RoleEP: Role Based Evolutionary Programming for Cooperative Mobile Agent Applications

Naoyasu Ubayashi
Toshiba Corporation
2-9 Suehiro-cho, Ome-city
Tokyo 198-8710, Japan
naoyasu.ubayashi@toshiba.co.jp

Tetsuo Tamai
University of Tokyo
3-8-1 Komaba, Meguro-ku
Tokyo 153-8902, Japan
tamai@graco.c.u-tokyo.ac.jp

ABSTRACT

Using mobile agent systems, cooperative distributed applications that run over the Internet can be constructed flexibly. However, there are some problems: it is difficult to understand collaborations among agents as a whole; it is difficult to define behaviors of agents because they are influenced by their external context dynamically. So, in general, constructions of cooperative distributed applications based on mobile agent systems are considered as very hard and difficult works.

In this paper, the concept of RoleEP(Role Based Evolutionary Programming) is proposed in order to alleviate these problems. RoleEP provides a systematic evolutionary programming style. In RoleEP, a field where a group of agents collaborate with each other is regarded as an *environment* and a function that an agent assumes in an environment is defined as a role. Descriptions only concerning to collaborations among agents can be abstracted by environments. An object becomes an agent by binding itself with a role that is defined in an environment, and acquires functions needed for collaborating with other agents that exist in the same environment. Distributed applications based on mobile agent systems, which may change their functions dynamically in order to adapt themselves to their external context, can be constructed by synthesizing environments dynamically.

Keywords

Evolution, Environment, Mobile Agents

1. INTRODUCTION

Recently, cooperative distributed applications based on mobile agent systems are increasing. Most of these applications are implemented in Java so that it can run on any platform[16]. Using mobile agents, we can develop cooperative distributed applications that run over the Internet more easily and more flexibly than before. One of the most typical applications is the information retrieval system that searches through the Internet. In this type of systems, an agent that receives requests from a user moves to a host where the information to be searched may exist. If the agent finds the target information, the agent negotiates an agent that manages the information and gets it from the peer agent. After that, the agent goes back to the previous host and returns the result to the user. If the agent cannot find the target information, it roams around other hosts and repeats the process. The agent may decide search-paths dynamically by using information that it gets on the way. The agent may acquire new functions - for example, a function to access directory services - on the way and searches information by using the function. Though it is beneficial to use mobile agent systems in order to build cooperative distributed applications, there are problems as follows: it is difficult to understand collaborations among agents and travels of individual agents as a whole because traveling/collaboration functions come to be intertwined in the code; it is difficult to define behaviors of agents because they are influenced by the external context. Properties of mobile agent based cooperative distributed applications are as follows:

1. An application is constructed through a number of collaborations among agents.
2. Number of agents and kinds of agents change dynamically. Agents may be dispersed over the Internet.
3. Agents may move over the Internet. So, topologies of collaborations among agents may change dynamically.
4. Agents may acquire new functions dynamically in order to adapt themselves to their external context. Although objects do not change their functions through their life-cycles, agents may change their functions dynamically.
5. An agent may participate in a number of collaborations simultaneously. In such a case, the agent is engaged in different roles in different collaborations.

This paper proposes the concept of RoleEP(Role Based Evolutionary Programming) in order to alleviate the above problems. RoleEP provides a systematic evolutionary programming style. In RoleEP, a field where a group of mobile agents collaborate with each other is regarded as an environment and a function that an agent assumes in the field is defined as a role[25]. An environment can be described as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2000 ACM 0-89791-88-6/97/05 ..$5.00

IEEE COMPUTER SOCIETY

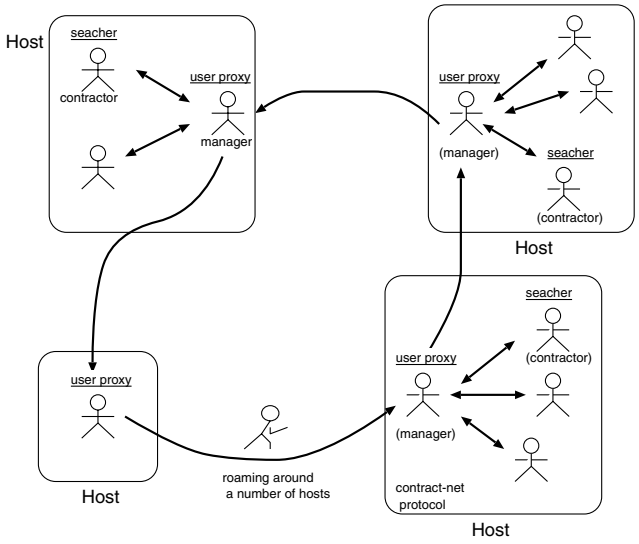

Figure 1: Distributed information retrieval system

an interaction among roles. Cooperative distributed applications based on mobile agent systems, which may change their functions dynamically in order to adapt themselves to their external context, can be constructed by synthesizing multiple environments dynamically. In this paper, problems that may occur when distributed applications are designed by using traditional construction approaches are pointed out in section 2. In section 3, the concept of RoleEP is introduced to address these problems. The framework *Epsilon/J*[1] that realizes RoleEP on Java is explained in section 4, and examples described in Epsilon/J are shown. Section 5 is a discussion on RoleEP. In section 6, reference to a number of works related to RoleEP is given. Lastly, in section 7, we conclude this paper.

2. PROBLEMS OF CONSTRUCTING COOPERATIVE MOBILE AGENT APPLICATIONS

In this section, a distributed information retrieval system – a typical example of cooperative distributed applications based on mobile agent systems – is described by using traditional development approaches, and problems that may occur in those approaches are pointed out. An example problem is illustrated in Figure 1.

Example problem

A user requests an agent to search information on specified topics. The agent divides the request into several subtasks according to the kinds of topics and assigns them to searcher agents that are dispersed over the Internet by roaming around a number of hosts and executing the contract-net protocol[22] at each host. The contract-net protocol is a protocol for assigning tasks to objects through negotiations. In the contract-net protocol, managers and contractors exist. First, a manager announces a task to all contractors. Then, each contractor compares his/her own condition with a condition shown by the manager, and if the latter condition satisfies the former condition, the contractor sends his/her bidding to the manager. The manager selects a contractor that shows the most satisfactory bid-condition and awards the contract to him/her.

[1]This name originates from the head letter of *environment*.

2.1 Case1: Orthodox approach

In the orthodox approach, a program description maps domain structures to program structures. The following program is written in Java [2].

```
public class UserProxy {
 public void roam(){
   :
   dispatch(getNextHostAddress(),
            "contractNet_start");
 }
 public void contractNet_start(){
   :
   // broadcasts a task-announcement message
   //     to all agents existing in the host.
 }
 public void contractNet_bid(){
   :
   // if all biddings are finished,
   //     selects the best contractor.
   :
   best-contractor.award();
 }
 public void contractNet_end(){
   :
   // saves results of the task execution.
   // moves to the next host.
 }}
```

This program starts from the *roam* method. In the *roam* method, the *dispatch* method is called, whose first parameter is a host address to which an agent moves and second parameter is a method that the agent executes at arriving to the host. In the above program, the agent moves to the next host and executes the *contractNet_start* method. In the *contractNet_start* method, the agent (as a manager) broadcasts a task announcement to all agents (as contractors) that exist in the host. The manager agent receives responses from other contractor agents by the *contractNet_bid* method and selects the best contractor agent. Then, the manager agent awards the contract to the contractor agent. The manager agent saves results of the task execution in the *contractNet_end* method and moves to the next host.

In this program, code for roaming around hosts is mixed with code for executing the contract-net protocol. In most cases, distributed applications based on mobile agent systems are described in such a programming style. However, the style has problems as follows:

- It is difficult to understand a program behavior as a whole since traveling/collaboration functions that compose a program are not described separately. For example, code for roaming around hosts is not separated from code for the contract-net protocol in the above program. As a result, it is difficult to extend program code.

If another function is added to this program, the *contractNet_start* method and the *contractNet_bid* method may be

[2]This program is described based on weak mobility[8]. In weak mobility, only program code and instance data are moved. Execution information such as a program counter is not moved. After an agent moves, program execution restarts from the beginning of the method specified at dispatching. The mobility such that execution information is moved is called strong mobility.

IEEE COMPUTER SOCIETY

changed so that this program adapts itself to the new function. These methods will include code that is not related to the contract-net protocol. The approach contains problems that program code becomes more complex as new functions are added to the code.

2.2 Case2: Design-pattern approach

Next, we take the design-pattern approach in order to alleviate the problems that are pointed out in Case1. Collaborations among agents can be structured using design patterns[9]. Recently, design patterns focused on mobile agents are proposed. For example, Aridor, Y. and Lange, D.B. propose design patterns for Aglets[13][20], a typical mobile agent system based on Java, as follows[2]:

- Traveling Patterns: Itinerary,Forwarding,Ticket, etc.
- Task Patterns: Master-Slave, Plan, etc.
- Collaboration Patterns: Meeting, Locker, Messenger, Facilitator, Organized Group, etc.

The following Aglets program is described using the *Itinerary* pattern, a design pattern for roaming around hosts and executing a task at each host. In Aglets, a mobile agent is defined as an instance that is created from a subclass of the *Aglets* class. In this pattern, information for roaming is encapsulated in the instance of the *Itinerary* class. It is only necessary to change the content of instance when host addresses for roaming are changed. Programs can be described that an agent roams around hosts and executes many kinds of tasks at each host by defining the subclass of the *Itinerary* class.

```
public class UserProxy extends Aglets{
 public void roam(){
   // sets sequential planning itinerary
   itinerary = new SeqPlanItinerary(this);
   itinerary.addPlan(HostAddress1,
                     "contractNet_start");
   itinerary.addPlan(HostAddress2,
                     "contractNet_start");
    ⋮
   itinerary.addPlan(HostAddressN,
                     "contractNet_start");
   // starts the trip
   itinerary.startTrip();
 }
 public void contractNet_start(){
    ⋮
   // broadcasts a task-announcement message
   //     to all agents existing in the host.
    ⋮
   // waits until contract-net process is finished
 }
 public void contractNet_bid(){
    ⋮
   // if all biddings are finished,
   //     selects the best contractor.
    ⋮
   best-contractor.award();
 }
 public void contractNet_end(){
    ⋮
   // saves results of the task execution.
    ⋮
   // notifies this agent.
 }}
```

In this program, an instance is created from the *SeqPlanItinerary* class that is a subclass of the *Itinerary* class, and host addresses for roaming and methods that are executed at each host are specified in the *addPlan* method [3]. Here, N host addresses and the *contractNet_start* method are specified. An agent starts to roam around hosts when the instance of the *SeqPlanItinerary* class receives the *startTrip* message. Though the *contractNet_start* method is directly specified as a method that is executed at each host in the *addPlan* method, it can be parameterized so that code for roaming around hosts is separated from code for executing the contract-net protocol. As shown in the program, however, if a roaming agent wants to behave as a manager at the host machine the agent moves into, functions requested for a manager should be described as methods of the agent. So, separations of traveling/collaboration descriptions are limited only within an agent.

2.3 Case3: AOP approach

AOP(Aspect Oriented Programming)[15][11] is a programming paradigm such that a system is divided into a number of aspects and a program is described per aspect. A function that is dispersed among a group of objects is defined as an aspect. A compiler, called *weaver*, weaves aspects and objects together into a system. For example, a complex distributed system is described by a number of aspects including a main aspect, communication aspects and failure handling aspects. These aspects are automatically woven into a single program. Using AOP, it is possible to define a function as a global module that cuts across a set of distributed objects.

AspectJ[3], AOP language, is an aspect-oriented extension to Java. A program in AspectJ is composed of aspect definitions and ordinary Java class definitions. An aspect is defined by *aspect* that is an AspectJ specific language extension to Java. Aspects and classes are are woven together by AspectJ weaver. Main language constructs in AspectJ are *introduces* and *advises*. *Introduces* adds a new method (cross-cutting code is described in the method) to a class that already exists. *Introduces* executes static weaving and affects all objects instantiated from a specified class. That is, all objects have a common introduced method. *Advises* modifies a method that already exists. *Advises* can append cross-cutting code to specified method. *Before* is used in order to append code before a given method, and *after* is used in order to append code after a given method. *Advises* executes static or dynamic weaving. In the case of dynamic weaving, cross-cutting code can be added to a specific object.

Kendall, E.A. proposed role model designs and implementations with AspectJ in [14]. Kendall showed that there are five options for AOP of Role Models as follows:

Option1: Static aspect introduces extrinsic role members to a core class.

Option2: Aspect instance implements role behavior by advising role members that already exist in a core instance.

[3] In the *Itinerary* pattern, only one method can be invoked at each host. However, a set of method executions are needed in order to perform the contract-net protocol. So, in the *contractNet_start* method, after a manager agent broadcasts a task announcement message, it must wait until it gets execution result. If the *contractNet_start* method is described as same as Case1, the agent may move to next host after broadcasting a task announcement.

IEEE COMPUTER SOCIETY

Option3: Aspect instance contains role members separate from a core instance. Two separate entities are used.

Option4: Aspect instance filters out invalid role members from a core instance with advise weaves. The core instance contains members for all roles.

Option5: Role and core are objects. Static aspect integrates or composes them, using introduce weaves. This option is called "Glue Aspects".

Kendall pointed out that no one solution is complete and recommended a hybrid approach: 1) introduces the interface for the role specific behavior to the core class (option1): 2) advises the implementation of the role specific behavior to instances of the core class dynamically (option2); 3) adds role relationships and role context in the aspect instance (option3). The the contract-net protocol can be described as follows by applying Kendall's approach.

```
public class UserProxy{
 public void roam(){
   itinerary = new SeqPlanItinerary(this);
    :
 }}
aspect Manager extends Role{
 // role relationships in aspect (uses option3)
 protected Contractor[] contractor;
 public void setContractor(){...}
 // introduces empty behavior
 //   to the class UserProxy (uses option1)
 introduce public void UserProxy.start(){}
 introduce public void UserProxy.bid(){}
 introduce public void UserProxy.end(){}
 // advise weaves for aspect instances
 //   that will be attached to an instance
 //   of the class UserProxy (uses option2)
 advise public void UserProxy.start(){
   before{ ... } }
 advise public void UserProxy.bid(){
   before{ ... } }
 advise public void UserProxy.end(){
   before{ ... } }
}

public class InfoSearcher{
 public void executeTask(){
   // do something.
 }}
aspect Contractor extends Role{
 // role relationships in aspect
 // (uses option3)
 protected Manager manager;
 public void setManager(){...}
 // introduces empty behavior
 //   to the class InfoSearcher (uses option1)
 introduce public void InfoSearcher.taskAnnounce(){}
 introduce public void InfoSearcher.award(){}
 // advise weaves for aspect instances
 //   that will be attached to an instance
 //   of the class InfoSearcher (uses option2)
 advise public void InfoSearcher.taskAnnounce(){
   before{ ... } }
 advise public void InfoSearcher.award(){
   before{
      :
     // calls a method of the class InfoSearcher
     executeTask();
       :
   }
 }}
```

Although this approach has good points, the following problems still remain.

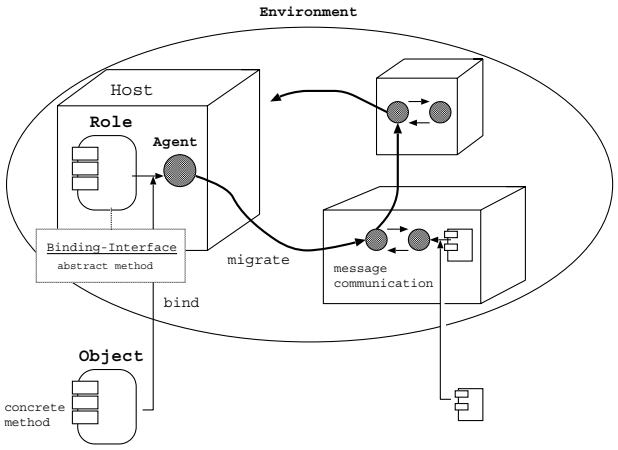

Figure 2: RoleEP model constructs

- Description of aspects depends on specific core classes. The name *UserProxy* appears in the definition of the aspect *Manager*. So, the description of *Manager* cannot be applied to other core classes.
- Description of role behavior depends on interface-names of core classes. That is, when a role uses a method of a core class, the role must call the method directly. In the aspect *Contractor*, *InfoSearcher.award()* must call *executeTask()* that is a method of the core class *InfoSearcher*. In general, there are many kinds of contractor agents that implement their own task execution methods whose names may be different. For example, a contractor agent that has an information searching function may have a task execution method named *searchInfo*. On the other hand, a contractor agent that has an information delivering function may have a task execution method named *deliverInfo*.
- Each aspect must be defined per a role. A description that cross-cuts roles may be dispersed in several aspects.

3. ROLE-EP

3.1 Basic concepts

In this section, RoleEP is proposed as one of approaches that address problems pointed out in section 2. RoleEP is composed of four model constructs – agents, roles, objects and environments as shown in Figure 2. Agents can roam around hosts, collaborate with other agents that exist in the same environment by sending messages to each other and execute its original functions. These functions requested to agents can be separated to traveling/collaboration functions and original functions. Corresponding to the example shown in section 2, functions for roaming around hosts can be regarded as traveling functions and functions for the contract-net protocol can be regarded as collaboration functions. A function that is executed by the *executeTask*(in Case 3) can be regarded as an original function. Original functions are functions that are not related to travels or collaborations directly. In the contract-net protocol, functions of the *executeTask* vary according to target applications. It is desirable to separate original functions from traveling/collaboration

IEEE COMPUTER SOCIETY

functions. If concrete functions of the *executeTask* can be described separately, applications that use the contract-net protocol can be implemented by changing the description of *executeTask*. In RoleEP, these two kinds of functions are described separately. Environments and roles are model constructs that describe traveling/collaboration functions, and objects are model constructs that describe original functions. An agent is composed dynamically by binding an object with a role that belongs to an environment. Detail definitions of environments, roles, agents and objects are as follows[4]:

```
environment ::= [environment attributes,
                 environment methods, roles]
role ::= [role attributes, role methods,
                        binding-interfaces]
agent ::= [roles, object]
object ::= [attributes, methods]
agent.binding-interface => object.method
```

The symbol ::= means that the left hand side is defined as the right hand side. The symbol ⇒ means that the left hand side is replaced by the right hand side. The symbol '.' means a message sending.

3.2 Environment and role

An environment is composed of environment attributes, environment methods and roles. A role, which can move between hosts that exist in an environment, is composed of role attributes, role methods and binding-interfaces. Traveling/collaboration functions including tours around hosts and message communications among agents are described by role attributes and role methods. Role attributes and role methods are only available in an environment that the role belongs to. A binding-interface, which looks like an abstract method interface, is used when an object binds itself with an role. Mechanisms of binding-interfaces and binding-operations are explained after. Common data and functions that are used in roles are described by environment attributes and methods. Directory services such as role-lookup-services are presented as built-in environment methods. A travel or collaboration is encapsulated by an environment and roles.

3.3 Object and agent

An object, which cannot move between hosts, is composed of attributes and methods. Although an object cannot move between hosts, it can move by binding itself with a role that has traveling/collaboration functions. An object can bind with roles that exist in the same host. If an object binds itself with two role – one role has a function for moving to *host A* and another role has a function for moving to *host B* –, the object cannot decide where it may go. An object cannot bind with roles that do not exist in the same host. An Object that binds itself with some roles and acquires traveling/collaboration functions is called *agent*. An object becomes an agent by binding itself with a role that belongs to an environment, and can collaborate with other agents within the environment. An object can participate in a number of environments simultaneously. Agent identity is same as object identity. Role identities can be regarded as aliases of the object identity. An agent can be recognized by its role identity from other agents that exist in the same environment.

[4]Environment, role, agent and object are instances.

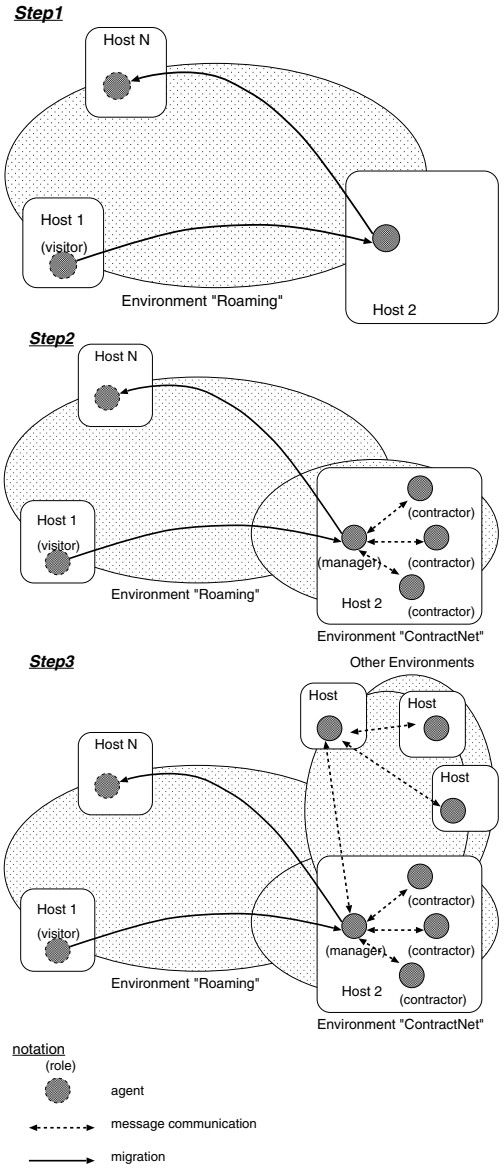

Figure 3: Dynamic evolution of environment

3.4 Binding-operation

Binding-operations are implemented by creating delegational relations between roles and objects dynamically. That is, if a role receives a message corresponding to a binding-interface, the role delegates the message to an object bound with the role. For example, if the binding-interface *executeTask* defined in a role is bound to the *searchInfo* method defined in an object, the message "executeTask" received by the role is renamed to "searchInfo" and delegated to the object. Many kinds of collaborations can be described by changing combinations between roles and objects. Binding-operations in RoleEP correspond to *weaver* in AOP.

3.5 Example

Figure 3 illustrates the notion of RoleEP. In general AOP, aspects that construct a system are statically defined when the system is designed, and do not change from the begin-

IEEE COMPUTER SOCIETY

ning of computation to the end. On the other hand, environments proposed in RoleEP can be defined dynamically and compositions of environments can be re-arranged dynamically. A distributed application based on mobile agent systems is composed of a number of environments that can be added or deleted dynamically. Number, kinds and topologies of collaborations among agents may change dynamically. Compositions of environments can be re-arranged dynamically as a distributed application evolves its functions dynamically in order to adapt itself to its external context. Participating in environments, an agent can be engaged in several roles and can collaborate with other agents that exist in each environment.

4. JAVA ROLE-EP FRAMEWORK

Epsilon/J is a framework that supports RoleEP concepts including environment and roles. This framework, which is presented as class libraries, is implemented on Aglets that is a mobile agent system based on Java. In this section, features of Epsilon/J are explained through describing the example presented in section 2.

4.1 Environment descriptions

In Epsilon/J, an environment class is defined as a subclass of the *Environment* class, and a role class is defined as a subclass of the *Role* class. The following is a program that defines the environment class *Roaming* and *ContractNet*. In the *Roaming* environment class, the *Visitor* role class is defined. An object becomes an agent that can roam around hosts by binding itself with a role instantiated from the *Visitor* class. On the other hand, in the *ContractNet* environment class, the *Manager* role class and the *Contractor* role class are defined. An agent, which arrives from other host, acquires new functions that are necessary to behave as a manager by binding itself with a role instantiated from the *Manager* role.

The Roaming environment class

```
public class Roaming extends Environment{
  public class Visitor extends Role{
    public void onRoleCreation(String roleName){
      // adds this role
      //    whose name is specified by "roleName"
      //    to the "Roaming" environment.
      thisEnvironment.addRole(roleName, this);
      addBindingInterface("executeTask");
    }
    public void roam(){
      // sets sequential planning itinerary.
      itinerary = new SeqPlanItinerary(this);
      itinerary.addPlan(HostAddress1, "executeTask");
      itinerary.addPlan(HostAddress2, "executeTask");
        ⋮
      itinerary.addPlan(HostAddressN, "executeTask");
      // starts the trip.
      itinerary.startTrip();
    }}}
```

The *onCreation* method is called when a role is instantiated. In the *Visitor* role class, the binding-interface *executeTask*, which is an interface of a method that is invoked when an agent arrives at a host, is added dynamically. An agent, which is composed of an object and an instance created from the *Visitor* role class, roams around hosts and execute the *executeTask* at each host.

The ContractNet environment class

```
public class ContractNet extends Environment{
  public class Manager extends Role{
    public void onRoleCreation(String roleName){
      thisEnvironment.addRole(roleName, this);
    }
    public void start(){}
    public void bid(){}
    public void end(){}
  }
  public class Contractor extends Role{
    public void onCreation(){
      thisEnvironment.addRole(roleName, this);
      addBindingInterface("executeTask");
    }
    public void award(){}
  }}
```

4.2 Object descriptions

The following is a program that define the class *UserProxy* and the class *SearchInfo*. An object instantiated from the *UserProxy* class is bound with a visitor role instantiated from the *Visitor* class and a manager role instantiated from the *Manager* class. A class of an Epsilon/J's object is defined as a subclass of the *EpsilonObj* class that presents functions for binding-operations.

The UserProxy class

```
public class UserProxy extends EpsilonObj{
  public void life() {
      :
    // searches a visitor role
    //   and binds itself with the visitor role.
    // this object becomes an agent
   //    that can roam around hosts.
    visitorRole.bind(this, "executeTask",
                           "executeContractNet");
    visitorRole.roam();
   }
  public void executeContractNet() {
      :
    // searches a manager role
    //   and binds itself with the manager role.
    // this object becomes an agent
    //   that can act as a manager
    //   in the contract-net protocol.
    managerRole.bind(this);
        // there are not binding-interfaces.
    managerRole.start();
  }}
```

The InfoSearcher class

```
public class InfoSearcher extends EpsilonObj{
  public void life() {
    // searches contractor roles existing in
    //   the environment "contractnetEnv"
    //   that is instantiated from "ContractNet".
    Contractor [] allContractorRoles
      = contractnetEnv.searchRole("Contractor");
    // select a role "contractorRole"
    //   from "allContractorRoles"
    //   and binds itself with the role.
    // this object becomes an agent
    //   that can act as a contractor
    //   in the "contractnetEnv".
    contractorRole.bind(this, "executeTask",
                              "searchInfo");
      :
  }
  public void searchInfo(){}
  public void search2Info(){}
}
```


IEEE COMPUTER SOCIETY

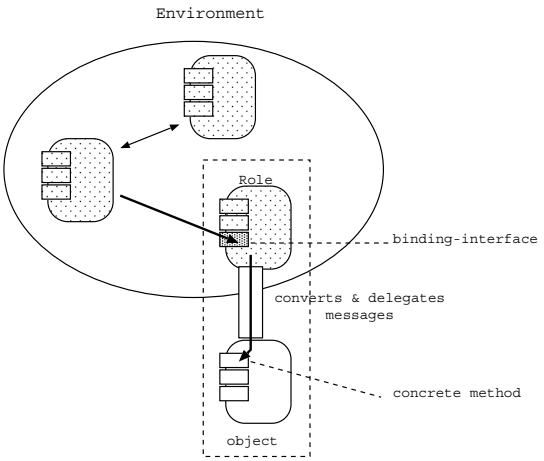

Figure 4: Binding-operation

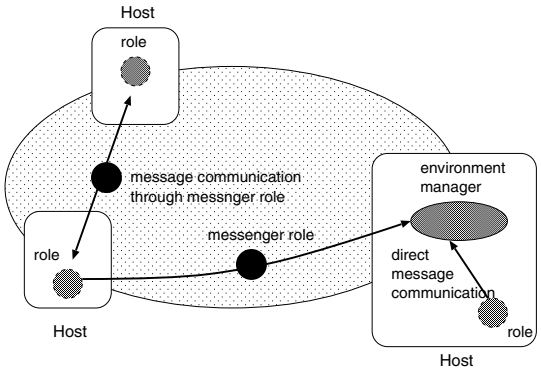

Figure 5: Messenger role

4.3 Epsilon/J implementation

Epsilon/J presents built-in classes including *Environment*, *Role* and *EpsilonObject*. Mechanisms such as binding-interfaces and binding-operations are contained in these built-in classes that are implemented by using Java core reflection APIs (Application Programming Interfaces). Using reflection mechanisms, method signatures defined in objects/roles/environments can be introspected and invoked dynamically. If a message received by a role corresponds to a binding-interface, the message is transformed to a signature that is specified as an argument in a binding-operation and delegated to an object bound with the role (Figure 4). Since mechanisms of binding-interfaces and binding-operations are implemented very simply in Epsilon/J, performance-down that is caused by adding this kind of RoleEP features to original Aglets mobile agent system is a little. In Aglets, moreover, dynamic method dispatching mechanism is already used in order to realize a message as an object.

Other features such as implementation of environment methods/attributes may raise a discussion. In Epsilon/J, information on an environment such as role references, role host addresses and environment methods/attributes is stored intensively in a host where the environment is instantiated. If a role does not reside in a host where the corresponding environment exists, the role has to execute remote-accessing in order to use above kind of information. In Epsilon/J, the notion of *messenger role* is introduced in order to realize role-to-role remote communication and role-to-environment remote communication as shown in Figure 5. Messenger role is a special role that brings a message object from one host to another host. In Epsilon/J, an API function for message communication is prepared to encapsulate existence of messenger roles. This API function decides automatically whether communication is remote or local. If communication is remote, the API function uses a messenger role. Otherwise, the function sends a message normally. Mechanism of messenger role may cause some kind of performance-down.

5. DISCUSSIONS

By introducing RoleEP, problems pointed out in section 2 can be solved as follows:

- Traveling/collaboration functions can be separated from original functions completely and can be encapsulated within environment descriptions. This solves problems appeared commonly in Case 1, 2, and 3.
- Agents, which are objects bound with roles, can acquire new functions in order to collaborate with other agents through binding-operations. This solves problems pointed out in Case 1 and 2.
- The problem that descriptions of role behavior depends on interface names of core classes in Case 3 can be solved by the binding-interface mechanism.

All characteristics of distributed applications based on mobile agent systems pointed out in section 1 can be described completely in RoleEP. Moreover, there are nice properties as follows:

1. Construction mechanisms for traveling/collaboration components: RoleEP is beneficial for constructing traveling/collaboration components. For example, the environment class *ContractNet* can be reused in many distributed applications based on mobile agent systems. Environment classes can be regarded as traveling/collaboration components.
2. Evolution mechanisms for agents: In RoleEP, an object becomes an agent by binding itself with a role that belongs to an environment. An object can dynamically evolve to an agent that can behave multiple roles. Using RoleEP, programs that adapt to external context can be described easily.
3. Agentification mechanisms: Genesereth, M.R. and Ketchpel, S.P. shows three approaches for converting objects into agents[10]: 1) an approach that implements a transducer that mediates between an object and other agents, 2) an approach that implements a wrapper, and 3) an approach that rewrites an original objects. In RoleEP, a role corresponds to a transducer that accepts messages from other agents and translates them into messages that an object can understand. Although general agentifications are implemented statically, a connection between an object and a transducer is created dynamically through a binding-operation in RoleEP. RoleEP can be regarded as one of dynamic agentification mechanisms.

IEEE COMPUTER SOCIETY

viewpoint	AOP	RoleEP
aspects	aspects	environments and roles
components	components	objects
joint points	join points	roles
(between aspects and components)		
weaving method	weaver	binding-operation
aspect reuse	emphasized	emphasized
dynamic aspect syntheses	not so emphasized	emphasized
dynamic evolution	not so emphasized	emphasized
dynamic method adding	emphasized	emphasized
dynamic method modification	emphasized	—

Table 1: AOP vs RoleEP

Table1, which extends an AOP comparison method proposed in [5], compares RoleEP with AOP. RoleEP emphasizes dynamic aspect syntheses and dynamic evolution.

In RoleEP, the use of binding-operation eliminates opportunities for AOP style weaving. *Introduces* weaving in AspectJ can be replaced by adding role methods through binding-operation. However, *advises* weaving does not correspond to any model constructs in RoleEP. This is a weak point of RoleEP, and reduces ability to prevent code duplication. From a viewpoint of static evolution, *advises* weaving is very useful because it prevent code duplication. From a viewpoint of dynamic evolution, however, *advises* weaving is slightly danger because it is difficult to understand real behaviors. In Kendall's approach, *introduces* weaving only adds a method interface, and the body of the method is added through *advises* weaving. This kind of *advises* weaving can be realized by binding-operation.

6. RELATED WORKS

Bardou, D. shows comparison AOP and related approaches including Role Modeling[1], Activities and roles[19], Subject-Oriented Programming[12], Split objects[4] and Us "a subjective version of SELF"[23] in [5].

Role modeling consists of *role model* and *role*. A *role model* is a design unit that describes a collaboration among roles. Several role models are synthesized by a technique called projection of role. By a projection, a new aggregate role is created that manages related roles. Role models in Role Modeling correspond to environments in RoleEP. Including Role modeling, there are several researches concerning to role concepts[17][18][7][24]. VanHilst, M. and Notkin, D. propose an idea of role components, which are described by C++ templates, to implement collaboration-based design[26]. This approach looks like the mix-in approach. In *Coordinated Roles* proposed by [21], descriptions of collaborations are separated from descriptions of objects by using role concepts. This approach looks like the binding-interface concepts. However, there are not concepts of dynamic binding between an object and a role in *Coordinated Roles*. In these approaches, dynamic evolution or dynamic synthesis of collaboration structures is not so emphasized. In most cases, collaboration structures are synthesized at compile-time. Split objects and Us are based on delegation mechanisms that enable dynamic evolution. In Split objects that presents a way to express viewpoints, each object in a delegation hierarchy of a split representation denotes a different viewpoint on the represented entity[5]. However, language constructs on collaborations are not presented sufficiently. Kristensen, B.B. proposed notion of *transverse activities* and *role*. An activity is a single entity defined by a set of participants and a directive that describes a collaboration among participants.

On the other hand, adaptations to external context are studied from the viewpoint how a single object evolves itself dynamically - for example, how a person acquires methods and attributes when he gets a job, he gets married and so on. In the Subject Oriented Programming, an object acquires new functions by participating in subjects. The concept of subjects are similar to the concept of environments in RoleEP. *Mobile Ambients* is a model that gives a layered agent structure[6]. In this model, agents run on fields constructed by synthesizing contexts (environments) dynamically.

7. CONCLUSIONS

Distributed applications that reside in mobile/Internet/Intra-net environments, whose structures change dynamically, are spreading rapidly. Most applications are implemented in traditional programming languages, and have many embedded logics according to individual environments. These applications must switch to new logics as environments change. As a result, these applications need to be restructured drastically when they must adapt to new environments. It is necessary to have new computation paradigms and programming languages in order to alleviate such a problem. RoleEP that we have proposed in this paper is one approach to address such a problem.

8. REFERENCES

[1] Andersen, E.P. and Reenskaug, T.: System Design by Composing Structures of Interacting Objects, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, Lecture Notes in Computer Science, Springer, vol.615, pp.133-152, 1992.

[2] Aridor, Y. and Lange, D.B.: Agent design patterns: Elements of agent application design, *Proceedings of Agents'98*, 1998.

[3] AspectJ. http://aspectj.org/.

[4] Bardou, D. and Dony, C.: Split Objects: a Disciplined Use of Delegation within Objects, *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'96)*, pp.122-137, 1996.

[5] Bardou, D.: Roles, Subjects and Aspects: How do they relate?, *Proceedings of the Aspect-Oriented*

IEEE COMPUTER SOCIETY

Programming Workshop at ECOOP'98, 1998.

[6] Cardelli, L. and Gordon, A.D.: Mobile Ambients (Extended Abstract), *the proceedings of the workshop on Higher Order Operational Techniques in Semantics*, 1997.

[7] Fowler, M.: Dealing with Roles, *Proceedings of the 4th Annual Conference on Pattern Languages of Programs*, 1997.

[8] Fuggetta, A., Picco, G.P.d and Vigna, G.: Understanding Code Mobility, *IEEE Transactions on Software Engineering*, vol.24, No.5, pp.342-361, 1998.

[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns*, Addison-Wesley Publishing Company, Inc., 1995.

[10] Genesereth, M.R. and Ketchpel, S.P.: *Software Agents*, *Communications of the ACM*, vol.37, No.7, pp.48-53, 1994.

[11] Guerraoui, R. et al.: Strategic Directions in Object-Oriented Programming. *ACM Computing Surveys*, Vol.28, No.4, pages 691-700, 1996.

[12] Harrison, W. and Ossher, H.: Subject-oriented Programming, *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'93)*, pp.411-428, 1993.

[13] IBM: *Aglets Software Development Kit Home Page*, http://www.trl.ibm.co.jp/aglets/index.html, 1999.

[14] Kendall, E.A.: Role Model Designs and Implementations with Aspect-oriented Programming, *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'99)*, pp.353-369, 1999.

[15] Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science, Springer, vol.1241, pp.220-242, 1997.

[16] Kiniry, J. and Zimmerman, D.: A Hands-On Look at Java Mobile Agents, *IEEE Internet Computing*, vol.1, No.4, 1997.

[17] Kristensen, B.B.: Object-oriented Modeling with Roles, *Proceedings of the 2nd International Conference on Object-oriented Information Systems (OOIS'95)*, 1996.

[18] Kristensen, B.B. and Osterbye, K.: Roles: Conceptual Abstraction Theory and Practical Language Issues, *Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-oriented Systems*, 1996.

[19] Kristensen, B.B. and May, D.C.M.: Activities: Abstractions for Collective Behavior, *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, Lecture Notes in Computer Science, Springer, vol.1098, pp.472-501, 1996.

[20] Lange, D. and Oshima M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[21] Murillo, J.M., Hernandez, J., Sanchez, F. and Alvarez, L.A.: Coordinated Roles: Promoting Re-usability of Coordinated Active Objects Using Event Notification Protocols, *COORDINATION'99 Proceedings*, pp.53-68, 1999.

[22] Smith, R.G.: The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. on Computers*, vol.29, No.12, pp.1104-1113, 1980.

[23] Smith, R.B. and Ungar, D.: Programming as an Experience: The Inspiration for Self, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, Lecture Notes in Computer Science, Springer, vol.952, pp.303-330, 1995.

[24] Tamai, T.: Objects and roles: modeling based on the dualistic view, *Information and Software Technology*, Vol. 41, No. 14, pp. 1005–1010, 1999.

[25] Ubayashi, N. and Tamai, T.: An Evolutional Cooperative Computation Based on Adaptation to Environment, *Proceedings of Sixth Asia Pacific Software Engineering Conference (APSEC'99)*, IEEE Computer Society, pp.334-341, 1999.

[26] VanHilst, M. and Notkin, D.: Using Role Components to Implement Collaboration-Based Designs, *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'96)*, pp.359-369, 1996.

IEEE COMPUTER SOCIETY