An Evolutional Cooperative Computation Based on Adaptation to Environment

Naoyasu Ubayashi and Tetsuo Tamai
Graduate School of Arts and Sciences, University of Tokyo
3-8-1 Komaba, Meguro-ku, Tokyo 153-8902, Japan
{uba,tamai}@graco.c.u-tokyo.ac.jp

Abstract

A framework in which a group of objects collaborating with each other evolve their functions dynamically is presented in this paper. We call the framework evolutional cooperative computation and present an environment-adaptive computation model for its foundation. Then, a programming language Epsilon/0, which supports the computation model and has the reflection mechanism, is presented. In this paper, the concept of environments that give objects collaboration fields is introduced. An object evolves itself and changes relations among other objects by adapting itself to environments or seceding from environments.

1. Introduction

A framework in which a group of objects collaborating with each other evolve their functions dynamically is presented in this paper. By collaboration, we mean that a number of objects engaging in their own roles send messages to each other and perform a job that cannot be executed by a single object. By evolution, we mean that an object acquires or discards functions and attributes dynamically. There are a number of researches on evolution targeted on a single object. However, researches that treat evolution in terms of collaboration among objects are few. In this paper, the concept of environments that provide fields for objects collaboration is introduced. An object evolves itself and changes relations with other objects by adapting itself to environments or seceding from environments. At the same time, an environment evolves as objects join into or leave from the environment. We call such a framework *evolutional cooperative computation* and present an environment-adaptive computation model for its foundation. Then, a programming language *Epsilon/0*[1], which supports the computation model and has the reflection mechanism, is presented.

This paper is organized as follows. In section 2, the environment-adaptive computation model is proposed. In section 3, a programming language Epsilon/0 that supports the computation model is introduced. In section 4, several examples described in Epsilon/0 are given. In section 5, methods for software architecture constructions in Epsilon/0 are shown. In section 6, reference to a number of works related to environment adaptation is given. Lastly, in section 7, we conclude this paper.

2. Environment Adaptation Model

2.1. Basic Concepts

In the environment-adaptive computation model, a logical field where a group of objects collaborate with each other is called *environment*, and a set of methods/attributes that each object should have in order to execute functions assigned to it in an environment is called *role*. Procedures for collaborating with other roles are described in role methods. Both environments and roles are instances. Types of environments are called environment classes and composed of following constructs.

- environment methods
- environment attributes
- role class definitions

Aims to introduce the concept of environments are as follows: 1) to give name spaces (fields). An object acquires a name from the space and collaborates with

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

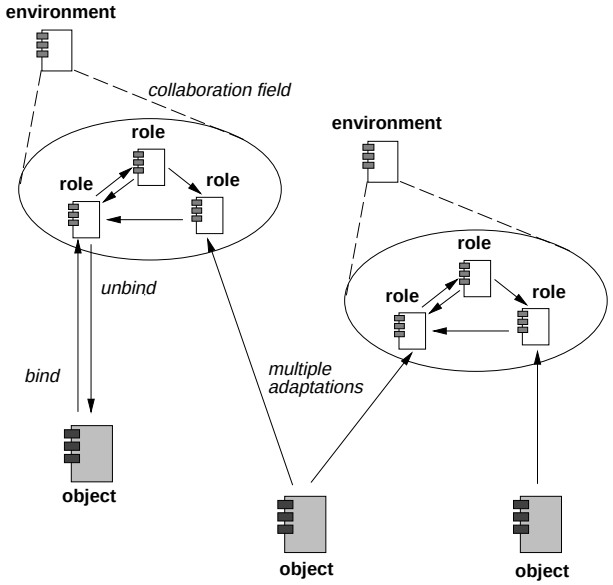

Figure 1. Environment adaptation

objects that have names; 2) to give common methods/attributes, which are called environment methods/attributes, in the field; 3) to coordinate behaviors of objects in the field; 4) to define object behaviors that are available only in the field. Environment methods/attributes are defined in environment classes to realize 1)2)3), and role classes are defined to realize 4). Using the name space facilities, an object can look for other objects the object wants to collaborate with by role (class) names. It is not necessary to use object references that make collaboration descriptions complicated.

An object has to acquire behaviors and naming conventions that are only available in an environment in order to collaborate with other objects in the environment. These behaviors and naming conventions are described in role class definitions that are composed of role methods/attributes.

An object joins into an environment (instance) by combining itself with roles (instances) that exist in the environment. This mechanism is called *adaptation to environment*. An object can join into a number of environments simultaneously. The operation for environment adaptation is called *binding-operation* and the reverse operation is called *unbinding-operation*. With binding-operations, an object can dynamically get role methods/attributes available in an environment and collaborate with other objects that are bound to roles in the same environment (Figure 1). An environment can join into another environment. In this case, the former environment behaves as the representative of roles that belong to the environment. As a result, a number of environments compose a layered structure.

Collaborations are performed by message passing among roles that belong to the same environment. After executing a binding-operation, the semantics of *self* changes. Both *self* appeared in an object and *self* appeared in a role indicate the former. All messages are received by an object. First, the object searches corresponding methods in roles, and after that, searches them in itself. An object can invoke not only its original methods but also role methods by sending messages to self [2].

2.2. Evolutional Mechanisms

The evolutional cooperative computation is autonomous as well as evolutional. Autonomy means that an object has strategies for its actions. Strategies is composed of the following program logics: what kind of environments an object adapts itself to; when an object adapts itself to an environment. Evolution means that an object adapts itself to an environment and acquires and discards its methods/attributes dynamically. An object may go through a number of evolutional processes according to its strategies.

The environment-adaptive computation model uses the reflection mechanism to realize evolution. The mechanism is as follows:

- Applications have models that represent computation structures and behaviors. These representations are called meta information;
- Applications can control their computation structures and behaviors by handling meta information.

In the reflection framework, interactions between the base level (the level to execute applications) and the meta level (the level to control meta information) are described in the same model.

In the environment-adaptive computation model, original object functions are described in the base level and strategies are described in the meta level. Since these descriptions are separated, an object can execute autonomous actions by rewriting meta level strategies.

3. Programming Language Epsilon/0

Epsilon/0 is a programming language that supports the environment-adaptive computation model

[2]When an object invoke role methods, it must specify an environment that it belongs to. A number of roles that have the same method name may exist in several environments.

and the reflection mechanism. This language is implemented using ABCL/R3 [3] that is a reflective concurrent object oriented programming language based on Scheme[5]. The constructs contained in the environment-adaptive computation model - for example, environments, roles, environment adaptation - are implemented by the reflection mechanism of ABCL/R3. Basic language constructs of Epsilon/0 are as follows.

```
;;--------------------------
;; Class
;;--------------------------
;; Class definition
(define-class CLASS-NAME (SUPER-CLASS)
  {SLOT-NAME | (SLOT-NAME EXP)}*)
;; Instantiation
(make CLASS-NAME {:SLOT-NAME EXP}*)
;; Method definition
(define-method CLASS-NAME (METHOD-NAME ARG+) BODY+)
;;--------------------------
;; Environment class
;;--------------------------
(define-context CONTEXT-NAME (SUPER-CLASS)
  {SLOT-NAME | (SLOT-NAME EXP)}*)
(make-context CONTEXT-NAME {:SLOT-NAME EXP}*)
(define-context-method CONTEXT-NAME
  (METHOD-NAME ARG+) BODY+)
;;--------------------------
;; Role class
;;--------------------------
(define-role CONTEXT-NAME ROLE-NAME (SUPER-CLASS)
  {SLOT-NAME | (SLOT-NAME EXP)}*)
(make-role CONTEXT-NAME ROLE-NAME CONTEXT-INSTANCE
  {:SLOT-NAME EXP}*)
(define-role-method CONTEXT-NAME ROLE-NAME
  (METHOD-NAME ARG+) BODY+)
;;-------------------------------------
;; Environment adaptation operations
;;-------------------------------------
;; bind
(bind CONTEXT-INSTANCE
  {CLASS- | CONTEXT-}INSTANCE ROLE-INSTANCE)
;; unbind
(unbind CONTEXT-INSTANCE
  {CLASS- | CONTEXT-}INSTANCE)
;; search combinations between objects and roles
(search-role {CLASS- | CONTEXT-}INSTANCE
   CONTEXT-INSTANCE)
;;-------------------------------------
;; Collaboration (message passing)
;;-------------------------------------
;; method invocation
(METHOD-NAME ARG+)
```

The syntax of classes and objects (instantiated from classes) are the same as ABCL/R3. Programs composed of only classes and objects can be executed by the ABCL/R3 interpretor. Environments, which are called *contexts* in Epsilon/0, are defined by *define-context*. Other language constructs on environments - including attributes (called slots in Epsilon/0), method definitions, instantiations and message passing - have the syntax based on ABCL/R3. The first argument of a method definition is constrained to be the instance that is the target of the method execution. Roles are defined by *define-role*. Like environments, other language constructs on roles have the syntax based on ABCL/R3. The built-in function *bind* is used when an object joins into an environment. On the other hand, the built-in function *unbind* is used when an object secedes from an environment. The built-in function *search-role* is used in order to search a role instance that is bound to an instance specified by the first argument in an environment specified by the second argument.

Considering implementation easiness, *bind/unbind* are defined as built-in functions. But it may be better to define them as standard environment methods. Moreover, it may be better to let programmers customize *bind/unbind* functions and behaviors by using the reflection mechanism that realize open implementation programming styles. If *bind/unbind* can be redefined in meta-objects of environments, their functions and behaviors are customized for target applications. Though *bind* executes a binding-operation unconditionally in the current implementation, it will be possible to check some conditions - including security checking and application specific constraints - by redefining *bind* in the future implementation.

A language processor for Epsilon/0 is implemented by extending ABCL/R3. Epsilon/0 introduces the idea of environments on ABCL/R3. Usually, reflective programming languages have problems on performance grounds. To resolve these problems, ABCL/R3 is implemented efficiently using the partial evaluation technique[3]. So, programs written in Epsilon/0 are also executed efficiently.

4. Example

In this section, an example called contract-net protocol[10] is described in Epsilon/0. This protocol is a collaboration model that divides a problem into several subproblems and assigns them to a number of objects through negotiations. Objects that take part in a contract-net protocol may become a manager that proposes a contract in some situation and may become a contractor that undertakes a contract in another situation. Roles that objects take may change dynamically. For simplicity, a contract-net protocol that contains only one manager is considered in this section (Figure 2).

Environment definition

The environment class *contract-net* and the two role classes *manager* and *contractor* are defined in the program. First, in the *start* method defined in the manager role class[3], the *task-announcement* message is broad-

[3] *Get-role* is one of the built-in functions that support the name space facilities. This function returns a list of roles that exist in an environment specified by the first argument and have a role class name specified by the second argument. *Context-of* returns an environment that a role specified by the argument belongs to. Details of these built-in functions will be explained in section 6.

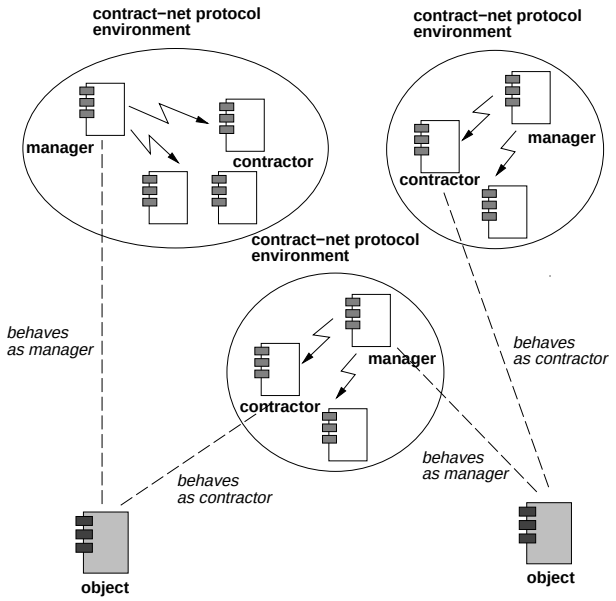

Figure 2. Contract-net protocol

casted to all contractor roles that belong to the same environment. Each contractor role that receives the *task-announcement* message compares his or her own condition with a condition shown by the manager role. If the latter condition satisfies the former condition, the contractor role sends the *bid* message to the manager role. The manager role selects one of the contractor roles that show satisfactory bid-conditions, and sends the *award* message to the selected contractor role. The contractor role invokes the *execute-task* method and executes the task requested by the manager role. The *execute-task* method is not defined as a role method. Thus, a method search is executed to ask the object bound to the role whether it has the specified method definition or not. If the *execute-task* method is defined in the object, this method is invoked. Otherwise, this program execution fails.

As this example shows, Epsilon/0 makes it possible to describe collaboration of the contract-net protocol explicitly and independent from runtime objects.

```
;; Environment contract-net
(define-context contract-net () )
;; Role manager
(define-role contract-net manager ()
   condition-shown-by-manager)
(define-role-method contract-net manager
  (start self)
  (let ((contractor-list
         (get-role (context-of self) 'contractor)))
   ... broadcasts a task-announcement message
       to all roles that are contained
       in the contractor-list.
  ))
(define-role-method contract-net manager
  (bid self a-contractor
       a-condition-shown-by-contractor)
   ... stores a bid-information
       into internal memories.
  (if (bidding is finished)
   ... selects a contractor role
       that shows the best bid-condition,
       and sends the award message
       to the selected contractor role.
   ... if there are not contractor roles
       that satisfy the award-condition,
       this negotiation fails.
  ))
;; Role contractor
(define-role contract-net contractor ()
  condition-shown-by-contractor)
(define-role-method contract-net contractor
  (task-announcement self
                    a-condition-shown-by-manager)
  (if (a-condition-shown-by-manager
           satisfies condition-shown-by-contractor)
      (let ((a-manager
             (car (get-role (context-of self)
              'manager))))
       ... sends a bid message to a-manager.
  )))
(define-role-method contract-net contractor
  (award self)
  (execute-task self))
```

Execution

The following program describes an execution of the contract-net protocol that follows evolutional process. In this program, an object that has the name *john* searches through a number of contract-net protocol environments and joins into some environments that satisfy his requirements. *John* has a job, which is composed of several sub-tasks, and owns funds to execute the job. When *john* does not have sufficient funds to execute a sub-task, he becomes a contractor to earn money. At this time, he collaborates with a manager. On the other hand, he becomes a manager to assign sub-tasks to contractors. At this time, he collaborates with contractors. This program terminates when *john* performs all sub-tasks. Figure 3 illustrates john's execution process.

```
;; class definition (base level)
(define-class person () name money)
(define-method person (execute-task self) ...)
(define-method person (life self job)
 (meta-life (meta-of self) job))
;; adaptation strategy (meta level)
(meta
 (define-class john-meta (metaobject))
 (define (make-john-meta class slots
                          evaluator options)
   (make john-meta :class class
                   :slots slots
                   :evaluator evaluator
                   :lock (make-lock)))
 (define-method john-meta (meta-life self job)
   (future-to-base
    ... John divides a job into a number of
        sub-tasks.
    ... He repeats the following activities
        until he performs all sub-tasks.
    (let ((env (get-all-context)))
```

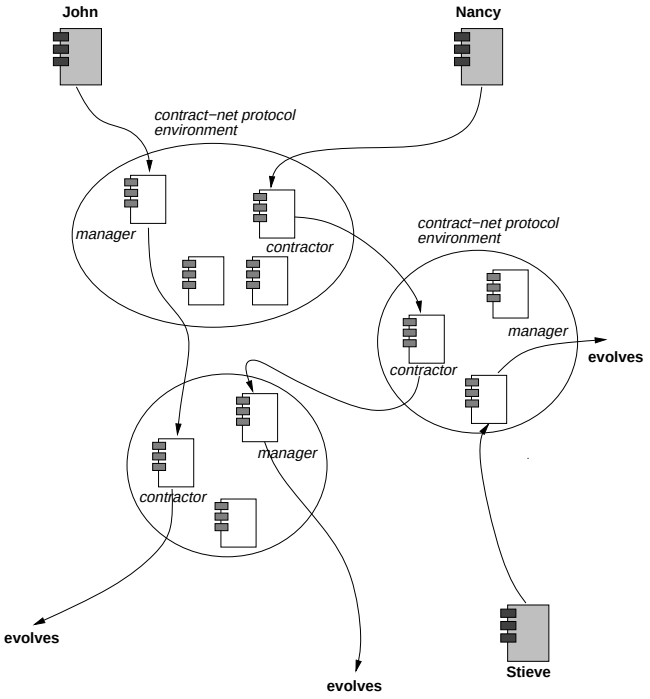

Figure 3. Contract net protocol that has evolutional processes

```
      (begin
       ... John searches through contract-net
           environments where he can join
           as a manager.
           If there are a number of environments,
           he selects the most fitting environment
           that satisfies his requirements.
       ... John joins into a contract-net environment
           that he selects and bind himself
           with a manager role.
        (bind selected-contract-net
              (den-of self) selected-manager)
        (start (search-role (den-of self)
                selected-contract-net))
       ... If John does not have sufficient funds
           and fails to make a contract,
           he searches through contract-net
           environments where he can join
           as a contractor.
        (bind selected-contract-net
             (den-of self) selected-contractor)
       ... If John succeeds in making this contract,
           he can increase his funds.
))))
;; start
(define john
    (make person
       :name "John Smith"
       :metaobject-creator make-john-meta))
(life john a-job)
```

The expression *(meta ...)* indicates that it is evaluated at the meta level. The class definition *john-meta* inherits the default meta object *metaobject*, and it is defined as a meta object. *Make-john-meta* is a function to create a meta object. This function is specified as an initial value for *metaobject-creator* slot when the object *john* is created. By this specification, a meta object of *john* is created.

The *life* method is defined in the *person* class. Actual contents of *life* are described in the *meta-life* method contained in the meta object. The *life* method merely calls the *meta-life*[4]. In the case of the object *john*, the *meta-life* method is defined in the meta object *john-meta*. The expression *(future-to-base ...)* is used in order to invoke a method contained in the base level object from the meta level program.

The *life* method can be regarded as a design pattern to describe autonomy of an object. By defining the *life* method in a meta object, an object can refer its meta information and act according to its computation states. For example, an object can dynamically adapt itself to appropriate environments by referring its loads.

Although a simple contract-net protocol environment that contains only one manager is described in this section, a general contract-net protocol environment that contains multiple managers and contractors can be described by combining these simple contract-net protocol environments. The outline is as follows: 1) create a set of simple contract-net protocol environments; 2) let a number of objects join into these environments. If an object joins into multiple environments as a contractor, it can make contracts with managers who belong to individual environments.

In the current programming paradigms such as the object oriented technology, it is believed that interfaces of program modules must be separated from implementations. It is not necessary for module users to know how they are implemented. Users only have to obey interfaces. Users are not influenced even when implementations are changed.

Such module encapsulation has advantages like program readability, maintenability and reusability. But, this type of encapsulation also has a problem of inflexibility in module usage. If module users can change a part of a program module implementation, they may be able to reuse the module. Unfortunately, it is not permitted in the current programming paradigms. Recently, the idea called open implementation, a programming paradigm that opens a part of program module implementations and permits module users to customize the part, is proposed in order to resolve this problem[1]. Open implementations usually need the reflection mechanism. There are a number of reflec-

[4]The *meta-of* function returns a meta object corresponding to the base level object specified as a parameter. On the other hand, the *den-of* function returns an object corresponding to a meta object specified as a parameter.

tive object oriented programming languages that permit programmers to customize basic language mechanisms such as message reception mechanism. Using the reflection mechanism, applications can be implemented such that meta information including load of message receptions and message logging information are preserved.

Collaboration protocols can be defined as patterns by the evolutional cooperative computation. Moreover, introducing the idea of open implementation, collaboration protocols can be defined as flexible collaboration parts. Meta patterns that let methods create design patterns are proposed by Pree[13]. Target problem domains are divided into invariant parts and variant parts by using meta patterns. Variant parts are picked up as hot spots. Methods to pick up hot spots, which will be exposed as reflective points, are important when parts based on open implementation are defined. In the evolutional cooperative computation, autonomy of an object can be described in the *life* method that is defined in the meta object and may be customized by programmers. This mechanism can be regarded as a kind of open implementation.

5. Constructing Cooperative Components

Epsilon/0 can abstract collaboration among objects as a global module that describes a software architecture. Software architectures can be mapped to Epsilon/0 language structures. Software architectures that are reused many times can be regarded as cooperative components. It is difficult to make reusable collaborative components by using traditional programming languages such as C++, because it is difficult to abstract global runtime structures as follows:

- Number, kinds and names of objects that compose a collaboration;
- Relations and network topologies among objects.

These structures may change dynamically whenever programs are executed. In order to define a collaboration as a reusable component, it is necessary to present abstraction mechanisms that encapsulate these runtime structures within language constructs. Epsilon/0 presents such runtime information as meta level information of environment constructs. Roles can introspect the meta level information within language frameworks by using the reflection mechanism. The runtime structures can be encapsulated within an environment. An environment can be regarded as a collaborative component. The Epsilon/0 presents a set of built-in functions as follows:

get-all-context: returns a list that contains all environments in a system;

get-all-contextname: returns a list that contains names of all environments in a system;

context-of: returns an environment where a role specified by an argument exists;

get-role: returns a list that contains roles that have a name specified by an argument;

get-all-role: returns a list that contains all roles that belong to an environment specified by an argument;

get-all-rolename: returns a list that contains name of all roles that belong to an environment specified by an argument;

bound?: judges whether a role specified by an argument is bound to another object or not.

5.1. Example

A collaborative component called *publish-subscribe* pattern, which is often used in software agent systems, is described as an example. This pattern is composed of a facilitator, subscribers and a publisher (Figure 4). Subscribers are objects that want to subscribe information and a publisher is an object that present information. A facilitator is an object that mediates between subscribers and a publisher. If a publisher presents information, a facilitator forwards the information to subscribers that need it. Using the built-in function *get-role*, a facilitator can be referenced by the role name of the environment. It is not necessary to use variables in order to manage role instances. Using built-in functions that support reflective facilities, references of role instances are not necessary to describe a collaboration.

```
;; Environment: publish-subscribe
(define-context publish-subscribe () )
;; Role: subscriber
(define-role publish-subscribe subscriber ())
(define-role-method publish-subscribe subscriber
  (start self)
```

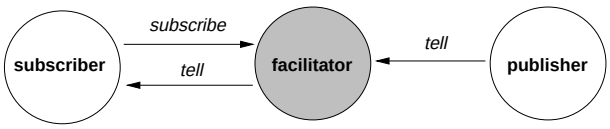

Figure 4. Publish-subscribe pattern

```
  (subscribe
    (car (get-role (context-of self) 'facilitator))
    self category))
(define-role-method publish-subscribe subscriber
  (tell self an-info) ...)
;; Role: publisher
(define-role publish-subscribe publisher ())
(define-role-method publish-subscribe publisher
  (publish self)
  (tell
    (car (get-role (context-of self) 'facilitator))
    category info))
;; Role facilitator
(define-role publish-subscribe facilitator ())
(define-role-method publish-subscribe facilitator
  (subscribe self a-subscriber a-category)
   ... 'a-subscriber' and 'a-category'
        are stored to a table
        that manages information forwarding.
 )
(define-role-method publish-subscribe facilitator
  (tell self a-category an-info)
   ... information is forwarded
       to subscribers that want information
       corresponding to 'a-category'.
 )
```

If an object wants to request some object to mediate something, the object may search an environment that can mediate it. A facilitator role exists in the environment at any time, and objects that join into the environment may merely make a request to the facilitaor role.

A facilitator is defined as a role in the above program. Environments are fields that mediate collaborations among objects. So, a facilitator can be defined as an environment as follows. This program is simpler than the above program.

```
;; Environment: facilitator
(define-context facilitator ())
(define-context-method facilitator
  (subscribe self a-subscriber a-category)
   ... 'a-subscriber' and 'a-category'
        are stored to a table
        that manages information forwarding.
 )
(define-context-method facilitator
  (tell self a-category an-info)
   ... information is forwarded
       to subscribers that want information
       corresponding to 'a-category'.
 )
;; Role: subscriber
(define-role facilitator subscriber ())
(define-role-method facilitator subscriber
  (start self)
  (subscribe (context-of self) self category))
(define-role-method facilitator subscriber
  (tell self an-info) ...)
;; Role: publisher
(define-role facilitator publisher ())
(define-role-method facilitator publisher
  (publish self)
  (tell (context-of self) category info))
```

Mediations - including exclusive controls, resource management and scheduling - can be described by using environments. The monitor concept can be described as an environment too. It is important to judge whether an element that is at the center of the collaboration should be defined as a role or as an environment. An element may be defined as an environment if there is only one object that mediates a collaboration. On the other hand, an element may be defined as a role if there are a number of objects that are at the center of the collaboration. In such a case, an environment may be defined as a common space such as the blackboard model[9].

Using the concept of environments that have the reflection mechanism, many kinds of software architectures - including resource allocations, load balancing and exclusive controls - can be described as language level components.

5.2. Adaptable AOP

Modules (or collaborative components, software architectures) in Epsilon/0 are constructed by the method such that a system is divided into a number of environments and a program is described per environment. This idea is similar to the concept of AOP(Aspect Oriented Programming)[2][11]. In AOP, a system is divided into a number of aspects and a program is described per aspect. A compiler, called *weaver*, weaves aspects together into a system. For example, a complex distributed system is described by a number of aspects - a main aspect that is programmed by an object oriented programming language, communication aspects that are programmed by a special language, failure handling aspects that are programmed by an other special language. These aspects are automatically woven into a single program. AOP resolve the problem that it is difficult to define a function as a global module that is implemented by a number of distributed objects. The idea of aspects corresponds to the idea of environments. Though AOP mechanism is different from the evolutional cooperative computation, the concept *programming by aspects (or environments)* is common.

In 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, in Epsilon/0, aspects (environments) can be defined dynamically and compositions of aspects can be re-arranged dynamically. It can be regarded that Epsilon/0 realizes the *Adaptable AOP*.

6. Related Works

Researches on collaborations have been considered different from researches on adaptation to environment. Methods to describe collaborations are mainly

studied in software engineering, especially in object oriented analysis and design. For example, in UML (Unified Modeling Language)[7], the standard object oriented description notation, collaborations among objects are described by sequence diagrams. Sequential diagrams describe interactions by temporal message tracings among objects. M.VanHilst and D.Notkin propose an idea of role components, which are described by C++ templates, to implement collaboration-based design[8]. Usually, collaboration structures described by these methods cannot be changed dynamically. Moreover, objects that participate in collaborations cannot acquire new functions dynamically.

On the other hand, environment adaptation is 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[12], an object acquires new functions by participating in subjects. The concept of subjects are similar to the concept of environments in the evolutional cooperative computation. In many researches, evolution of static objects are mainly considered, and dynamic interactions among objects are not so emphasized. Researches on mobile agents also treat aspects of environment adaptation. A mobile agent moves on networks from one computer to another in order to perform tasks as a proxy of a user. An agent moves to remote places and uses functions that objects located in the places provide. Though the concept of environments does not exist clearly in mobile agents, the capability explained above is a kind of environment adaptation. Recently, a number of mobile agent systems and languages are proposed including Telescript[6] and Aglets[4]. As other approaches, there are researches on reflective operating systems such as Aperios (aka. Apertos or Muse)[14]. In Aperios, the concept of meta-spaces composed of a group of meta objects that offer some kinds of services to objects is introduced. An object can use many services by moving among meta-spaces.

Actually, researches on collaborations must be closely related to those of environment adaptation. When objects collaborate with each other, each object must perform its own roles. An object may dynamically change its roles or acquire new roles. In such a case, an object has to evolve itself. An object evolves itself in order to collaborate with other objects by using functions that it acquires. It is meaningless to evolve without collaborations. As mentioned here, researches, which are concerning to only collaborations or environment adaptations, already exist. But, researches combining the two approaches are few.

7. Conclusions

Distributed applications that reside in mobile/internet/intranet 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 resolve such a problem. The evolutional cooperative computation that we have proposed in this paper is one approach to resolve such a problem.

References

[1] G.Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[2] G.Kiczales, et al. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag LNCS 1241*, 1997.

[3] H.Masuhara, S.Matsuoka and A.Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. *In Reflection'96 Conference, San Francisco, California*, April 1996.

[4] IBM. http://www.trl.ibm.co.jp/aglets/index.html. 1999.

[5] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. 1990.

[6] Jeffrey M.Bradshaw. *Software Agents*. The MIT Press, 1997.

[7] M.Fowler and K.Scott. *UML Distilled*. Addison-Wesley Publishing Company, Inc., 1997.

[8] M.VanHilst and D.Notkin. Using role components to implement collaboration-based designs. *Proceedings of OOPSLA'96 Conference*, pages 359–369, 1996.

[9] R.Engelmore, et at. *Blackboard System*. Addison Wesley, 1988.

[10] R.G.Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. on Computers*, 29(12):1104–1113, 1980.

[11] R.Guerraoui, et al. Strategic directions in object-oriented programming. *ACM Computing Surveys*, 28(4):691–700, 1996.

[12] W.Harrison and H.Ossher. Subject-oriented programming. *Proceedings of OOPSLA'93 Conference*, pages 411–428, 1993.

[13] W.Pree. *Design Patterns for Object-Oriented Software Development*. the ACM Press, 1995.

[14] Y.Yokote. The apertos reflective operating system: The concept and its implementation. *Proceedings of OOPSLA'92 Conference*, pages 414–434, 1992.