Separation of Concerns in Mobile Agent Applications

Naoyasu Ubayashi[1] and Tetsuo Tamai[2]

[1] Systems Integration Technology Center, Toshiba Corporation, Tokyo, Japan,
naoyasu.ubayashi@toshiba.co.jp
[2] Graduate School of Arts and Sciences, University of Tokyo, Tokyo, 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 and travels of individual agents as a whole because mobility/collaboration functions tend to be intertwined in the code; it is difficult to define behaviors of agents explicitly because they are influenced by their external context dynamically. Many aspects of mobility/collaboration strategies including traveling, coordination constraints, synchronization constraints and security-checking strategies should be considered when mobile agent applications are constructed.
In this paper, the concept of RoleEP(Role Based Evolutionary Programming) is proposed in order to alleviate these problems. In RoleEP, a field where a group of agents roam around hosts and collaborate with each other is regarded as an *environment* and mobility/collaboration functions that an agent should assume in an environment are defined as roles. An object becomes an agent by binding itself to a role that is defined in an environment, and acquires mobility/collaboration functions dynamically. RoleEP provides a mechanism for separating concerns about mobility/collaboration into environments and a systematic evolutionary programming style. 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.

1 Introduction

Recently, cooperative distributed applications based on mobile agent systems are increasing. Most of these applications are implemented in Java so that they can run on any platform[18]. Using mobile agents, cooperative distributed applications can be developed that run over the Internet more easily and more flexibly than before. However, there are problems as follows: it is difficult to understand collaborations among agents and travels of individual agents as a whole because mobility/collaboration functions tend to be intertwined in the code; it is difficult to define behaviors of agents explicitly because they are influenced by the external context. Many aspects of mobility/collaboration strategies including

traveling, task executions, coordination constraints, synchronization constraints, security-checking strategies and error-checking strategies should be considered when mobile agent applications are constructed.

This paper proposes the concept of RoleEP(Role Based Evolutionary Programming) in order to alleviate the above problems. In RoleEP, a field where a group of agents roam around hosts and collaborate with each other is regarded as an *environment* and mobility/collaboration functions that an agent should assume in an environment are defined as roles[28]. Mobile agent applications that may change their functions dynamically in order to adapt themselves to their external context can be constructed by synthesizing multiple environments dynamically. There are two contributions in this paper: 1) RoleEP provides a mechanism for separating concerns about mobility/collaboration in mobile agent applications; 2) RoleEP gives a systematic and dynamic evolutionary programming style. 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 shows how to implement Epsilon/J using a reflection mechanism. Section 6 is a discussion on RoleEP. In section 7, reference is made to a number of works related to RoleEP. Lastly, in section 8, 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 approaches, and problems that may occur in those approaches are pointed out. An example is illustrated in Figure 1.

Example 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 hosts and executing the contract-net protocol[25] 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 its own condition with a condition shown by the manager, and if the former condition satisfies the latter condition, the contractor sends its bid to the manager. The manager selects a contractor that shows the most satisfactory bid-condition and awards the contract to it. There are two aspects to be considered in this example. One is a mobility aspect and the other is a collaboration aspect (contract-net protocol). We want to describe these two aspects as separately as possible.

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

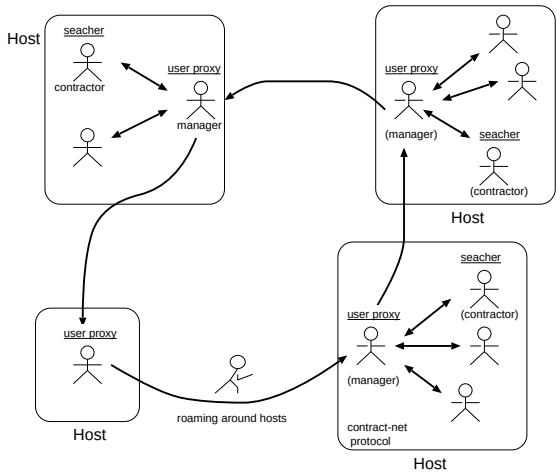

Fig. 1. Distributed information retrieval system

2.1 Case1: Orthodox approach

In the orthodox approach, a program description maps domain structures to program structures. The following program is written in quasi-code similar to Java.

```
/**
 *  User Proxy
 */
public class UserProxy{
 public UserProxy(){
   roam();
 }
 public void roam(){
   // Move to the next host
   //   and execute the method "contractNet_start".
 }
 public void contractNet_start(){
   // Multicast a task-announcement message "contractNet_taskAnnounce"
   //   to all information searchers that exist in the current host.
 }
 public void contractNet_bid(InfoSearcher i){
   // Receive responses from information searchers.
   // Select the "best_contractor"
   //   if all bids are finished.
   best_contractor.contractNet_award(this);
 }
 public void contractNet_end(Result r){
   // Save the information from the "best_contractor".
   // Move to the next host
   //   and execute the method "contractNet_start" again.
   roam();
 }
}
```

```
/**
 *  Information Searcher
 */
public class InfoSearcher{
 public InfoSearcher(){}
 public void contractNet_taskAnnounce(UserProxy  u){
   // Compare my condition with a condition shown by the user proxy.
   // Send a message "contractNet_bid" to the user proxy
   //   if the searcher's condition satisfies the user proxy's condition.
   u.contractNet_bid(this);
 }
 public void contractNet_award(UserProxy  u){
   // Execute task
   //   and return the result to the user proxy.
   u.contractNet_end(executeTask());
 }
 public Result executeTask(){
   // Search information.
 }
}
public class Result{}
```

This program is described based on weak mobility[10] in which only program code and instance data are moved. This program starts from the constructor *UserProxy()* that calls *roam* method. In the *roam* method, 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.

It is difficult to understand behaviors of this program as a whole since mobility/collaboration functions that compose a program are not described separately. Code for roaming around hosts is mixed with code for executing the contract-net protocol. Moreover, 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 have to be changed to adapt itself to the new function. These methods will include code that is not related to the contract-net protocol. The problem with this approach is that the 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[11] approach that may alleviate the problems that are pointed out in Case1. Recently, design patterns focused on mobile agents are proposed. For example, Aridor and Lange propose design patterns for Aglets[15][22], a typical mobile agent system based on Java, as follows[3]:

1. Traveling Patterns: Itinerary, Forwarding, Ticket, etc.
2. Task Patterns: Master-Slave, Plan, etc.
3. 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 created from a subclass of the *Aglets* class. In this pattern, information for roaming is encapsulated in an instance of the *Itinerary* class. It is only necessary to change the content of the instance when host addresses for roaming are changed.

```
/**
 *  User Proxy Aglet
 */
public class UserProxy extends Aglets{
 public void onCreation(Object init){
   // Initialize the aglet.
   // Only called the very first time this aglet is created.
   roam();
 }
 public void roam(){
   // Set sequential planning itinerary.
   SeqPlanItinerary itinerary = new SeqPlanItinerary(this);
   itinerary.addPlan(HostAddress1, "contractNet_start");
   itinerary.addPlan(HostAddress2, "contractNet_start");
    :
   itinerary.addPlan(HostAddressN, "contractNet_start");
   // Start the trip.
   itinerary.startTrip();
 }
 public void contractNet_start(){}
 public void contractNet_bid(InfoSearcher i){}
 public void contractNet_end(Result r){}
}

/**
 *  Information Searcher Aglet
 */
public class InfoSearcher{
 public void onCreation(Object init){}
 public void taskAnnounce(UserProxy u){}
 public void award(UserProxy u){}
 public Result executeTask(){}
}
```

In Aglets, a constructor is specified by the *onCreation* method. The *onCreation* is only called the very first time an aglet is created. In this program, *onCreation* calls *roam* method in which 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. Here, N host addresses and the *contractNet_start* method are specified. An agent starts to roam around hosts when an instance of the *SeqPlanItinerary* class receives a *startTrip* message.

Although a mobility function (code for roaming around hosts) is separated from a collaboration function (code for executing the contract-net protocol), both of these two functions must be described as methods of an agent. So, separations of mobility/collaboration descriptions are limited only within an agent. If the user proxy agent must have other collaboration functions in addition to the contract-net protocol, the code of this agent will be more complex. The mix-in approach is often used in order to address this kind of problems. In this case, functions requested for a manager can be described in a superclass. If an agent has many roles, the agent must inherit corresponding superclasses statically. So, program code must be modified whenever roles requested for an agent are added or deleted. Moreover, multiple inheritances are not allowed in Java.

2.3 Case3: Role model & AOP approach

AOP(Aspect Oriented Programming)[17][13][8] is a programming paradigm such that a concern that cross-cuts a group of objects is modularized as an aspect. A compiler, called *weaver*, weaves aspects and objects together into a system. Concerns including error-checking strategies, synchronization policies, resource sharing, distribution concerns and performance optimizations are examples of aspects.

AspectJ[4], 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 woven together by AspectJ weaver. Main language notions in AspectJ are *introduces* and *advises*. *Introduces* adds a new method in which cross-cutting code is described to a class that already exists. *Advises* modifies a method that already exists. *Advises* can append cross-cutting code to a 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.

Kendall proposed role model designs and implementations with AspectJ in [16]. In the role model, an object has core intrinsic attributes/methods and a role that adds extrinsic attributes/methods provides perspectives that can be used by other objects. Kendall recommended an approach: 1) introduce the interface for the role specific behavior to the core class; 2) advise the implementation of the role specific behavior to instances of the core class; 3) add role relationships and role contexts in the aspect instance. The contract-net protocol can be described as follows[2].

[2] The syntax of this program is based on AspectJ 0.8beta3.

UserProxy

```
/**
 *  User Proxy Aglet
 */
public class UserProxy extends Aglets{
 public void onCreation(Object init){
   roam();
 }
 public void roam(){
   SeqPlanItinerary itinerary = new SeqPlanItinerary(this);
    :
 }
}

/**
 *  Manager Aspect
 */
aspect Manager extends Role{
 // Role relationships in aspect
 protected Contractor[] contractor;
 // Introduce empty behavior to the class UserProxy.
 public void UserProxy.start(){}
 public void UserProxy.bid(InfoSearcher i){}
 public void UserProxy.end(Result r){}
 // Advise an instance of the class UserProxy.
 before(UserProxy u): instanceof(u) && receptions(public void start){
   // Multicast a task-announcement message "taskAnnounce"
   //   to all information searchers that exist in the current host.
 }
 before(UserProxy u): instanceof(u) && receptions(public void bid){
   // Receive responses from information searchers.
   // Select the best one "best_contractor"
   //   if all bids are finished.
   best_contractor.award(this);
 }
 before(UserProxy u): instanceof(u) && receptions(public void end){
   // Save the information from the "best_contractor".
 }
}
```

InfoSearcher

```
/**
 *  Information Searcher Aglet
 */
public class InfoSearcher extends Aglets{
 public void onCreation(Object init){}
 public Result executeTask(){}
}

/**
 *  Contractor Aspect
 */
aspect Contractor extends Role{
 // Role relationships in aspect
 protected Manager manager;
```

```
 // Introduce empty behavior to the class InfoSearcher.
 public void InfoSearcher.taskAnnounce(UserProxy  u){}
 public void InfoSearcher.award(UserProxy u){}

 // Advise an instance of the class InfoSearcher.
 before(InfoSearcher i): instanceof(i) && receptions(public void taskAnnounce){
   // Compare my condition with a condition shown by the user proxy.
   // Send a message "bid" to the user proxy
   //   if the searcher's condition satisfies the user proxy's condition.
   u.bid(this);
 }
 before(InfoSearcher i): instanceof(i) && receptions(public void award){
   // Execute task
   //   and return the result to the user proxy.
   u.end(executeTask());
 }
}
```

Although a mobility function is completely separated from a collaboration function in this approach, the following problems still remain.

1. 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.
2. Description of role behavior depends on method 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 contractors that implement their own task execution methods whose names may be different. For example, a contractor that has an information searching function may have a task execution method named *searchInfo*. On the other hand, a contractor that has an information delivering function may have a task execution method named *deliverInfo*.
3. An aspect must be defined per role. A description that cross-cuts roles may be dispersed in several aspects.

3 RoleEP

3.1 Basic concepts

In this section, RoleEP, an approach that addresses the problems pointed out in section 2, is proposed. RoleEP provides the following for constructing mobile agent applications: 1) a mechanism for separating concerns about mobility/collaboration including traveling, task executions, coordination constraints, synchronization constraints, security-checking strategies and error-checking strategies; 2) a systematic and dynamically evolvable programming style.

RoleEP is composed of model constructs including agents, roles, objects and environments as shown in Figure 2. Agents can roam around hosts, collaborate

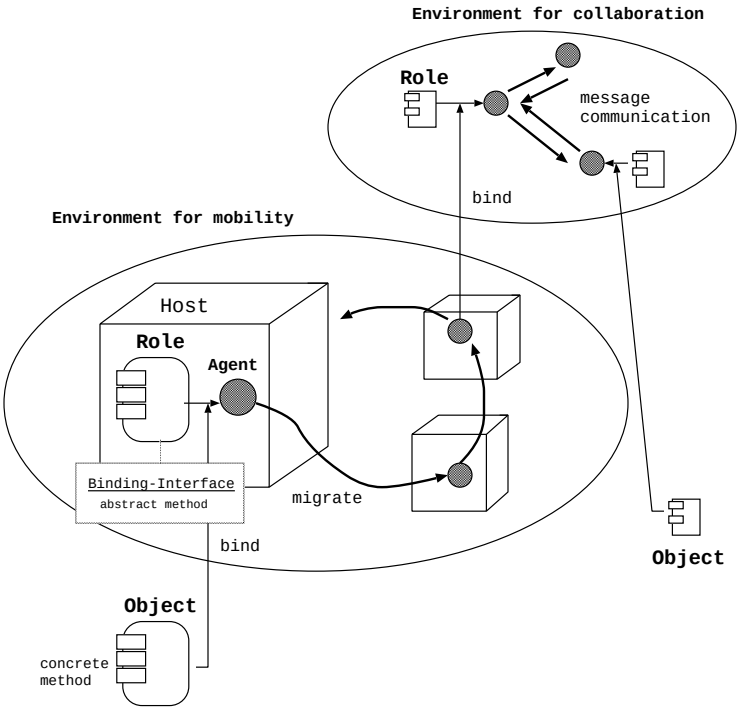

Fig. 2. RoleEP model constructs

with other agents that exist in the same environment by sending messages to each other and execute their original functions. These functions requested to agents can be separated to mobility/collaboration functions and original functions. Original functions are common to all kinds of environments and do not contain mobility/collaboration functions that are given by specific environments. Corresponding to the example shown in section 2, functions for roaming around hosts can be regarded as mobility 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 that is commonly used by not only the contract-net protocol but also other kinds of collaborations. 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 mobility/collaboration 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 mobility/collaboration functions, and objects are model constructs that describe original functions[28]. An agent is composed dynami-

cally by binding an object to a role that belongs to an environment. Syntactic definitions of environments, roles, agents and objects are as follows[3]:

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

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. Mobility/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 to which the role belongs. A binding-interface, which is similar to an abstract method interface, is used when an object binds itself to a role. The mechanisms of binding-interfaces and binding-operations are explained later. 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. The notion of roles in RoleEP extends the role model in 2.3 so that roles can have not only collaboration functions but also mobility functions.

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 to a role that has mobility functions. An object becomes an agent by binding itself to 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. The agent identifier is the same as the object identifier. Role identifiers can be regarded as aliases of the object identifier. An agent can be referenced by its role identifier from other agents that exist in the same environment.

3.4 Binding-operation

Figure 3 shows the notion of the *binding-operation* that binds binding-interfaces of roles to concrete methods of objects. The binding-interface defines the interface in order to receive messages from other roles existing in the same environment. Using the binding-interface, collaborations among a set of roles can be described separately from each object. The binding-operation is permitted only

[3] Environment, role, agent and object are instances. The symbol ::= means that the left-hand side is defined by the right-hand side.

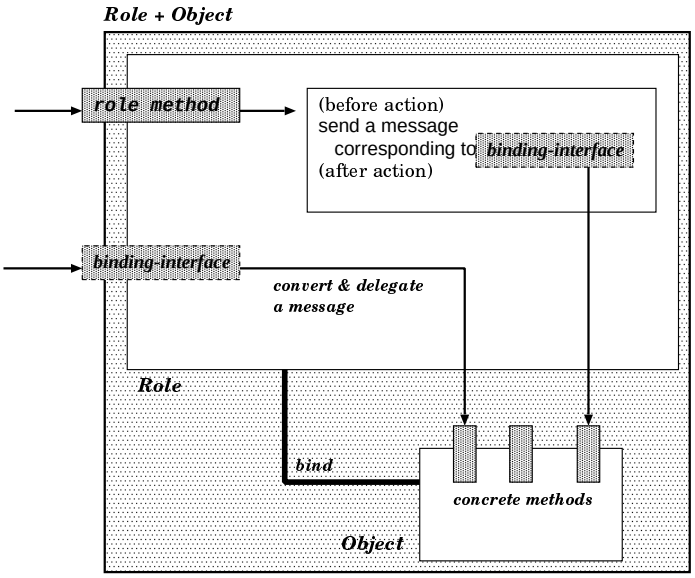

Fig. 3. Binding-operation

when an object has methods corresponding to the binding-interface. Binding-operations are implemented by creating delegational relations between roles and objects dynamically. That is, if a role receives a message corresponding to its binding-interface from other roles or itself, the role delegates the message to an object bound to 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 "searchInfo" and delegated to the object. Many kinds of collaborations can be described by changing combinations of roles and objects. Binding-operations correspond to *weaver* in AOP, and binding-interfaces correspond to *join points* that are weaving points in AOP.

3.5 Example

Figure 4 illustrates the example in section 2 using the notion of RoleEP. In step 1, the user proxy object binds itself to the *visitor* role in the *Roaming* environment at host 1 and becomes the agent. This agent can roam around hosts using mobility functions given by the *visitor* role. In step 2, the user proxy agent binds itself to the *manager* role in the *ContractNet* environment at host 2 and can execute the contract-net protocol using collaboration functions given by the *manager* role. Step 3 shows that the user proxy agent acquires other kinds of role functions after step 2. Figure 4 shows dynamic compositions of environments. Environments of the user proxy agent are composed by *Roaming*, *ContractNet* and so on.

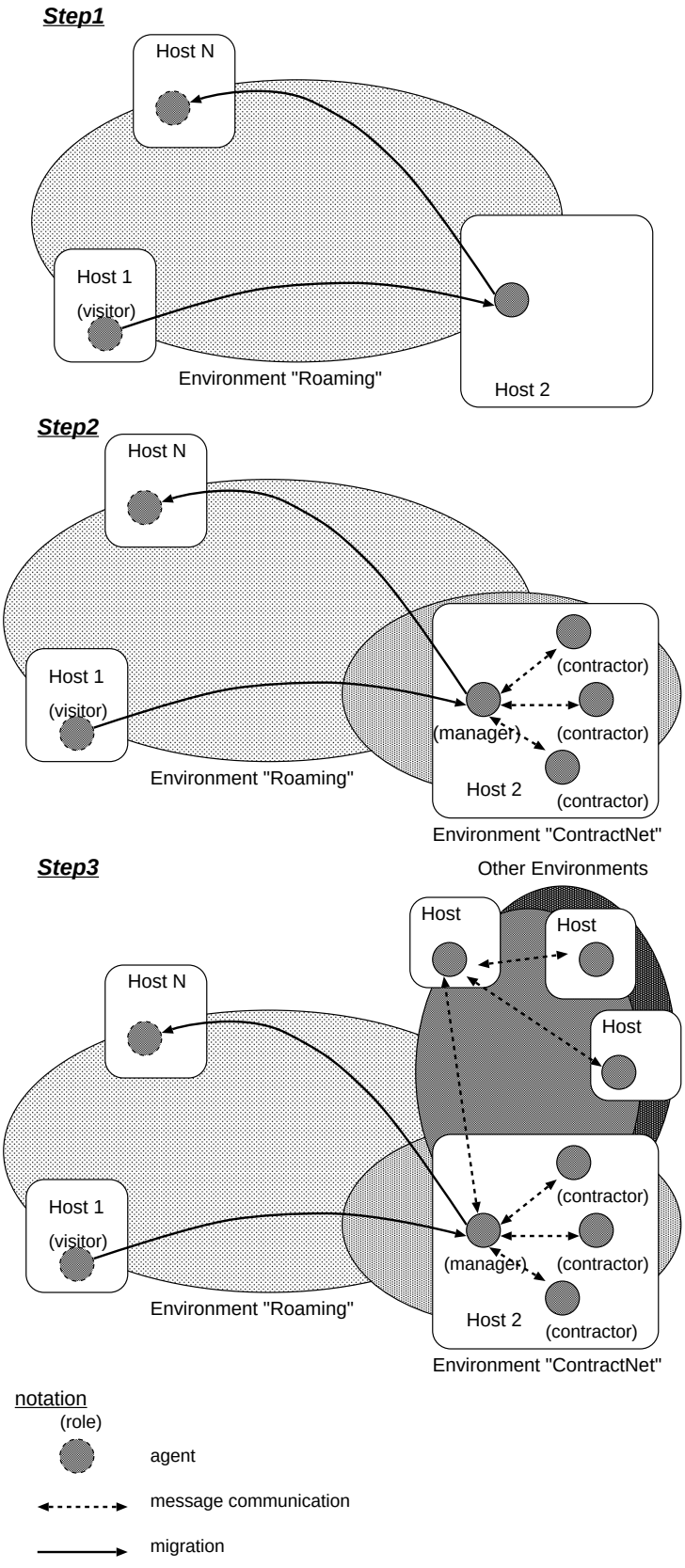

Fig. 4. Dynamic evolution of environment

In general AOP, aspects that construct a system are statically defined when the system is designed, and do not change from the beginning of computation to the end. On the other hand, environments proposed in RoleEP can be defined separately and compositions of environments can be re-arranged dynamically. A distributed application based on mobile agents is composed 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 engage in multiple roles and collaborate with other agents of each environment.

4 Java RoleEP Framework Epsilon/J

Epsilon/J is a framework that supports RoleEP concepts including environment and roles. This framework is implemented on Aglets. In this section, features of Epsilon/J are explained by 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 *Role* class is implemented as a subclass of the *Aglets* class that has mobility functions. 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 to 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 another host, acquires new functions that are necessary for behaving as a manager by binding itself to a role instantiated from the *Manager* role.

The *Roaming* environment class

```
public class Roaming extends Environment{
  public void onEnvironmentCreation(String environmentName){}
  public class Visitor extends Role{
    public void onRoleCreation(String roleName){
      // Add this role whose name is specified by "roleName"
      //    to the "Roaming" environment.
      Roaming.this.addRole(roleName, this);
      addBindingInterface("executeTask");
    }
    public void roam(){
      SeqPlanItinerary itinerary = new SeqPlanItinerary(this);
        :
    }
  }
}
```

The *onEnvironmentCreation* is a constructor that is called when an environment is instantiated. The *onRoleCreation* is a constructor that is called when a role is instantiated. These names are based on the name of the Aglet's constructor *onCreation*. The role's name, the parameter of the *onRoleCreation*, is an identifier of a role instance. This name identifies roles instantiated from the same *Visitor* role class. Roles that belong to the same role class can exist in the same environment. Epsilon/J presents directory services based on role names. The environment name in the *onEnvironmentCreation* is treated in the same manner.

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 executes the *executeTask* at each host.

The *ContractNet* environment class

```
public class ContractNet extends Environment{
  public void onEnvironmentCreation(String environmentName){}
  public class Manager extends Role{
    public void onRoleCreation(String roleName){
      ContractNet.this.addRole(roleName, this);
    }
    public void start(){}
    public void bid(Contractor c){}
    public void end(Info info){}
  }
  public class Contractor extends Role{
    public void onRoleCreation(String roleName){
      ContractNet.this.addRole(roleName, this);
      addBindingInterface("executeTask");
    }
    public void taskAnnounce(Manager m){}
    public void award(Manager m){
     m.end(executeTask());
    }
  }
}
```

4.2 Object descriptions

The following is a program that defines the class *UserProxy* and the class *SearchInfo*. An object instantiated from the *UserProxy* class is bound to 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 *onEpsilonObjCreation* is a constructor that is called when an EpsilonObj is instantiated.

The *UserProxy* class

```
public class UserProxy extends EpsilonObj{
  public void onEpsilonObjCreation() {
    // Search a visitor role "visitorRole"
    //   and bind this user proxy to the visitor role.
    // This user proxy becomes an agent that can roam around hosts.
    visitorRole.bind(this, "executeTask", "executeContractNet");
    visitorRole.roam();
   }
  public void executeContractNet() {
    // Search a manager role "managerRole"
    //   and binds this user proxy to the manager role.
    // This user proxy becomes an agent that can act as a manager.
    managerRole.bind(this);  // there are not binding-interfaces.
    managerRole.start();
  }
}
```

The *InfoSearcher* class

```
public class InfoSearcher extends EpsilonObj{
  public void onEpsilonObjCreation() {
    // Search contractor roles existing in the environment "contractnetEnv"
    //   that is instantiated from "ContractNet".
    Contractor [] allContractorRoles
      = contractnetEnv.searchRole("Contractor");
    // Select a role "contractorRole" from "allContractorRoles"
    //   and bind this information searcher to the role.
    // This information searcher becomes an agent
    //   that can act as a contractor in the "contractnetEnv".
    contractorRole.bind(this, "executeTask", "searchInfo");
  }
  public void searchInfo(){...}
  public void search2Info(){...}
}
```

5 Reflection and Epsilon/J implementation

5.1 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 a reflection mechanism, 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[4] and delegated to an object bound to the role. This

[4] In the current implementation, a transformation of a signature is limited to renaming a message name.

approach is similar to the Composition Filters(CF)[1][8] approach. In CF, an object consists of an interface layer that contains input/output message filters and a kernel object. A role in RoleEP corresponds to a message filter in CF. As shown in Figure 3, binding-interfaces can be wrapped with before/after actions that describe synchronization constraints, error checking strategies, security checking strategies, coordination constraints and so forth.

Since mechanisms of binding-interfaces and binding-operations are implemented very simply in Epsilon/J, decline in performance caused by adding RoleEP features of this kind to the original Aglets mobile agent system is slight. In Aglets, moreover, a 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 prompt 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 the above kinds of information. In Epsilon/J, the notion of *messenger role* is introduced to realize role-to-role remote communication and role-to-environment remote communication. A 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. The mechanism of a messenger role may cause some kind of decline in performance.

5.2 Reflection facilities in Epsilon/J

Epsilon/J gives reflection facilities that can introspect environments and roles —for example, a list of environment instances/names, a list of method names in an environment, a list of role instances/names in an environment, a list of method/binding-interface names in a role. Using these facilities, dynamic aspects of programs can be described easily: 1) An object can search an environment that the object wants to adapt to; 2) An object can check if the object is able to bind to a role. The object must have methods corresponding to binding-interfaces of the roles. Environments in RoleEP can be considered as execution environments for objects. Binding-operations and facilities for introspecting environments/roles can be regarded as some kinds of MOPs (Metaobject protocols) that are provided as Epsilon/J class libraries. For example, a binding-operation is implemented in *Role* class and a customized binding-operation such as a binding-operation with security checking can be described by defining a subclass of *Role* class.

6 Discussion

6.1 Merits of RoleEP

RoleEP addresses problems pointed out in section 2 as follows:

- Mobility/collaboration functions can be separated from original functions completely and can be encapsulated within environment descriptions. This solves problems that appeared in Case 1 and 2.
- The problem that descriptions of role behavior depend on interface names of core classes in Case 3 can be solved by the binding-interface mechanism.

Moreover, there are attractive properties as follows:

- Construction mechanism for mobility/collaboration components: RoleEP is beneficial for constructing mobility/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 mobility/collaboration components.
- Evolution mechanism for agents: In RoleEP, an object becomes an agent by binding itself to a role that belongs to an environment. An object can dynamically evolve to an agent that can play multiple roles. Using RoleEP, programs that adapt to external context can be described easily.
- Agentification mechanism: Genesereth and Ketchpel show three approaches for converting objects into agents[12]: 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 object. 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 the dynamic agentification mechanisms. In RoleEP, mobility/collaboration functions needed for mobile agents are described in terms of roles and other functions are described in terms of objects. In order to separate these two kinds of functions completely, roles have facilities as transducers that translate message interactions among roles to message interactions among objects.

6.2 Comparisons with AOP

Table 1, which extends an AOP comparison method proposed in [6], compares RoleEP with AOP. RoleEP emphasizes dynamic aspect syntheses and dynamic evolution. Although AOP and RoleEP have common viewpoints, there is a big difference between them. In AOP, an executable program can be constructed only by objects even if there are no aspects that add cross-cutting properties to

viewpoint	AOP	RoleEP
aspects	aspects	environments and roles
components	components	objects
join points	join points	binding-interfaces
(between aspects and components)		
weaving method	weaver	binding-operation
aspect reuse	emphasized	emphasized
dynamic evolution	not emphasized	emphasized

Table 1. AOP vs RoleEP

objects. On the other hand, objects cannot organize a program without environments in RoleEP.

In RoleEP, the use of binding-operation eliminates the necessity of 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 the ability to prevent code duplication. From the viewpoint of static evolution, *advises* weaving is very useful because it prevents code duplication. From the viewpoint of dynamic evolution, however, *advises* weaving is slightly risky 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 the binding-operation.

6.3 Comparisons with the pluggable composite adapter

Mezini, Seiter and Lieberherr propose a new language construct, called a *pluggable composite adapter*, for expressing component gluing[23]. In the pluggable composite adapter, a component is a set of collaborating classes that defines some functionality. A pluggable composite adapter defines how to dynamically extend a component with a new collaboration. The component is adapted dynamically to play roles in the collaboration without changing the component's classes. In addition, a pluggable composite adapter defines how to glue together two independently developed components C1 and C2, where C2 is an abstract collaboration. The following is the structure of the pluggable composite adapter.

```
adapter A {
  Field_Method_Defs
  Helper_Class_Defs
  { adapter R adapts C1.B [extends C2.S] adaptation_body }*
}
```

C1.B is a class of a component C1. Through *adapts* relation, an instance of *C1.B* acquires a function given in adaptation_body. An instance of *adapter R* is created only when an instance of *C1.B* comes into R's scope. *Adapter A* and *adapter R* correspond to an environment and a role in RoleEP respectively.

Adaptations are similar to binding-operations in RoleEP. Using the pluggable composite adapter, the contract-net protocol can be described as follows.

```
adapter ContractNet {
  adapter Manager adapts UserProxy {
    public void start(){...}
    public void bid()  {...}
    public void end()  {...}
  }
  adapter Contractor adapts InfoSearcher {
    public void taskAnnounce(){...}
    public void award(){ InfoSearcher.this.searchInfo();  }
  }
}
```

Notions of the pluggable composite adapter are quite similar to RoleEP. However, there are some differences between them as follows: 1) A relation between an adapter and an adaptee is described statically in the the pluggable composite adapter; 2) Descriptions of adapter's behavior depend on interface names of adaptee's class in the the pluggable composite adapter.

7 Related works

Bardou shows comparisons between AOP and related approaches, namely Role Modeling[2], Activities and roles[21], Subject-Oriented Programming[14], Split objects[5] and Us "a subjective version of SELF"[26] in [6]. Besides Role modeling, a lot of other research has been done concerning role concepts[19][20][9][27]. Van Hilst and Notkin propose the idea of role components, which are described by C++ templates, to implement collaboration-based design[29]. This approach is similar to the mix-in approach. In *Coordinated Roles* proposed in [24], descriptions of collaborations are separated from descriptions of objects by using role concepts. Although this approach is similar to the binding-interface concepts, there is no concept 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 emphasized. Split objects and Us are based on a delegation mechanism that enables dynamic evolution.

On the other hand, adaptations to external context are studied from the viewpoint of how a single object evolves dynamically—for example, how a person acquires methods and attributes when he/she does a job, marries and so on. In the Subject Oriented Programming, an object acquires new functions by participating in subjects. The concept of subjects is similar to the concept of environments in RoleEP. *Mobile Ambients* is a model that gives a layered agent structure[7]. In this model, agents run on fields constructed by synthesizing contexts (environments) dynamically.

8 Conclusions

Distributed applications that reside in the Internet environment, whose structures change dynamically, are spreading rapidly. Most applications are imple-

mented 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 have to adapt to new environments. New computation paradigms and programming languages are necessary in order to alleviate this requirement. RoleEP that we have proposed in this paper is one approach to address this issue. In this paper, the effectiveness of RoleEP was discussed from the viewpoint of mobile agent applications. However, the notion of RoleEP is quite general and can be applied to other kinds of applications that need mechanisms for *separation of concerns*.

References

1. Akist, M. and Tripathi, A.: Data Abstraction Mechanisms in Sina/ST, *Proceedings of the Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'88)*, pp.265-275, 1988.
2. Andersen, E.P. and Reenskaug, T.: System Design by Composing Structures of Interacting Objects, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, Lecture Notes in Computer Science, Springer, vol.615, pp.133-152, 1992.
3. Aridor, Y. and Lange, D.B.: Agent design patterns: Elements of agent application design, *Proceedings of Agents'98*, 1998.
4. AspectJ. http://aspectj.org/.
5. Bardou, D. and Dony, C.: Split Objects: a Disciplined Use of Delegation within Objects, *Proceedings of the Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'96)*, pp.122-137, 1996.
6. Bardou, D.: Roles, Subjects and Aspects: How do they relate?, *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*, 1998.
7. Cardelli, L. and Gordon, A.D.: Mobile Ambients (Extended Abstract), *the proceedings of the workshop on Higher Order Operational Techniques in Semantics*, 1997.
8. Czarnecki, K. and Eisenecker, U.W.: *Generative Programming*, Addison-Wesley, 2000.
9. Fowler, M.: Dealing with Roles, *Proceedings of the 4th Annual Conference on Pattern Languages of Programs*, 1997.
10. 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.
11. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns*, Addison-Wesley Publishing Company, Inc., 1995.
12. Genesereth, M.R. and Ketchpel, S.P.: *Software Agents*, *Communications of the ACM*, vol.37, No.7, pp.48-53, 1994.
13. Guerraoui, R. et al.: Strategic Directions in Object-Oriented Programming. *ACM Computing Surveys*, Vol.28, No.4, pages 691-700, 1996.
14. 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.
15. IBM: *Aglets Software Development Kit Home Page*, http://www.trl.ibm.co.jp/aglets/index.html, 1999.

16. Kendall, E.A.: Role Model Designs and Implementations with Aspect-oriented Programming, *Proceedings of the Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'99)*, pp.353-369, 1999.
17. Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J.: Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science, Springer, vol.1241, pp.220-242, 1997.
18. Kiniry, J. and Zimmerman, D.: A Hands-On Look at Java Mobile Agents, *IEEE Internet Computing*, vol.1, No.4, 1997.
19. Kristensen, B.B.: Object-oriented Modeling with Roles, *Proceedings of the 2nd International Conference on Object-oriented Information Systems (OOIS'95)*, 1996.
20. 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.
21. Kristensen, B.B. and May, D.C.M.: Activities: Abstractions for Collective Behavior, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'96)*, Lecture Notes in Computer Science, Springer, vol.1098, pp.472-501, 1996.
22. Lange, D. and Oshima M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.
23. Mezini, M., Seiter, L. and Lieberherr, K.: Component Integration with Pluggable Composite Adapters, *Software Architectures and Component Technology: The State of the Art in Research and Practice, Mehmet Aksit, editor*, Kluwer Academic Publishers, 2000.
24. 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.
25. 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.
26. Smith, R.B. and Ungar, D.: Programming as an Experience: The Inspiration for Self, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, Lecture Notes in Computer Science, Springer, vol.952, pp.303-330, 1995.
27. Tamai, T.: Objects and roles: modeling based on the dualistic view, *Information and Software Technology*, Vol. 41, No. 14, pp. 1005–1010, 1999.
28. 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.
29. VanHilst, M. and Notkin, D.: Using Role Components to Implement Collaboration-Based Designs, *Proceedings of the Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'96)*, pp.359-369, 1996.