as good as possible in a sense of the given metrics under the given restrictions.

To derive such a division, the statistical data about the amount and number of communication among modules are needed. For remote method invocation in Java, many studies are done such as a study of measurement and optimization [45], studies of measurement and scheduling for improvement of real-time systems [46, 47] or a study of efficient implementation of RMI [48]. However, these studies are based on the measurement of performance of distributed applications in practical environments. So, it is difficult to apply those techniques to divide applications that are not specified as programs running on distributed environments.

For statistical performance evaluation, there are many studies based on analysis of source codes such as studies of slicing of parallel Java programs [11, 12]. However, here for simplicity of discussion, we use a simulation based performance evaluation technique. In our technique, additional codes for performance evaluation are inserted to the given source code automatically. The codes are inserted to be called before all the method invocations and record the amount and number of communication between two modules (classes). The statistical performance data can be collected by executing the modified code repeatedly with considering various situations.

Then a division is derived where it satisfies all the restrictions such as the memory size and it is optimized under the given metrics. Here, we have proven that this division problem is NP-hard. Since in general the optimized solution cannot be derived in practical time, we use a heuristic algorithm to get an approximated solution. To solve such NP-hard problems, there are many studies for approximation such as Min-Cut method [49] proposed by Kernighan and Lin, its advanced method [50] and a combination of Min-Cut based graph division algorithm and GA (generic algorithm) [51]. Moreover, SA(Simulated Annealing) [52], liner-time heuristic division method [53] and other methods are proposed. In our technique, we use SA based method because it is known that relatively good solutions can be derived in reasonable computation time.

We have developed a performance evaluation tool and division tool based on SA. Then we have applied our method to some example applications by some dividing metrics. By this evaluation, we have checked our method can reduce calculation time extremely by compared with brute force method and the derived division can be as good as the optimized answer by searching widely.

In the following Sect. 4.2, we give the outline of the proposed technique. In Sect. 4.3, we explain how we can collect the statical information about communication between each pair of two modules. In Sect. 4.4, we define the module assignment problem formally and prove it as NP-hard. In Sect. 4.5, we propose a division technique using SA. In Sect. 4.6, in order to show

the applicability of our technique, we describe typical mobile applications and divide their programs into two sets of modules. Finally, Sect. 4.7 concludes the chapter.

4.2 Outline of Proposed Technique

In our proposed technique, a given application is divided into two parts where each class is assigned to either server side modules or client side modules. These two parts of modules communicate with each other so that their behavior is the same as the given application.

4.2.1 Restrictions for Target Applications

Here, we give some restrictions for the target applications.

- The source codes of the target application are available.
- All the interactions between classes are done by method invocation.
- All the parameter variables are simple data structure that can be passed by value.
- After each method invocation, the called class never keeps the reference to parameter variables of the invocation.

At first, since in our method performance evaluation is done by modifying source codes, the source codes of the target application must be available. If a part of source codes of some classes is not available, these classes are not cared in division and they are placed on both the server side and client side (the source codes of the standard library are not needed since these classes are always available on both sides without explicitly assigned). Secondly, in our method, all the classes of target applications must interact with each other only by method invocation. If some shared variables are used for interaction, they must be replaced by access methods of private variables before applying our method. Thirdly, all the parameters must be passed as if they were passed by value. The special objects such as the objects with connections to the environments like sockets objects must have neither parameters or return values. This restriction is placed in order to measure the amount of communication correctly. Lastly, the references to parameter variables must not be kept after the method invocation because of the same reason as the third restriction. Even in the case that some classes keep the references to parameter variables, our method can be applied by defining wrapper classes for these classes that hide such complicated interactions into them.

4.2.2 Assumption for Target Environment

In our technique, the divided two parts interact with each other by using Java RMI, in target environment. RMI must be available between the server side modules and client side modules in both directions.

Now in many environments of mobile phones RMI is not supported and only pull type communication from terminals is served. However, push type communication from servers is defined in WAP2.0, which is the next generation of handheld communication device standard. So by using such services to simulate RMI, our technique can be applied. On the other hand, even in the current generation of cellular phones, push type communication from the server side can be available by using short mail services provided in some cellular phone environments such as C-mail service of ezplus provided by KDDI Japan. So by using such services as triggers, RMI can be simulated in these environments.

4.3 Estimation Method of Statistical Information for Optimal Division

In our method, we collect statistical data to divide given applications by simulation. By inserting special codes to given source nodes of applications that collect statistical information, and then by executing the inserted codes, the simulation is carried out.

4.3.1 Statistical Information and Optimizing Parameters

At first, we define the target of optimization as follows.

- amount of communication
- communication delay
- power consumption in handheld devices

Here, we assume each parameter is estimated as follows. The amount of communication is evaluated by measuring the communication between modules. The communication delay is approximated by the number of method invocation, since the delay is mainly caused by the overhead of remote method invocation. We assume the power consumption is in proportion to the amount of codes on handheld devices. So we estimate the power consumption by the duration that the modules are executed in handheld devices. Moreover, since the handheld devices often has much less power than servers, we can make the divided application faster by assigning the modules consuming much time into the server side. In our technique, the optimization process is based on an objective function that is constructed as the weighted summation of these three parameters in order to get various optimized division according as we need. The weight should be decided according to characteristics of the target environment such

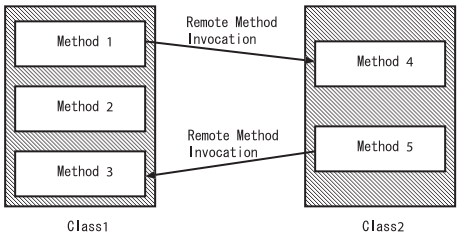


Figure 4.1: Remote method invocation

as relation between available time and usage of wireless connection determined by capacity of the battery [54].

Hereafter, we show our evaluation method.

Estimation of the Amount of Communication

Assume that there are two classes as shown in Fig. 4.1, and they are placed on different machines where all interactions are done by remote method invocation. In this paper, we use the summation of size of data exchanged by these remote method invocations through the execution of the application as the total amount of communication. Since the target of our method is standalone Java applications, we must estimate it by assuming that all methods are invoked through remote method invocation and by measuring the amount of data exchanged by method invocations.

Here we define the amount of communication of a remote method invocation as *data size of parameter values + data size of the return value + overhead of remote method invocation*. For example, by assuming that the overhead of remote method invocation costs 20 bytes and a method is called with two integer typed parameters (4 bytes for each value) and returns one long typed value (8 bytes), the amount of communication for this method invocation comes to $4+4+8+20 = 3$ bytes. Since the actual amount of communication is slightly different in each execution of application, we must use the averaged data after executing the application many times with practical usage.

Estimation of Communication Delay

In our technique, we assume that communication delay is mainly caused by the overhead of communication and estimate it by the number of remote method invocations simply. In practice, since the communication delay depends on the data size and their transmission speed, we must coordinate the weight of the amount of communication and the number of remote method invocations.

Estimation of Time Spent by Each Module

In order to estimate the power consumption of a class, we measure the time spent by the methods of that class. Here we use a simple method to measure them to implement easily. The additional codes for the measurement record the current time before each method invocation and the duration from the recorded time and the time when the invocation is ended. It is considered as the time spent by the method invocation (If other methods are called from that method, the time spent by them is taken away). In this method, we assume that there is no other large load on the environment for measurement, and the target application is not multi-threaded. If we cannot assume such a condition, we must measure them by using more detailed profiling tools such as Extensible Java Profiler [55].

4.3.2 Insertion of Measurement Codes

In our method, to measure the above three profiles, the measurement codes are inserted to the given application automatically as follows.

1. Define a class that holds the recorded information and write down after the execution. This class consists of the following elements.
   - A data array to hold the recorded information.
   - Some methods to measure and record the statistical information.
   - A method to output the result.
2. Insert the statement to call the measurement method before each method invocation.
3. Insert the statement to call the method that outputs the result at the end of the target application.

```
class Main {
 public static void main(String args[]) {
       :
  Result.AddData(CLASS_Main, CLASS_Sub1,
                 RMI_OVERHEAD+4);        // Inserted
  sub1.method1(x);
  Result.AddData(CLASS_Main, CLASS_Sub2,
                 RMI_OVERHEAD+8);        // Inserted
  sub2.method2(y);
       :
 }
}

class Sub1 {
 public void method1(int a) {
       :
 }
}

class Sub2 {
 public void method2(long a) {
       :
```

```
 }
}


// class for measurement
class Result {
 public static final int RMI_OVERHEAD = 20;
 public static final int CLASS_Main = 0;
 public static final int CLASS_Sub1 = 1;
 public static final int CLASS_Sub2 = 2;
 public static final int NUM_CLASSES = 3;
 // amount of communication
 private static int T[][] = new int[NUM_CLASSES][NUM_CLASSES];
 // number of method invocation
 private static int C[][] = new int[NUM_CLASSES][NUM_CLASSES];
 // time spent by each class
 private static long Q[] = new long[NUM_CLASSES];
 // the class that currently executing method belongs to
 private static int CurrentClass = CLASS_Main;
 // the time when currently executing method was invoked
 private static long InvokedTime = System.currentTimeMillis();
 public static void AddData(int from, int to, int size) {
  T[from][to] += size;
  C[from][to] ++;
  long CurrentTime = System.currentTimeMillis();
  Q[from] = CurrentTime - InvokedTime;
  InvokedTime = CurrentTime;
 }
}
```

In the above example program, we assume that the overhead of RMI is 20 bytes to calculate the amount of communication. A constant is defined for each class. The exchanged data size and the number of method invocations are recorded in two dimensional arrays T and C, respectively. Those two dimensional arrays are reffered by a pair of a caller class and a callee class. The time spent by each class is recorded in array Q.

4.4 Formulation of Module Assignment Problem

We have formulated our module assignment problem as follows.

input:

- a set of modules $M = \{m_1, ..., m_n\}$
- the memory size of each module $S : M \mapsto \mathbf{N}$
- available memory on the client side $m_c \in \mathbf{N}$
- modules explicitly assigned to the server side or client side $M_S, M_C \subset M$
- the amount of communication between modules $T : M \times M \mapsto \mathbf{N}$
- the number of method invocations between modules $C : M \times M \mapsto \mathbf{N}$
- the time spent by each module $Q : M \mapsto \mathbf{N}$

- the weights for the optimization parameters $K_1, K_2, K_3 \in \mathbf{N}$

output:

- a pair of assignments M_s and M_c that satisfy the following restriction and minimize the value of the objective function (here, M_s and M_c denote the server side modules and client side modules, respectively).

restriction:

- $M_s \cup M_c = M, M_s \cap M_c = \emptyset$
- $\sum_{m \in M_1} S(m) \leq m_c$

objective function:

$$F(M_s, M_c) = K_1 \sum_{m_1 \in M_s, m_2 \in M_c} T(m_1, m_2) + K_2 \sum_{m_1 \in M_s, m_2 \in M_c} C(m_1, m_2) + K_3 \sum_{m \in M_c} Q(m)$$

Here, we assume that $T(m_1, m_2)$ denotes the sum of the ammounts of communication from m_1 to m_2 and that for the reverse direction, and that the weight K_1 is given for the sum. If we prefer to give different weights for the both directions, we can do so. $C(m_1, m_2)$ also denotes the sum of the numbers of method invocations from m_1 to m_2 and those from m_2 to m_1. Here, K_1, K_2 and K_3 denote the weights for the amount of communication, the number of method invocations and the CPU time spent in the client side, respectively. We can optimize the amount of communication, communication delay and power consumption by adjusting the values of K_1, K_2 and K_3.

Here, we discuss about the computational complexity for this module assignment problem. Since a graph partitioning problem dividing a given set of vertexes $V = v_1, ..., v_{2n}$ of $G = (V, E)$ into two sets V_1 and $V_2 (|V_1| = |V_2| = n, V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset)$ by minimizing the cut $W(V_1, V_2) = \sum_{v_1 \in V_1, v_2 \in V_2, (v_1, v_2) \in E} 1$ is known as a NP-hard problem [56]. Here, by reducing this graph partitioning problem to our module assignment problem polynomially, we prove that our module assignment problem is also NP-hard.

proof: Any graph partitioning problem can be reduced to the module assignment problem as follows.

- $M = V \cup \{v_c\}$
- $\forall v \in M : S(v) = 1$
- $m_c = |V|/2 + 1$
- $\{v_c\} \in M_c$
- $T(v_1, v_2) = \begin{cases} 1 & ((v_1, v_2) \in E) \\ |E| + 1 & (v_1 = v_c \text{ or } v_2 = v_c) \\ 0 & (\text{otherwise}) \end{cases}$

- $K_1 = 1, K_2 = 0, K_3 = 0$, and all the other parameters are set to 0 or $\emptyset$.

For the given cost function $W(V_1, V_2)$ of the graph partitioning problem, we define the objective function of our module assignment problem as $F(M_s, M_c) = |M_s| \times (|E| + 1) + W(V_1, V_2)$ where $M_s = V_1$ and $M_c = V_2 \cup \{v_c\}$ hold. For any division of $G = (V, E)$, the cost W holds $W(V_1, V_2) \leq |E|$. Therefore, for any two divisions $\{M_{s1}, M_{c1}\}$ and $\{M_{s2}, M_{c2}\}$, $|M_{s1}| < |M_{s2}| \Rightarrow F(M_{s1}, M_{c1}) < F(M_{s2}, M_{c2})$ holds. So the optimal result M_s of our module assignment problem always minimizes the size of $|M_s|$. Also, the number of elements in $|M_s|$ holds $|M_s| = |M| - m_c = |V|/2$ because at most $|V|/2$ vertexes can be assigend to M_c. In this case, since the value of the objective function holds $F(M_s, M_c) = (|V|/2) \times (|E| + 1) + W(M_s, M_c - \{v_c\})$, the optimized division must minimize $W(M_s, M_c - \{v_c\})$. So we can get the optimized division of $G = (V, E)$ from the optimized assignment. Since this reduction can be done in polynomial time, our module assignment problem is proved as NP-hard. □

4.5 Optimizing the Assignment of Modules

Since this module assignment problem is NP-hard, we cannot the optimized result in practical time. So we use a heuristic algorithm to get approximate results. Here, we use a SA (Simulated Annealing) based method.

In SA based methods, candidate results are repeatedly improved in order to obtain the optimized result. A neighbor of the current candidate is selected as the new candidate randomly and if the new candidate is better than the current one, the current candidate is replaced by the new candidate. Also even if the new candidate is worse than the current one, the replacement occurs in some probability. The probability starts with a large value and is decreased mildly. It is known that by making the decrement very mild and trying enough times, relatively good results can be obtained.

4.5.1 SA Based Assignment Algorithm

In our method, we construct the optimization algorithm as follows.

1. set the initial temperature T_0 and give the initial candidate.
2. repeat the following processes for the specified times (hereafter this repetition number is called as the *loop number*).
   (a) generate a new candidate D' from the current candidate D.
   (b) calculate the difference between the values of the objective functions for the current candidate $F(D)$ and the new candidate $F(D')$ as $\Delta C = F(D') - F(D)$.
   (c) if the value of the objective function is smaller than all the candidates already checked, record the candidate D' and its minimum value.
   (d) if $\Delta C < 0$, replace the current candidate by the new candidate D'.

(e) otherwise, replace the current candidate by the new one in probability $\exp(-\Delta C/T)$.

3. decrease the temperature T as follows and continue from (2).
   $T_{k+1} = k\ T_k$ (k : *cooling coefficient*)

4. if the temperature T comes to T_{fin}, output the best candidate as the result and end this process.

Here, we have tried the following two methods for giving the initial candidates.

- (SA1): The modules are assigned to the client side in order of the time that the modules are invoked until the memory space is exhausted. All the left modules that cannot be assigned to the client side are assigned to the server side.
- (SA2): The modules are assigned to the client side in order of the amount of communication to the modules explicitly assigned to the client side.

Also the candidates are generated as follows.

- A module is selected.
- If the module has been assigned to the client side, the module is moved to the server side.
- If the module has been assigned to the server side, the module is moved to the client side where if the new assignment does not satisfy the memory restriction, the movement is cancelled and another module is selected as a new candidate.
- The above processes are carried out repeatedly.

4.6 Example Applications

We have implemented and evaluated our method by applying it to some example applications.

4.6.1 Ex1: Randomly Generated Modules

At first, we have applied our technique to some randomly generated problems.

- available memory on client side: 120.0KB
- number of modules
  - 30, 50

Table 4.1: Amount of communication between server and clients

(1) 30 modules

# of loops	10	50	100	150
SA1	265KB (0.06s)	136KB (0.28s)	136KB (0.57s)	98KB (0.85s)
SA2	170KB (0.06s)	130KB (0.29s)	104KB (0.56s)	104KB (0.85s)
brute force	98KB(200.54s)			

(2) 50 modules

# of loops	10	100	1000	2000
SA1	339KB (0.14s)	278KB (1.41s)	240KB (14.14s)	213KB (28.29s)
SA2	276KB (0.14s)	276KB (1.42s)	253KB (14.13s)	213KB (28.31s)
brute force	—(—)			

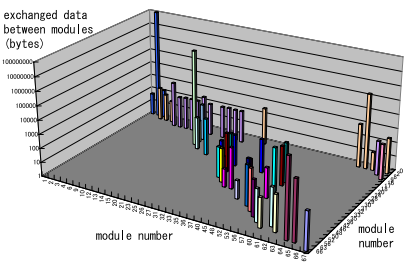
- size of modules
  - randomly generated at most 30.0KB.
- initial assignment
  - following two assignments (SA1) and (SA2)
- amount of communication
  - randomly generated

The results are shown in Table 4.1. Each table shows the amount of communication between the server and client where the numbers shown in the parentheses are time spent for calculation.

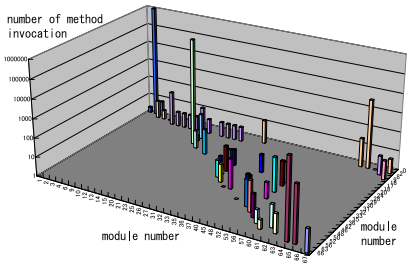
From the above results, we can say that by using SA based method we can obtain enough good results as an approximation for large sized problems that cannot be solved by the round robin method. And it is shown that enough large counts of loops are needed to derive good results close to the optimal ones. (SA2) can derive rather good results if the counts of loops are small, however, almost the same results can be derived by (SA1) and (SA2) if the counts of loops are large.

4.6.2 Ex2: an Existing Application

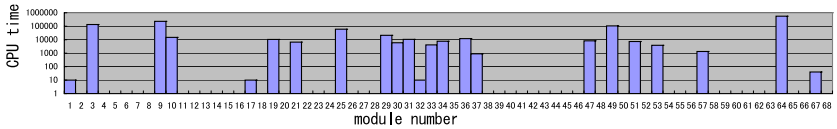
We have chosen an existing Java application for editing pictures. This application consists of 68 classes where the average size of classes is about 5 KB. At the first, we have collected our statistical information for this application. The



(1) Exchanged data between modules



(2) Number of method invocations between modules



(3) CPU time spent by modules

Figure 4.2: Interaction among modules and CPU time spent by modules in an example application

results are shown in Fig.4.2. Each graph shows (1) the amount of communication and (2) the number of method invocation between each pair of two classes (the pairs that never communicate with each other are omitted in these graphs) and (3) CPU time spent by each module.

Then, we have applied our module assignment algorithm to the application based on the above information. Here we have decided the parameters of SA as follows from the results of some examinations.

- initial temperature : $T_0 = 400.0$
- final temperature : $T_{fin} = 1.0$
- cooling coefficient : $k = 0.95$

To evaluate the usefulness of our technique, we have evaluated in some conditions by changing the optimization parameters K_1, K_2, K_3 and available memory on the client. The results are shown in Table 4.2. The first column shows the result of optimization for the amount of communication where only parameter K_1 is used while K_2 and K_3 are ignored. In this case, the amount of communication is significantly decreased by increasing the client's memory size. The second column shows the result of optimization for the number of method invocations in the same way. In this case, the number of method invocation between the client side modules and the sever side modules are decreased. The last column shows the result of optimization of power consumption. In this case, the increment of available memory causes the decrement of the CPU time spent in the client side.

Table 4.2: Result of division for the example application

memory of client (KB)	amount of comm. (bytes)	number of method invocation	CPU time of client (ms)
		$K_1 = 1,\ K_2 = 0,\ K_3 = 0$	
45	234014	4162	134199
50	18566	873	687890
55	2870	104	709080
60	2242	73	709080
		$K_1 = 0,\ K_2 = 1,\ K_3 = 0$	
45	234014	4162	134199
50	18958	908	687900
55	3594	136	755907
60	3462	108	691717
		$K_1 = 0,\ K_2 = 0,\ K_3 = 1$	
45	238198	4297	363280
50	236322	4236	378151
55	236090	4230	378151
60	235294	4218	378151

From these results, we can say our technique is useful for designing practical applications running on mobile terminals.

4.7 Conclusion

In this chapter, we have developed a tool for gathering statistical information of Java programs and proposed a module assignment technique where the amount of communication between the server side and client side, elapsed time or power consumption on handheld devices are minimized. We also have applied our technique to some examples and obtained useful results in reasonable time.

As our future work, we are planning to apply our technique to various applications. We also would like to find more effective approximation algorithms.

Chapter 5

An Efficient Deadlock Detection Method Using Symmetries for Distributed Applications

5.1 Introduction

According to the progress of high-speed networks in recent years, many distributed cooperative systems such as network meeting, remote lecturing and distributed multimedia authoring have been developed. In such distributed cooperative systems, the number of participants is often changed, and various constraints are added depending on the number of participants and network environment. However, if given constraints are inconsistent with each other, there may not exist executable behavior satisfying all the constraints, i.e., the given system may enter a deadlock state. In order to develop high reliable distributed systems, in this paper, we propose a model for specifying such a distributed cooperative system with unspecific number of participants and an efficient reachability analysis method for detecting the deadlocks in the model.

To specify such distributed cooperative systems simply and hierarchically, we use a constraint-oriented style, which is close to the description styles in [57, 7]. Such a constraint-oriented style is also used in the formal specification language LOTOS [5]. In the proposed constraint-oriented style, we describe a specification of a distributed cooperative system (concurrent system) as (A) a set of coloured Petri-nets and (B) synchronization among them. Each Petri-net describes either (A-1) each participant's behavior or (A-2) the temporal ordering of multiple participants' actions and/or constraints among those participants. Here, we call the above (A-1) and (A-2) descriptions as a *behavior net* and a *constraint net*, respectively. In a behavior net, each participant's

behavior is specified independently of other participants' behavior. Many and unspecific participants' behavior can be specified as a coloured Petri-net with multiple coloured tokens if those participants' behavior is essentially the same. In constraint nets, mutual exclusion among behavior nets and/or constraints as the total system can be specified. In the description of synchronization in the above (B), by letting an action in a behavior net and the same action in a constraint net be executed simultaneously, we can make the temporal ordering of the actions in behavior nets satisfy all the given constraints. Moreover, we can specify not only *one-to-one* synchronization but also *n-to-k* synchronization where arbitrary k processes in given n processes satisfying the constraints can synchronize with each other.

If we use this model to describe the specification of a total system, by changing the constraint nets, we can easily modify the total behavior of the system. However, the system may have the possibility to reach a deadlock state if we specify inconsistent constraints simultaneously. Since general reachability analysis techniques for coloured Petri-nets can detect such deadlocks if the number of states is finite, we impose a restriction on our model that limits the number of reachable states to finite. But if we use such techniques simply the cost for the verification still becomes large and the state explosion problem may occur.

In order to reduce the verification costs, Ref. [13] propose techniques to merge equivalent states into one and make a reduced size's reachability graph called *OS graph* [14] from the original reachability graph. Ref. [15, 16] propose another kind of efficient reachability analysis techniques using *symbolic reachability graph*. Ref. [17, 18] use invariants for reducing the verification costs. Ref. [19, 20] use stochastic Petri-nets, and Ref. [21] uses compositional high level Petri-nets for efficient reachability analysis.

There have not been proposed general techniques for finding equivalence relation between two reachable states mechanically from given specifications. However, in distributed cooperative systems, there are a lot of cases that multiple participants carry out essentially the same behavior and they do not cause different results for reachability analysis. For example, in a simple network meeting system where only one of multiple participants can be a speaker at each moment, the number of the participants does not affect the reachability analysis of the system specification, since the behavior of those participants can be regarded as the same, i.e., the specification has symmetries.

Here, we propose an efficient reachability analysis method where equivalence relation between two reachable states is found mechanically from a given specification by using the information about the symmetries. Moreover, we can reduce the number of reachable states, by some reduction rules based on the symmetries applied for CPN before reachablity analysis. We have also developed a verification tool and shown the usefulness of the method using some examples of network meeting systems.

The following Sect. 5.2 explains our constraint-oriented model and the details of coloured Petri-nets. Sect. 5.3 describes the reachability analysis method using symmetries. The experimental results are also given. Sect. 5.4 concludes this chapter.

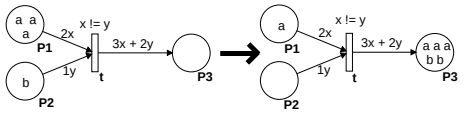


Figure 5.1: Firing of transition in CPN

5.2 Specification of Distributed Cooperative Systems in Proposed Model

5.2.1 Petri Net and Coloured Petri Net

A Petri-net (PN in short) is a weighted directed graph that consists of two types of nodes, *places* and *transitions*. Each directed edge a between a place and a transition is called an *arc* where an integer called *weight* (denoted as $W(a)$) is associated. Each place may have *tokens*, and an assignment of tokens to places is called a *marking*. A marking m assigns $m(p)$ tokens to place p. A transition t may fire *iff* each its input place p of t connected by an arc a has $W(a)$ tokens. If t fires, $W(a)$ tokens are taken from each input place p, and $W(a')$ tokens are added to each output place p' of t connected by an arc a'.

A coloured Petri-net (CPN in short) is a high level Petri net[58] where each token has a value. The values are distinguished by types called *colours*. The colours are, for example, integers, real numbers and characters. A colour is associated with each token of CPN, The weight $W(a)$ of an arc a in CPN is a multi-set of *binding variables*. And the assignment of tokens $m(p)$ represented by a marking m to a place p is also a multi-set of tokens. Here, a multi-set is a set that may include more than one identical element. For each binding variable, a colour is associated and a token with the same colour can be assigned to the binding variable. Hereafter, an assignment of a multi-set of tokens to a multi-set of binding variables is simply called a *binding*. For each transition t, a boolean function of binding variables that appear in the weights of the incoming arcs of t is associated and called a *guard*.

A transition t may fire *iff* (i) each input place p of t connected by an arc a has a multi-set of tokens that can be assigned to $W(a)$ and (ii) the value of the guard of t on this binding is true. If t fires, the multi-set of tokens are taken from each input place p of t, and the multi-set of tokens $W(a')$ that are determined by each binding to $W(a)$ are added to each output place p' of t connected by an arc a'. Note that the binding variables in the weight $W(a')$ of each outgoing arc a' must appear in at least one of the weights of the incoming arcs of t.

Fig. 5.1 shows an example. Suppose that x and y are binding variables and a and b are values of the same colours as x and y, respectively. $kx + hy$ represents a multi-set of x and y where the numbers of x and y are k and h, respectively. In this case, since a binding $[x = a, y = b]$ satisfies the guard "$x! = y$" $(x \neq y)$ of t true, t can fire on this binding. If t fires, tokens a and b are taken from places $P1$ and $P2$ respectively, and the multi-set of tokens $3a + 2b$ is added to place

$P3$.

5.2.2 CPN Specification in Constraint-Oriented Style

CPN is a suitable model to describe specifications of distributed cooperative systems. This is mainly because the same behavior of participants (*e.g.* the behavior of students in remote lecturing), can be described as a single net where each coloured token represents an individual participant. However, for the efficient design of distributed cooperative systems where the temporal ordering of actions of participants should be specified, we introduce the concept of constraint-oriented style into CPN.

A specification of a distributed cooperative system written in CPN in constraint-oriented style consists of a set of *behavior nets*, a set of *constraint nets* and *synchronization*. Each behavior net includes tokens that represent participants. Each constraint net specifies the temporal ordering of transitions in behavior nets. Each synchronization associates each transition of a constraint net with one of the transitions in behavior nets. For each synchronization, a boolean function of binding variables of the two transitions can be specified as a guard.

The CPN specification of n participants' system (for simplicity, participants are denoted as integers $1..n$ hereafter) with k different types of behavior ($k \leq n$) consists of k behavior nets, h constraint nets ($h \geq 0$) and l synchronization ($l \geq 0$). We assume that the specification must be described in the following style.

- Hereafter, each behavior net is denoted as BN_i ($1 \leq i \leq k$). BN_i contains a set of tokens t_i of the colour "person", an enumerative type with elements $1..n$. Here (a) $\cup_{1 \leq j \leq k} t_j = \{1..n\}$ and $t_j \cap t_{j'} = \emptyset$ must hold, (b) the set of existing tokens in BN_j is always equals to t_j. These indicate that each token in BN_i represents a participant. Moreover, each transition of BN_i has guard "true".

- Hereafter, each constraint net is denoted as CN_j ($1 \leq j \leq h$). The colours allowed to use in constraint nets are "person", the limited number of integers and "e", which represents an empty colour (tokens that have no value). Here, considering the need for specifying "arbitrary number of participants", we introduce a new colour "variant" for the binding variables of constraint nets. A binding variable of this colour is a special variable where any multi-set of tokens can be assigned. Each CN_j must be bounded. Moreover, each transition of CN_j has guard "true".

- Hereafter, each synchronization is denoted as $sync_x$ ($1 \leq x \leq l$). For each $sync_x$ that associates t_v of CN_j with t_u of BN_i, a boolean function of binding variables used in the weights of the incoming arcs of t_u or t_v can be specified as a guard. Here, we introduce a special function $\#(v)$ of binding variable v of colour "variant". This function returns the number of tokens that are assigned to v.

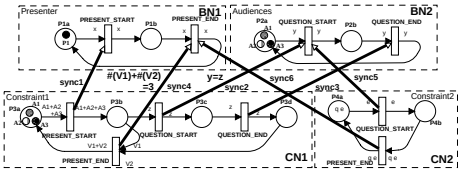


Figure 5.2: CPN specification of simple network meeting system in constraint oriented style

	Behavior Net		Constraint Net		Guard
	Net	Transition	Net	Transition	
$sync_1$	BN_1	PRESENT_START	CN_1	PRESENT_START	true
$sync_2$	BN_2	QUESTION_START	CN_1	QUESTION_START	$y = z$
$sync_3$	BN_2	QUESTION_END	CN_1	QUESTION_END	true
$sync_4$	BN_1	PRESENT_END	CN_1	PRESENT_END	$\#(V1)+$ $\#(V2) = 3$
$sync_5$	BN_2	QUESTION_START	CN_2	QUESTION_START	true
$sync_6$	BN_1	PRESENT_END	CN_2	PRESENT_END	true

Table 5.1: Description of synchronization

5.2.3 Example Specification

Fig. 5.2 and Table 5.1 shows an example specification of a simple network meeting system written in CPN in this style, where one presenter and three audiences participate in the meeting. BN_1 and BN_2 are behavior nets that represent the behavior of the presenter and the audiences, respectively. CN_1 and CN_2 are constraint nets. Six synchronization with guards $sync_1$, .. and $sync_6$ are specified in Table 5.1 and represented as arrows in Fig. 5.2.

In BN_1, two actions "PRESENT_START" (start presentation) and "PRESENT_END" (end presentation) are specified for the presenter. In BN_2, two actions "QUESTION_START" (start a question) and "QUESTION_END" (end a question) are specified for the three audiences. x and y are binding variables of colour "person". For these behavior nets, the two constraint nets and synchronization specify the temporal ordering of their actions. They represent the following constraints.

(i) Questions must not be started before the presentation is started.

(ii) Each audience may ask a question only once before the presentation is stopped.

(iii) q questions must be asked before the presentation is stopped.

The constraint (i) is represented by CN_1 and two synchronization $sync_1$ and $sync_2$. In $sync_1$, two transitions "PRESENT_START" of BN_1 and CN_1 synchronize, and in $sync_2$, two transitions "QUESTION_START" of BN_2 and CN_1

synchronize. According to CN_1, "QUESTION_START" cannot be executed without the presence of tokens of colour "person" in place P_{3b} of CN_1. Those tokens are produced by the firing of "PRESENT_START". Therefore, "QUESTION_START" cannot fire before the firing of "PRESENT_START".

The constraint (ii) is represented by CN_1 and three synchronization $sync_2$, $sync_3$ and $sync_4$. In order to execute "PRESENT_END", the value of the guard of $sync_4$ must be true. The guard includes two binding variables $V1$ and $V2$ of colour "variant" where any multi-set of tokens can be assigned. Here, each token in place P_{3b} represents an audience who has not asked a question yet. On the other hand, each token in place P_{3d} represents an audience who has already asked a question by firing of "QUESTION_START" and "QUESTION_END". These tokens are removed if "PRESENT_END" fires because the guard of $sync_4$ "$\#(V1) + \#(V2) = 3$" needs all the three tokens in P_{3b} and P_{3d} to be assigned to $V1$ and $V2$. This means that each audience who has already asked a question is never allowed to ask a question once again before the firing of "PRESENT_END".

The constraint (iii) is represented by CN_2 and synchronization $sync_5$ and $sync_6$. By each firing of "QUESTION_START" in CN_2, token "e" is produced in place P_{4b}. The tokens in place P_{4b} represent the number of questions that have been asked after the presentation has been started. In order to end the presentation by the firing of "PRESENT_END", there must be q tokens of "e" in P_{4b}.

In Sect. 5.3, we explain how a specification described in this model is transformed into a pure CPN where the reachability analysis can be performed. The state space reduction by using the symmetries of CPN is also explained.

5.3 Reachability Analysis

In our model, since we specify a system specification as a set of behavior nets, constraint nets and synchronization among them, the total system may include deadlock states due to some constraints inconsistent with each other. Generally we can check whether a system includes a deadlock state or not by constructing the reachability graph of the system. Here, we adopt a policy to check the deadlock-free property or liveness property of a given system by constructing an occurrence graph[14], which is known as a kind of abstraction of the reachability graph. To construct an occurrence graph, we transform a given specification (consisting of several CPNs and synchronization among them) into the equivalent single CPN. Then for the derived CPN, we construct the corresponding occurrence graph so that the number of nodes in the graph is reduced using symmetries.

5.3.1 Derivation of Single CPN from CPN Specification in Constraint-Oriented Style

The proposed transformation technique is as follows.

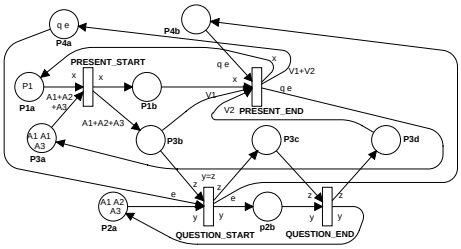


Figure 5.3: Specification of total system derived from Fig.2

(i) Each pair of transitions in a constraint net and a behavior net that are specified to synchronize is merged into a single transition. If a guard is specified in the synchronization, the guard should be added as the guard function of the transition (Fig.5.3).

(ii) Each transition with variant type variables is replaced by a set of transitions without those variables (Fig.5.4). This replacement is carried out as follows.

   (a) Enumerate the possible bindings for the variant type variables, where each binding satisfies the specified guard functions.

   (b) For each possible binding, generate a new transition with appropriate tokens and variables for the binding.

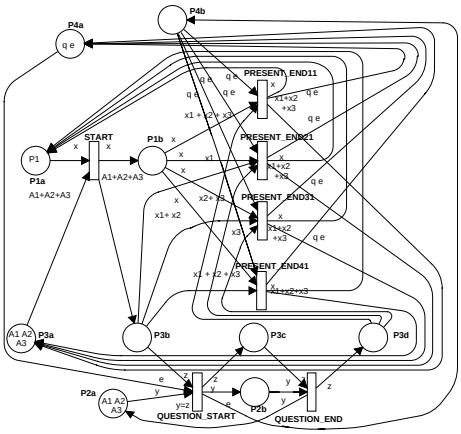


Figure 5.4: Transformed specification of total system

To make sure that (a) is always possible, we suppose that the number of possible bindings for each variant type variable is finite. Fig.5.4 shows the

CPN derived from the specification in Fig.5.2. As an example, the transitions PRESENT_END1 ,.., PRESENT_END4 in Fig.5.4 are derived by applying the step (ii). These transitions are generated from the combination of transitions PRESENT_END of BN_1 and CN_1 in Fig.5.2 where the variant type variables of PRESENT_END of CN_1 are replaced by the variables $x1$,..,$x3$. From the guard "$\#(V1) + \#(V2) = 3$", the possible bindings of $V1$ and $V2$ are the following four patterns: $\{(\#(V1), \#(V2))\} = \{(0,1),(1,2),(2,1),(3,0)\}$. Transitions PRESENT_END1 ,.., PRESENT_END4 correspond to those patterns, respectively. For example, variable $V1$ in PRESENT_END2 is replaced by variable $x3$ and variable $V2$ is replaced by $x1 + x2$ as a result of the binding. In this way, a set of concurrent finite CPNs including variant type variables are transformed into a single finite CPN.

5.3.2 Reachability Analysis with OS Graph

Let us suppose that a network conferencing system consisting of a chairman, k presenters and n audiences is modeled as a set of $k + n + 1$ CPNs in general. In this case, if we strictly distinguish the $k + n + 1$ types of tokens from each other, a quite large *occurrence graph* (reachability graph) will be derived from the given specification.

However, if we allow any audience to ask a question in a given system specification, we can regard that the global state transition of the system is the same whoever asks a question. In such a case, we need not distinguish each individual audience from the others in constructing the occurrence graph. Instead, we can construct the corresponding *OS graph* (occurrence graph with symmetries) where the number of possible states is reduced using symmetries, and carry out the reachability analysis efficiently. The OS graph is a reachability graph where each node corresponds to an equivalent class of states classified depending on the given symmetries. Since in the OS graph, a node can represent multiple states, the size of reachability graph can be reduced and we can efficiently carry out the reachability analysis.

The occurrence graph is defined as a pair (M, A). $M = \{m_1, .., m_n\}$ is the set of all possible markings and $A \subseteq M \times M$ represents a state transition. Here $(m_i, m_j) \in A$ holds if and only if marking m_j is reachable by the firing of one transition from marking m_i. The OS graph is a graph where the number of states is reduced by unifying "equivalent" states into a single state. For a given equivalence relation E, the OS graph is represented as a pair $(\mathbf{M}, \mathbf{A})$, where $\mathbf{M}$ is a family of sets of equivalent markings and $\mathbf{A}$ is a set of state transitions on $\mathbf{M}$. Hereafter, $[m]$ denotes the set of markings that are equivalent with m. $\mathbf{M} = \{[m_1], .., [m_m]\}$ is a family of sets of equivalent markings calculated by classifying M by E. $\mathbf{A}$ is defined such that

$$\forall m_i, m_j : (m_i, m_j) \in A \rightarrow ([m_i], [m_j]) \in \mathbf{A} \tag{5.1}$$

Here, if the following condition holds, it is known that $(\mathbf{M}, \mathbf{A})$ is a deadlock free

OS graph if and only if the original occurrence graph is deadlock free[14].

$$([m_i],[m_j]) \in \mathbf{A}$$
$$\rightarrow \forall m'_i \in [m_i] : \exists m'_j \in [m_j] : (m'_i, m'_j) \in A \tag{5.2}$$

Intuitive Proof : From the condition (5.1), if m_j is reachable from m_i in the graph (M, A), $[m_j]$ is also reachable from $[m_i]$ in the graph $(\mathbf{M}, \mathbf{A})$. From the condition (5.2), if $[m_j]$ is reachable from $[m_i]$ in the graph $(\mathbf{M}, \mathbf{A})$, for any marking $m'_i \in [m_i]$, there exists a marking $m'_j \in [m_j]$ where m'_j is reachable from m'_i. □

5.3.3 Sufficient Condition of Symmetries

There are no general methods for finding an equivalence relation satisfying the above (1) and (2) automatically. Here, we would like to give a sufficient condition for finding such an equivalence relation automatically. Here, we consider a set of *symmetric* tokens. The word "a set S of *symmetric* tokens" means that for the initial marking, weight and guard in a given specification, either the following (i) or (ii) holds;

(i) any colour of tokens contained in S is not specified.

(ii) all the colours of tokens contained in S are specified.

For example, for the specification in Fig.5.2, let S denote the set of tokens $\{A1, A2, A3\}$. In Fig.5.2, the initial marking contains either all elements in S or no elements in S. For each weight and guard, the same property holds. So, $S = \{A1, A2, A3\}$ can be regarded as a set of *symmetric* tokens.

For a given CPN specification in constraint-oriented style, by checking all initial marking, weights and guards step-by-step, we can extract the sets of tokens appeared in the initial marking, weights and guards. For each set of tokens, we can mechanically check whether either the above (i) or (ii) holds. Then, we can mechanically find a set of symmetric tokens for a given CPN specification if there exists such a set.

Hereafter, we will propose a method to generate an equivalence relation E satisfying the above conditions (1) and (2) mechanically from the derived set of symmetric tokens. Note that for a set S of symmetric tokens such as $S = \{A1, A2, A3\}$, let p_1 and p_2 denote two different lists containing all the tokens in S (for example, $p_1 = [A1, A2, A3]$ and $p_2 = [A2, A1, A3]$). Then, we say that p_1 is a *permutation* of p_2, vise versa.

Here, for a given set S of symmetric tokens, let us consider a relation E between two reachable markings m_i and m'_i where m'_i is obtained from m_i by replacing S in m_i with a *permutation* of S and vise versa. For example, for the set of symmetric tokens $S = \{A1, A2, A3\}$ in Fig. 5.2, two reachable markings $m_i = (\emptyset, P1, A2 + A3, A1, A2 + A3, A1, \emptyset, 2e, e)$ and $m'_i = (\emptyset, P1, A1 + A3, A2, A1 + A3, A2, \emptyset, 2e, e)$ satisfy the relation E since m'_i is obtained from m_i by replacing $[A1, A2, A3]$ in m_i with one of its permutations $[A2, A1, A3]$ and vise versa. In our method, this relation E is an equivalence relation.

Intuitively this is clear because all the symmetric tokens move in the same way and make no difference between two markings m_i and m'_i where a token proceeds in m_i and one of its symmetric tokens proceeds in m'_i. The sketch of proof is given as follows.

Assume that two reachable markings m_i and m'_i satisfy the relation E based on the set S of symmetric tokens, where m'_i is obtained by replacing S in m_i with one of its permutations (this permutation is denoted as p hereafter). Also assume a marking m_j reachable from m_i by the firing of a transition T on a binding B. Here, since the tokens of S are symmetric, those tokens in binding B can be replaced with the permutation p, and this replacement makes a new binding B'. Obviously, since m'_i is obtained by the same permutation p, there exists a state transition from m'_i to a marking m'_j by the firing of the same transition T on binding B'. Then we can say that m_j and m'_j satisfy the relation E by the same permutation p, since permutation p replaces the tokens of S in m_j that are in each input (or output) place of T with the ones in the same input (or output) place in m'_j. Consequently, for every marking $m'_i \in [m_i]$, there exist a marking $m'_j \in [m_j]$ and a state transition from m'_i to m'_j $((m'_i, m'_j) \in A)$. Therefore, the sufficient condition for equivalence relation in Sect 5.3.2 holds.

Now we can say that since the two markings $m_i = (\emptyset, P1, A2 + A3, A1, A2 + A3, A1, \emptyset, 2e, e)$ and $m'_i = (\emptyset, P1, A1 + A3, A2, A1 + A3, A2, \emptyset, 2e, e)$ in Fig.5.2 satisfy the above equivalence relation E, they are merged in the corresponding OS graph.

5.3.4 Further Reduction of CPN and Omitting Colour Information

In our method, we use the following techniques for reducing the size of CPNs so that the reachability analysis can be efficiently carried out.

- A consecutive sequence of transitions is transformed into one transition. The transitions that will obviously fire sequentially and are not specified to synchronize with other CPNs are replaced by one transition.
- If there are consecutive transitions t_1, t_2 in a net and consecutive transition t'_1, t'_2 in another net, and if t_1 and t_2 synchronize with t'_1 and t'_2, respectively, then the two synchronization relation can be merged into one synchronization.
- The synchronization guard checking the colours of tokens can be deleted if the guard is always true in checking the condition of firing for any reachable marking and there is no more synchronization that is specified with guard in a given specification.
- The colour information of a set of tokens can be omitted, if the colours of these tokens are never checked anywhere. To do so, we introduce a new token that represents all of the tokens belonging to the set, and replace the existing tokens by the new token.

Table 5.2: Experimental result

	(i) Occurrence Graph			(ii) OS Graph			(iii) OS Graph (Omit Colour Info.)		
Audience	# Nodes	# Arcs	Time (sec.)	# Nodes	# Arcs	Time (sec.)	# Nodes	# Arcs	Time (sec.)
3	28	61	1	11	14	1	11	14	1
4	66	177	1	11	14	1	11	14	1
5	132	451	1	11	14	1	11	14	1
6	234	1333	7	11	14	3	11	14	1
7	380	6063	212	11	14	586	11	14	1
15							11	14	3
16							11	14	11
17							11	14	17

For example, in Fig.5.2 the synchronization 2 and 3 are sequential and their guards are the same. So, each QUESTION_START and QUESTION_END can be combined into one transition T. As a result, a set of tokens $\{A1, A2, A3\}$ is obviously always placed on the input place of the transition T of BN2. And other tokens except tokens $A1$, $A2$ and $A3$ are never placed on the input place of transition T of CN1. So, if a token is on the place of CN1, this token must be also placed on the place of BN2 and the guard $y = z$ holds. And since no other synchronization between BN2 and CN1 is specified, this guard of synchronization can be deleted. As a result, since there is no specification that distinguishes tokens $A1$, $A2$ and $A3$, they can be regarded as the same token. So we can replaces them by a new token A for reachability analysis.

5.3.5 Reachability Analysis System

In order to evaluate our method, we have developed a system to derive a single CPN from a given specification consisting of behavior nets, constraint nets and synchronization. This system derives a single CPN by coupling the transitions of the behavior nets and constraint nets specified to synchronize for some specific cases. And then it reduces the size of CPN by picking up the symmetric tokens and by omitting the colour information needless to distinguish. Then we check the deadlock-free property with a general formal model checker for CPN.

5.3.6 Experimental Result

We have used a tool to design and simulate Petri-nets called Design/CPN[59] in our reachability analysis system. We have measured the time to examine the deadlock-free property of the example of Fig.5.2. For this example, we have checked the example as changing the number of audiences and the number of questions. Table 5.2 shows the results. In this table, the size of the reachability graph and the time consumed for the calculation are shown for the following three cases: (i) the case of generating occurrence graph without considering

symmetries; (ii) the case of making the OS graph by using symmetries; and (iii) the case of making the OS graph after omitting some colour information. In case (ii), the size of the graph is very small compared with case (i), while the consumed time may increase due to calculation of symmetries. In case (iii), the consumed time is substantially reduced, especially in large specifications.

5.4 Conclusion and Future Work

We have proposed a constraint-oriented model for developing distributed cooperative systems with symmetries. In our model, we describe the specification of a system by a set of coloured Petri-nets and synchronization among them. A specification of each node is described independently and the interactions among them are specified using constraints and synchronization among them.

We have also proposed a method for efficient reachability analysis for this model. In our method, the symmetries are automatically detected from a given specification and the reachability analysis is quickly carried out by making an OS graph using the symmetries. We have adopted this method for an example, and we can reduce the size of reachability graphs and the required time for reachability analysis.

To extend this model so that we can specify time constraints and to develop a method of efficient reachability analysis for such a model are our future work.

Chapter 6

Conclusion

In this paper, we have proposed a new language called LOTOS/M suitable for description and implementation of wireless mobile applications.

In LOTOS/M, we can describe dynamic establishment of multi-way synchronization channels among agents so that the agents which happen to meet in a common radio range can dynamically communicate by multi-way synchronization. Also, LOTOS/M can naturally handle the case that some of the combined agents are dynamically isolated (e.g., by leaving from a radio range).

We also have proposed a Java-based middleware for supporting interactions among agents in wireless ad hoc networks based on LOTOS/M. With our middleware, we can easily handle group communication such as multicast data distribution and mutual exclusion among multiple agents by the methods of the middleware for dynamically establishing multi-way synchronization channels. Also, we can treat the case that some agents in an agent group are isolated by leaving from the common radio range.

In our implementation method, since we manage membership information of each agent group as a binary tree based on LOTOS semantics, we could easily implement multi-way synchronization among agents by keeping the latest tree and collecting/processing request messages from agents along the tree based on the technique in [24]. Through some experiments, we have confirmed that our proposed technique is enough applicable to describe and implement wireless mobile applications.

Next, we have developed a tool for gathering statistical information of Java programs and proposed a module assignment technique where the amount of communication between the server side and client side, elapsed time or power consumption on handheld devices are minimized. We also have applied our technique to some examples and obtained useful results in reasonable time.

Lastly, we have proposed a constraint-oriented model for developing distributed cooperative systems with symmetries. In our model, we describe the specification of a system by a set of coloured Petri-nets and synchronization among them. A specification of each node is described independently and the interactions among them are specified using constraints and synchronization

among them.

We have also proposed a method for efficient reachability analysis for this model. In our method, the symmetries are automatically detected from a given specification and the reachability analysis is quickly carried out by making an OS graph using the symmetries. We have adopted this method for an example, and we can reduce the size of reachability graphs and the required time for reachability analysis.

Now we are studying about more efficient formal description language for modeling ad-hoc mobile applications based on LOTOS. In this language, each dynamically established multi-way synchronization channel and process can hold more information about its role in the application. So the special processes such as processes specifying constraints can be explicitly distinguished and handled in different ways from the other processes. We think we can describe distributed cooperative systems in ad-hoc environments by using constraint oriented style more directly in this language.

We are planning to evaluate if this language is useful and to implement a compiler. For this purpose, we will develop a simulator of mobile environments with ad-hoc networks and a language for describing movements of nodes such as persons and cars. In this simulator, we will specify not only the behavior of an application but also how how each node moves by using this language. Then the simulator will simulate the nodes' movements and specified network application cooperating with some ad-hoc network simulator. By using this simulator, we think we can evaluate languages, middleware and applications for mobile ad-hoc networks with behavior of them in nearer environments to practical environments.

Acknowledgement

This work could not be achieved without support of many individuals.

First, I would like to thank my supervisor Professor Teruo Higashino of Osaka University, for his continuous support, encouragement and guidance of the work.

I am very grateful to Professor Makoto Imase and Professor Hirotaka Nakano of Osaka University for their invaluable comments and useful suggestions concerning this thesis.

Many of the courses in Osaka University that I have taken during my graduate career have been helpful to write this thesis. I would like to acknowledge the guidance of Professors Toru Fujiwara, Toshinobu Kashiwabara, Toru Kikuno, Masaharu Imai, Masayuki Murata, Hideo Matsuda, Kenichi Taniguchi, Kenichi Hagihara, Katsurou Inoue, Toshimitsu Masuzawa, Hideo Miyahara, Shinichi Tamura, Akihiro Hashimoto and Nobuki Tokura.

I would like to express my thanks to Professor Minoru Ito of Nara Institute of Science and Technology, who has provided many valuable comments.

I would like to thank Professor Nobuo Funabiki of Okayama University for his valuable comments and discussions.

I am very grateful to Associate Professor Akio Nakata of Osaka University, Associate Professor Keiichi Yasumoto of Nara Institute of Science and Technology and Associate Professor Junji Kitamichi of Aizu University for their insightful and constructive comments.

I wish to thank Research Associate Toshiaki Yoshioka and Hirozumi Yamaguchi of Osaka University for their helpful comments.

At last, I would like to thank all the members of Higashino Laboratory of Osaka University for their helpful advice.

Bibliography

[1] Umedu T., Terashima Y., Yasumoto K., Nakata A., Higashino T. and Taniguchi K. : A Language for Describing Wireless Mobile Applications with Dynamic Establishment of Multi-way Synchronization Channels, *Proceedings of International Symposium of Formal Methods Europe(FME2002)*, pp.607-624 (Jul. 2002).

[2] Umedu T., Yasumoto K., Nakata A., Yamaguchi H. and Higashino T. : Middleware for Synchronous Group Communication in Wireless Ad Hoc Networks, *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN2002)*, pp. 48-53 (Nov. 2002).

[3] Umedu T., Urata S., Nakata A. and Higashino T. : Automatic Decomposition of Java Program for Implementation on Mobile Terminals, *Proceedings of 19th IEEE International Conference on Advanced Information Networking and Applications (AINA2005)* (Jun. 2005). (to appear)

[4] Umedu T., Yamaguchi H., Yasumoto K., Nakata A. and Higashino T. : Constraint-Oriented Model for Describing Distributed Cooperative Systems and Efficient Verification Using Symmetries, *International Journal of Computer and Information Science*, Vol. 3, No. 2, pp. 125-136 (Jun. 2002).

[5] ISO : Information Processing System, Open Systems Interconnection, LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO 8807 (1989).

[6] Bolognesi T. : Toward Constraint-Object-Oriented Development, *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, pp. 594 – 616 (2000).

[7] Vissers C. A., Scollo G. and Sinderen M. v. : Architecture and Specification Style in Formal Descriptions of Distributed Systems, *Proceedings 8th International Symposium on Protocol Specification, Testing, and Verification (PSTV-VIII)*, pp. 189–204 (1988).

[8] Milner R., Parrow J., Walker D. : A Calculus of Mobile Processes: Parts I & II, *Information and Computation No. 100*, pp. 1– 77 (1992).

[9] Fevrier, A., Najm, E., Leduc, G. and Leonard, L. : Compositional Specification of ODP Binding Objects, *Proceedings of 6th IFIP/ ICCC Conference* (1996).

[10] Najm E., Stefani J. B. and Fevrier A. : Towards a Mobile LOTOS, *Proceedings of 8th IFIP International Conference on Formal Description Techniques (FORTE'95)* (1995).

[11] Zhao J. : Slicing Concurrent Java Programs, *Proceedings of 7th International Workshop on Program Comprehension*, pp. 126–133 (1999).

[12] Zhao J. : Multithreaded Dependence Graphs for Concurrent Java Program, *Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pp. 13–23 (1999).

[13] Jorgensen J. B. and Kristensen L. M.: Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Colored Petri Nets and Occurrence Graphs with Symmetries, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 7, pp.714–722 (1999).

[14] Jensen K. : Coloured Petri Nets, *EATCS Monographs in Theoretical Computer Science* Vol. 2., Springer-Verlag (1997).

[15] Hameurlain N. and Sibertin-Blanc C. : Finite Symbolic Reachability Graphs for High-Level Petri Nets, *Proceedings of 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97 / ICSC '97)*, pp. 150–159 (1997).

[16] Cortadella J. : Combining Structural and Symbolic Methods for the Verification of Concurrent Systems, *Proceedings of International Conference on Application of Concurrency to System Design (CSD '98)*, pp. 152–157 (1998).

[17] Miyamoto T. and Kumagai S. : Calculating Place Capacity for Petri Nets Using Unfoldings, *Proceedings of International Conference on Application of Concurrency to System Design (CSD '98)*, pp. 143–151 (1998).

[18] Nakamura M., Kakuda Y. and Kikuno T. : Analyzing Non-determinism in Telecommunication Services Using P-invariant of Petri-Net Model, *Proceedings of INFOCOM '97*, pp.1253–1259 (1997).

[19] Capra L., Franceschinis G., Dutheillet C. and Ilie J. M. : Towards Performance Analysis with Partially Symmetrical SWN, *Proceedings of 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 148–155 (1998).

[20] Gaeta R. : Efficient Discrete-Event Simulation of Colored Petri Nets, *IEEE Transactions on Software Engineering*, Vol. 22, No. 9, pp. 629–639 (1996).

[21] Grahlmann B. and Fleischhack H. : Towards Compositional Verification of SDL Systems, *Proceedings of 31st Hawaii International Conference on System Sciences (HICSS'98)*, pp. 404–414 (1998).

[22] Hodes T. D., Katz R. H., Schreiber E. S. and Rowe L. : Composable Ad-hoc Mobile Services for Universal Interaction, *Proceedings of Mobile Computing and Networking(MOBICOM'97)* (1997).

[23] Johnson D. B., Maltz D. A., Hu Y. C. and Jetcheva J. G. : The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks, *IETF Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr04.txt* (2000).

[24] Yasumoto K., Higashino T. and Taniguchi K. : A compiler to implement LOTOS specifications in distributed environments, *Computer Networks*, Vol. 36, No.2-3, pp. 291–310 (2001).

[25] Tuok R., Logrippo L. : Formal Specification and Use Case Generation for a Mobile Telephony System, *Computer Networks*, Vol. 30, No. 11, pp. 1045-1063 (1998).

[26] Ando T., Takahashi K., Kato Y. and Shiratori N. : A Concurrent Calculus with Geographical Constraints, *IEICE Transactions on Fundamentals*, Vol. E81-A, No. 4, pp. 547–555 (1998).

[27] Sangiorgi D. : From π-calculus to Higher-Order π-calculus — and back, *Proceedings of Theory and Practice of Software Development (TAPSOFT'93)*, Lecture Notes in Computer Science Vol. 668, pp. 151 – 166 (1993).

[28] Groote J. F. : Transition System Specification with Negative Premises, *Theoretical Computer Science*, Vol.118, No.2, pp.263-299 (1993).

[29] The Official Bluetooth Website, *http://www.bluetooth.com*.

[30] Kortuem G. : Proem: A Peer-to-Peer Computing Platform for Mobile Adhoc Networks in [60].

[31] Migliardi M. : Mobile Interfaces to Metacomputing and Collaboration Systems in [60].

[32] Cugola G. and Nitto E. D. : Using a Publish/Subscribe Middleware to Support Mobile Computing in [60].

[33] Jung D. G., Paek K. J. and Kim T. Y. : Design of MOBILE MOM: Message Oriented Middleware Service for Mobile Computing, *Proceedings of 1999 International Workshops on Parallel Processing (ICPP-99)* (1999).

[34] Reinstorf T., Ruggaber R., Seitz J. and Zitterbart M. : A WAP-Based Session Layer Supporting Distributed Applications in Nomadic Environments, *Proceedings of 2001 IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pp.56–76 (2001).

[35] Sow D. M., Banavar G., Davis II J. S., Sussman J. and Rwebangira M. R. : Preparing the Edge of the Network for Pervasive Content Delivery in [60].

[36] Ebling M. R., Hunt G. D. H. and Lei H. : Issues for Context Services for Pervasive Computing in [60].

[37] Grace P. and Blair G. S. : Integrating Middleware Paradigms to Support a Mobile Sport News Application in [60].

[38] Jacobsen H. A. : Middleware Services for Selective and Location-based Information Dissemination in Mobile Wireless Networks in [60].

[39] Meier R., Killijian M. O., Cunningham R. and Cahill V. : Towards Proximity Group Communication in [60].

[40] Bellavista, P., Corradi, A. and Stefanelli, C. : Protection and Interoperability for Mobile Agents: A Secure and Open Programming Environment, *IEEE Transactions on Communication*, May 2000, pp. 961–972 (2000).

[41] Bellavista, P. and Stefanelli, C. : Mobile Agent Middleware for Mobile Computing, *IEEE Computer*, pp. 73–81 (2001).

[42] Lange D. B. and Chang D. T. : IBM Aglets Workbench - Programming Mobile Agents in Java, IBM Corp. White Paper, http://www.ibm.co.jp /trl/aglets (1996).

[43] Grosso W. : Java RMI, *O'Reilly & Associates, Inc.* (2002).

[44] HORB Open : http://www.horbopen.org/ (2001).

[45] Matjaz B. J., Ivan R., Marjan H., Alan P. S. and Simon N. : Java 2 Distributed Object Models Performance Analysis, Comparison and Optimization, *Proceedings of 7th International Conference on Parallel and Distributed Systems (ICPADS'00)*, pp. 239–246 (2000).

[46] Kalogeraki V., Melliar-Smith P. M. and Moser L. E. : Using Multiple Feedback Loops for Object Profiling, Scheduling and Migration in Soft Real-Time Distributed Object Systems, *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 291–300 (1999).

[47] Flores A. P., Nacul A., Silva L., Netto J., Pereira C. E. and Bacellar L. : Quantitative Evaluation of Distributed Object-Oriented Programming Environments for Real-Time Applications, *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 133–138 (1999).

[48] Kono K. and Masuda T. : Efficient RMI, Dynamic Specialization of Object Serialization, *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pp. 308–315 (2000).

[49] Kernighan B. W. and Lin S. : An Efficient Heuristic Procedure for Partitioning Graphs, *Bell Systems Technical Journal*, Vol.49, No.2, pp. 291–307 (1970).

[50] Krishnamurthy B. : An Improved Min-Cut Algorithm for Partitioning VLSI Networks, *IEEE Transactions on Computers*, Vol.33, No.5, pp. 438–446 (1984).

[51] Bui T. N. and Moon B. R. : Generic Algorithm and Graph Partitioning, *IEEE Transactions on Computers*, Vol.45, No.7, pp. 841–855 (1996).

[52] Johnson D. S., Aragin C., McGeoch L., and Schevon C. : Optimization by Simulated Annealing, An Experimental Evaluation, Part1, Graph Partitioning, *Operations Research*, Vol.37, pp. 865–892 (1987).

[53] Fiduccia C. M. and Mattheyses R. M. : A Linear-Time Heuristic for Improving Network Partitions, *Proceedings of 19th Design Automation Conference*, pp. 175–181 (1982).

[54] Shenoy P. and Radkov P. : Proxy-assisted Power-friendly Streaming to Mobile Devices, *Proceedings of the 2003 Multimedia Computing and Networking Conference (MMCN)*, pp. 177-191 (2003).

[55] Sebastien Vauclair : Extensible Java Profiler, http://ejp.sourceforge.net/

[56] Garey M. R. and Johnson D. S. : Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman (1979).

[57] Bolognesi T. : Toward Constraint-Object Oriented Development, *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, pp. 594 – 616 (2000).

[58] Jensen K. and Rozenberg G. (eds.) : High-level Petri Nets. Theory and Application, Springer-Verlag (1991).

[59] CPN group at the University of Aarhus, Denmark : Design/CPN, Ver. 4.0.4, *http://www.daimi.aau.dk/designCPN/*

[60] Advanced Topic Workshop Middleware for Mobile Computing, In association with IFIP/ACM Middleware 2001 Conference, URL: http://http://www.cs.arizona.edu/mmc/ (2001).