

An Adaptive Object Model with Dynamic Role Binding

Tetsuo Tamai
The University of Tokyo
Tokyo, Japan
tamai@acm.org

Naoyasu Ubayashi
Kyushu Institute of Technology
Fukuoka, Japan
ubayashi@acm.org

Ryoichi Ichiyama
The University of Tokyo
Tokyo, Japan
ir@bellbind.net

Abstract

To achieve the goal of realizing object adaptation to environments, a new role-based model Epsilon and a language EpsilonJ is proposed. In Epsilon, an environment is defined as a field of collaboration between roles and an object adapts to the environment assuming one of the roles. Objects can freely enter or leave environments and belong to multiple environments at a time so that dynamic adaptation or evolution of objects is realized. Environments and roles are the first class constructs at runtime as well as at model description time so that separation of concerns is not only materialized as a static structure but also observed as behaviors. Environments encapsulating collaboration are independent reuse components to be deployed separately from objects. In this paper, the Epsilon model and the language are explained with some examples. The effectiveness of the model is illustrated by a case study on the problem of integrated systems. Implementation of the language is also reported.

1. Introduction

Objects represent things or concepts of the real world and it is this representation feature that gives the object-oriented technology the high modelling capability. Objects in the real world exist in various environments. If an object permanently resides in a fixed environment, the structure and behavior of the object can possibly stay unchanged over time. However, environments surrounding objects may not be stable due to various reasons. If objects are humans or manufacturing equipment, their environment changes periodically between the day and the night and between the weekdays and the weekend. When an object moves, the surrounding environment naturally changes. Even if an object stays at the same place for a certain period of time, the environment itself may dynamically change. Corresponding to such environmental change, objects adaptively change themselves. Conversely, objects may spontaneously evolve,

causing change in their relation to the environment and that in turn may trigger change in the environment. Moreover, there generally exist multiple environments around an object and the object may selectively belong to a subset of them at a time and the selection of environments may also change dynamically.

How is such adaptation or evolution of objects handled in the world of object-oriented modelling and programming? As many researchers have pointed out, current widely-used object-oriented modelling and programming languages do not conveniently support such flexibility. Motivation of our research is to build a computational model that is flexible enough to cope with future changes but simple enough to describe and reason about the design validity.

To explain motivation for our work more concretely, we give some typical examples introduced by other researchers to illustrate their work. We share similar objectives and the difference of approaches will become clearer by handling the same problems.

Y. Honda et al. [9] gave an example of adaptation. A woman Hanako, modelled as an object, marries with Taro and adapts to the environment *family*. She then gets employed as a researcher by a research laboratory and adapts to the environment *laboratory*. The adaptation should be made dynamically, thus it can be regarded as a kind of evolution. At the same time, the object Hanako should preserve its identity when she enters a new environment like the lab or even after she quits the lab for some reason.

In Honda et al.'s model *Morphe*, suppose an object (e.g. Hanako) enters an environment (e.g. a laboratory) and assumes a role (e.g. a researcher), then the object acquires a new set of attributes and behaviors or alters some of the attributes and behaviors already possessed by the object through following transformation rules associated with the role. This strategy of employing transformation rules must have been adopted mainly because the underlying language of their work was a constraint based object-oriented language. The problem is that once an object undergoes transformation, it is often difficult to reverse it and thus the role discarding transformation is hard to implement.

M. Fowler [4] gave an example of personnel roles in a company to be assumed by employees. He listed up engineers, salesmen, directors and accountants as roles and put a question how to deal with situations such that a person plays more than one role or a person changes his or her role in the lifetime. He showed several patterns that solve this problem and gave a generic name *role pattern*. Those patterns, however, employ very clever and ad hoc techniques, revealing the difficulties of describing such situations naturally in the conventional object-oriented framework.

E. Kendall [13] gave an example of the bureaucracy pattern. There are five roles in the pattern: Director, Manager, Subordinate, Clerk and Client. A client deals with a clerk. Manager and Subordinate are subclasses of Clerk. A manager supervises subordinates and reports to a director. There exist two environments: a bureaucracy of a sales company and a trading relation between clients and clerks. A clerk or a manager may belong to both environments. Kendall treats this situation using AspectJ, indicating the relation between the role based approach and the aspect-oriented technology.

Our motivation and these examples suggest a role model where an environment is defined as a field of collaboration between roles and an object adapts to the environment assuming one of the roles. There have been proposed a number of role models but our model to be described in the succeeding sections has the following features.

1. Objects can freely enter or leave environments and belong to multiple environments at a time so that dynamic adaptation or evolution of objects is realized.
2. Environments as fields of collaborations as well as roles are the first class constructs at runtime as well as at model description time so that separation of concerns is not only materialized as a static structure but also observed as behaviors.
3. Environments encapsulating collaboration are independent reuse components to be deployed separately from objects that participate in them.

In this paper, we introduce our role model *Epsilon* and a language based on the model *EpsilonJ* with examples. We also explain ways of language implementation.

2. Role Model

2.1. Collaboration and Role Model

The history of object-oriented technology is abundant with role models [30]. The major objective of considering roles has been to describe collaboration of objects and identify clear and solid boundary of each object. An object may take part in multiple collaborations assuming different

roles in different collaborations. Thus, the characteristics of an object may be clarified by consolidating roles the object plays in multiple collaborations.

A typical way of describing a collaboration is by specifying use cases or behavioral scenarios as observable behaviors of the collaboration. Originally advocated by I. Jacobson [11] as a method OOSE and inherited by the Unified Modeling Process [10], the use case approach is now well practiced. A similar approach was taken by Wirfs-Brock et al. [37], where responsibilities to be taken by objects for achieving various types of collaborations were given much attention and a kind of Class-Responsibilities-Collaboration Cards [3] was adopted to describe these relations. In these methodologies, the word *role* is not necessarily used. In UML, “roles” had been historically used to denote directions of static associations (AssociationRole) and relating the term to collaboration (CollaborationRole) was relatively new. Whatever the name, “role” or corresponding concept in these methods is captured as an aspect of objects engaged in collaboration. Roles are considered for listing up functions or behaviors of an object to define a clear boundary of the object, thus their granularity is smaller than objects and conceptually comparable to the level of methods.

In some other OO development methodologies, the concept of roles is given a higher position so that the term *role modelling* is created and extensively used. A typical example is the *OOram* methodology [21], which not only defines role models but also integrates them with OO models through the step of role model synthesis.

In these methodologies, roles play an important part in the phases of analysis and design but usually become invisible in the implementation phase. However, there are some work that aim at preserving roles explicitly in programs. For example, VanHilst & Notkin [35, 34] used class templates of C++ to implement roles. Smaragdakis & Batory [25] introduced a construct of *mixin layers* where collaboration fields are described as layers composed of roles, and roles are filled by objects a la mixin style. Subject-oriented programming [7] and its successor, Multi-Dimensional Separation of Concerns (MDSOC) [19], has a longer history but the idea of describing collaboration fields in separate dimensions and defining classes by consolidating roles in those dimensions is similar. All these approaches are class based and composition of objects consolidating roles is done statically.

2.2. Epsilon Model

Our aim is to support description of collaboration not just at the model level but also at the programming level. Collaboration model is built not for identifying objects but to manipulate collaboration environments and their roles directly and reuse them as program components. Up to that

point, we share the same objective as VanHilst & Notkin, Smaragdakis & Batory, or Hyper/J, a language for MDSOC.

However, as we stated in the previous section, the major motivation for our research is to devise a mechanism for object adaptation to environments. An environment in the context of role model is regarded as a collaboration field and in order to realize adaptation, objects should be allowed to enter collaboration environments by assuming roles and to leave from environments by discarding roles dynamically. At this point, our approach parts from the above other methods.

The basic design principles of our model *Epsilon* can be summarized as follows.

Support adaptive evolution In our model, objects evolve by participating in environments and assuming their roles. Participation can be made dynamically and leaving the environment is also allowed dynamically. An object is free to belong to multiple environments at a time.

Describe separation of concerns Each environment represents a concern so that separation of concerns is explicitly supported by the model. Interactions between concerns are realized through objects simultaneously assuming roles of different collaboration environments.

Advance reuse Besides objects, environments including roles can be units of reuse. Moreover, since environments and roles are given the status of first class constructs in a proposed programming language, collaboration patterns can be reused directly as programming level components.

2.3. Language

Our language named *EpsilonJ* has the following constructs to support the above mentioned model features. *EpsilonJ* is an extension of Java, basically following the Java syntax.

2.3.1 Declaration of environments and roles

In *EpsilonJ*, environments are called “contexts”. `Context` and `role` are declared with attributes and methods just like object classes. Declaration of `role` is placed inside of `context` declaration, similar to inner classes of Java but the coupling between a context and its roles is stronger than that of an outer class and inner classes as we will see later. Instances of contexts and roles are created dynamically.

2.3.2 Encapsulation of roles in environments

As declaration of roles is confined in a context, their interaction is encapsulated within the context. Roles in a context

can communicate with each other but cannot access to other contexts and roles in other contexts directly. Collaboration is naturally described on the role instance basis.

Following is an example program to show how `context` and `role` are declared and collaboration between roles are described.

```
context Company {
    static role Employer {
        int salary = 100;
        void pay() {
            Employee.getPaid(salary);
        }
    }
    role Employee {
        int save;
        void getPaid(int salary) {
            save += salary;
        }
    }
}
```

When the qualifier “static” is declared in a role definition, there is exactly one instance of that role in a context instance and it is created at the time of the context instance creation. Note that this semantics of “static” is different from that of the Java nested classes. In Java, a static class declared in a class is not an inner class; it has no current instance of the enclosing class. On the other hand, a “static” role in a context is associated with its enclosing context instance. It only means the role instance is a singleton in the context. The singleton role instance can be referred by the role name within the context and by the role name qualified with the context instance reference outside of the context.

For example, after a context instance is created as:

```
Context c = new Company();
```

the role instance of `Employer` can be referred by `c.Employer`.

2.3.3 Role instance creation

When a role is not declared “static” its role instances can be created by an indefinite number, using the keyword “new” and a constructor. A role instance is necessarily associated with the enclosing context instance and thus the constructor should be qualified by a context instance reference, not by the context type. Continuing the above example, `new c.Employee()` will create a new role instance of `Employee`.

Here, a role is regarded as a class or a role instance factory but even in this case, the role instances can be referred collectively by the role name, possibly qualified with the context instance reference. When a method of a role that has multiple instances is called by referring just its role name, the method is invoked for all the role instances in nondeterministic order. Thus, the method call `Employee.getPaid(salary);` in the method declaration of `pay()` in `Employer` role is interpreted as calling methods `getPaid` of all the `Employee` instances.

2.3.4 Binding of objects with roles

An object can be dynamically bound to a role of a context and can be unbound later. An object may be bound to multiple roles of different contexts. When an object is bound to a role, it acquires the functions of the role, i.e. it can call the role's methods as the following example shows.

```
class Person {
    int money;
}
Person tanaka = new Person();
Person sasaki = new Person();
Company todai = new Company();
todai.Employer.bind(sasaki);
todai.Employee.newBind(tanaka);
sasaki.(todai.Employer).pay();
tanaka.(todai.Employee).getPaid();
```

A method `bind(Object o)` is defined to all roles, meaning to bind the Object `o` to the role instance whose `bind` method is being invoked. A method `newBind(Object o)` is defined to all non-static roles, meaning to create a new role instance and bind the Object `o` to it. The `newBind` method is actually a two step process: `todai.Employee.newBind(suzuki)` is equivalent to `(new todai.Employee()).bind(suzuki)`.

After the binding, the object bound to a role acquires an access to the role instance and thus can use the role methods as shown in the above program piece. Conversely, role methods cannot be accessed unless the role instance is bound to an object. This mechanism of role method access through the binding to an object can be regarded as a kind of delegation.

Our design choice of defining the binding operation between an object and a role instance, not between an object class and a role class, is intentional and has rationale. We could have characterized a role as a slot or a template where a binding object is inserted. We did not do so, because in some cases, it would be useful for a role instance to retain its own state after detaching from the binding object. For example, suppose a person object Tanaka took a role of the account department head at the company Todai but then the role was replaced by Suzuki. It would be appropriate that the role instance of the account head still retains the state of the work left unfinished by Tanaka and lets Suzuki succeed it.

Instead of `sasaki.(todai.Employer).pay()`, one may want to write just:

```
sasaki.pay();
```

but it is not allowed for the following two reasons.

1. Since an object can be bound to multiple role instances, the above expression can be ambiguous.
2. By explicitly indicating the bound role, static type checking is possible¹.

A method `unbind()` is defined to all roles. This method can be applied to a role instance or a static role. When the role is bound

¹However, as binding and unbinding are dynamic operations, whether the object is really bound to the designated role so that the method can be found without failure should be checked dynamically.

to an object, its binding is dissolved and the reference to the role instance is returned. When the role is not bound to an object, its effect is no operation.

2.3.5 Required interface

If binding an object with a role just brings about disjoint union of the methods in the object and the role, nothing particularly interesting will happen. There should be some interaction between the object and the role that are bound together so that the state and the behavior of the object should be affected by the binding.

For that purpose, there is a way of defining an interface to a role and it is used at the time of binding with an object, requiring the object to supply that interface, i.e. the binding object should possess all the methods specified in the interface. A required interface can be declared using the `requires` phrase as follows.

```
interface Deposit {
    void deposit(int);
}
context Company {
    role Employee requires Deposit {
        void getSalary(int salary) {
            deposit(salary);
        }
        ...
    }
}
```

To reduce a plethora of names, there is an anonymous required interface expression as follows.

```
role Employee requires
    {void deposit(int);} {
    void getSalary(int salary) {
        deposit(salary);
    }
    ...
}
```

2.3.6 Method import

When a required interface is declared to a role, methods can be imported to the role from the binding object. For example, suppose the class `Person` has a method `deposit` such as:

```
class Person {
    string name; int money;
    void deposit(int s) {
        money+=s;
    }
}
```

and the variable `tanaka` has a reference to its instance. Using the binding operation:

```
todai.Employee.newBind(tanaka)
```

the method `deposit(int)` of `tanaka` is imported to the `Employee` role instance through the interface. The binding object class may explicitly implement the interface like:

```
class Person implements Deposit {
    string name; int money;
    void deposit(int s) {
        money+=s;
    }
}
```

but it is not mandatory. It is only necessary to have a method that has the same name and the same signature required by the role. After the binding, whenever the method `deposit(int)` of the role instance is called, the corresponding method of `tanaka` is invoked.

The binding object may even have a method with a different name but the same signature as the required method. In that case, binding with the `replacing` phrase is used to specify the correspondence. For example, suppose the class `Person` is defined as:

```
class Person {
    string name; int money;
    void save(int s) {
        money+=s;
    }
}
```

Then, the binding operation should be given by:

```
today.Employee.newBind(tanaka)
    replacing deposit(int) with save(int);
```

After this binding, whenever the method `deposit(int)` of the role instance is called, the method `save(int)` of `tanaka` is invoked instead.

In general, when a role has a required interface declaration, every interface method should be explicitly replaced at the time of binding by a binding object method, except when the object possesses a method with the same name and the same signature.

2.3.7 Method export

All public methods declared in `role`'s are "exported" in a sense that they can be used from the binding object. But here, we focus on the case where an interface method is overridden in the role body. For example,

```
context Company { ...
    role Employee requires
        {void deposit(int);} {
        void deposit(int salary) {
            ...
        }}
}
```

In this case, when the `Person` object referred by the variable `tanaka` is bound to `Employee` role as before:

```
today.Employee.newBind(tanaka)
    replacing deposit(int) with save(int);
```

thereafter whenever the method `save` of `tanaka` is called, the overriding role method `deposit` is invoked instead. This can be regarded as method export from the role to the binding object.

2.3.8 Method import/export

When an interface method is overridden by the corresponding role method, the replacing method of the binding object becomes hidden. If there is a need for invoking the hidden method in the context, either in the body of the overriding method or in other role (or context) methods, it is possible to invoke it by attaching the qualifier `super` to the method name. For example,

```
context Company {...
    role Employee requires
        {void deposit(int);} {
        void deposit(int salary) {
            ...
            super.deposit(salary);
            ...
        }}
}
```

2.3.9 Multiple method replacement

As an object may bind to multiple role instances, the same object method may replace multiple role methods. Such a situation may appear to cause much complexity but it can be comprehended systematically. We divide the situation into three cases. As binding and unbinding are dynamic operations, the situation may change from one case to other dynamically.

Case 1: replaced methods are all imported (not overridden)

This is the simplest case. Each call of the role interface method will actually call the replacing object method.

Case 2: replaced methods are all exported (overridden)

When the replacing object method is called, all the overriding methods are called. The order of invocation is compiler dependent. The order of binding is one of the natural orders but it is not stipulated, allowing the possibility of concurrent invocation. When the method has a return value, the one from the last invocation will be returned, which actually means which one will be returned is not determined. Note that when one of the overriding methods (role methods) is called, the effect is the same, i.e. all the other overriding methods that share the same replacing method are called, because the effect of overriding is that the invocation of the overriding method is equivalent to the invocation of the replacing object method.

Case 3: some replaced methods are imported, others are exported

Call to the replacing object method that is overridden (and equivalently, call to the overriding role method) will result in the same behavior as stipulated in Case 2.

Call to an interface method in the role with "super" qualifier will always result in calling the original replacing method of the binding object however it is overridden. When a call to an interface method is not qualified with "super", it calls the current overridden object method, the effect of which is the same as Case 2, i.e. when more than one role methods are overriding it, all of them will be called.

This case may look complicated but the principle is very simple. The binding/unbinding mechanism is dynamic in nature and the current status of binding is always respected, except the explicit call of the original method with "super".

3. Case Study

It is straightforward to write the three examples introduced in the introduction with our Epsilon model as they are analogous to the example of the `Company` context explained above.

3.1. Integrated System

Here, we take the problem of integrated systems for a case study. An integrated system is a system integrating independent but related components. When a component takes an action, related components behave accordingly. For example, a collected system of an editor, a compiler, and a debugger is a typical integrated system. When the compiler detects a syntax error or the debugger stops at a breakpoint, the editor scrolls to the corresponding source statement.

A simplified model of integrated systems was introduced by K. Sullivan et al.[28]. In this model, the components subject to integration are objects that have just a binary state, “on” and “off.” We call these objects Bits. An instance of Bit has operations “set” and “clear,” that changes the state to “on” and “off”, respectively. Binary relations, Equality and Trigger, are defined between Bits. The Equality relation always makes the states of the related Bits the same, while the Trigger relation activates the target Bit to be “on” if the source Bit becomes “on,” but takes no actions on the other situations.

For example, let us assume the structure as illustrated in Figure 1. In this system, the four nodes, b1, b2, b3 and b4, represent

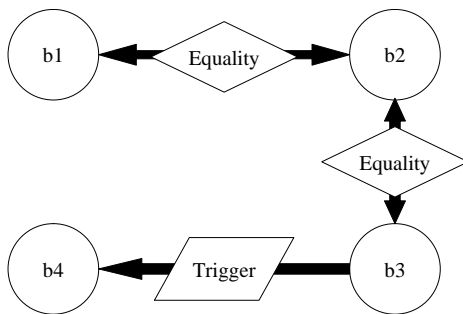


Figure 1. Bit Relations

instances of Bit; b1 and b2 are connected by an Equality relation and so are b2 and b3; b3 triggers b4. If b1 receives a message “set,” then the “set” message is also sent to b2, which in turn sends the “set” message to b3. Furthermore, the “set” message is sent to b4 because b3 is a trigger of b4. However, no matter what is sent to b4, nothing happens to b3 or to the other nodes. Note that each of b2 and b3 is involved in two different relations, which requires special care. Also, some mechanism of preventing the message propagation reflecting back to the sender is required, otherwise the propagation will continue infinitely.

The problem is to make this system scalable and evolvable, separating the definitions of the Bit objects and the Equality and Trigger relations. More concretely, the questions are:

- Is it easy to add a new node?
- Is it easy to add a new type of nodes?
- Is it easy to add a new relation?
- Is it easy to add a new type of relations?

3.2. Problems with Current OO Approaches

It has been argued that conventional object oriented techniques are not adequate to meet the above requirements [28, 12, 24]. A simple approach of embedding relations in the Bit class definition obviously harms the independence of Bit from relation definitions. Applying design patterns, particularly the mediator pattern and the observer pattern, may look promising. However, they are not so effective as expected, because:

Mediator Pattern Bit class must know the mediator that implements Equality, Trigger, etc. and thus it directly depends on the mediator definition.

Observer Pattern The observer pattern is better than the mediator pattern for dealing with the case where a Bit instance is involved in multiple relations. However, Bit has to accept observers and also has to notify observers when it changes its state. The former is usually implemented by inheriting “Subject” superclass or interface and the latter introduces some change to the Bit method declaration, which makes it impossible to reuse the existing Bit definition entirely. Moreover, an observer should be created for each event distinguishing the event source and the event type. Thus, for each Equality relation, two observers should be allocated for each operation, set and clear, resulting in four observers for one Equality instance, which is awkward. It also implies that the observer should have to know what kind of events it is to watch, which harms the independence of the relation definition.

Using AspectJ doesn’t work nicely either [28, 24]. Implementing Equality as an Aspect does not scale for new equality instance introduction. Preventing unbounded recursion doesn’t work due to lack of Aspect instantiation².

One probable solution to the problem is maintaining a table that keeps data of all relations as well as another table keeping data of all nodes. Then, it would be relatively easy to add new relations or new nodes. However, these tables and the control procedure using them are global by nature, while adding (and deleting) relations and nodes is a local operation. This is a solution that solves a local problem globally, which in general is not desirable.

Sullivan & Notkin [29] treated this problem with ABT (Abstract Behavior Types). In their solution, the Bit class defines operations, “set” and “clear” and also announces events, “justset” and “justcleared”. Relations such as Equality are defined as a mediator that listens to events and invokes corresponding operations. Compared to the mediator pattern in the ordinary object-oriented framework, the Bit object doesn’t have to know the existence and the interface of the mediator but it is required to explicitly raise events to be used by the mediator.

3.3. Our Solution

We claim that the problem is solved elegantly using EpsilonJ. The definition of Bit is natural and simple, totally independent from the Equality and Trigger relations.

²The latest version of AspectJ allows Aspect instantiation but there still remain related problems as Sakurai et al.[24] shows.

```

class Bit {
  boolean state=false;
  void set() {state=true;}
  void clear() {state=false;}
  boolean get() {return state;}
}

```

The Equality relation is defined as a context, requiring an interface `ActionInterface` that simply declares a method without parameters and return values.

```

interface ActionInterface {
  void fire();}
context Equality {
  boolean busy=false;
  // A flag to prevent infinite recursion
  static role Actor1 requires ActionInterface {
    void fire() {
      super.fire();
      if(!busy) {
        busy=true;
        Actor2.fire();
        // Role instance method "fire" of
        // all Actor2 instances are called
        busy=false;
      }}}
  static role Actor2 requires ActionInterface {
    // Similar to Actor1
  }
}

```

The Equality definition above is totally independent from that of Bit or any other types of components. The structure of this context is illustrated in Figure 2.

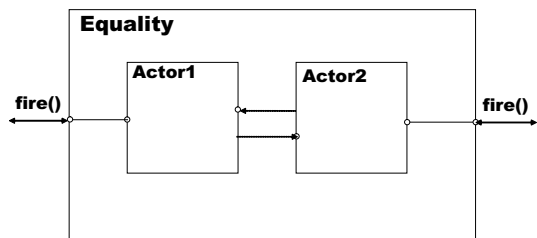


Figure 2. Structure of Equality Context

Combining this context and Bit, the integrated system of Figure 1 can be defined exploiting binding operations as follows.

```

Bit b1=new Bit(); Bit b2=new Bit();
Bit b3=new Bit(); Bit b4=new Bit();
Equality e12set = new Equality();
Equality e23set = new Equality();
Trigger t34 = new Trigger();
e12set.Actor1.bind(b1)
  replacing fire() with set();
e12set.Actor2.bind(b2)
  replacing fire() with set();
...

```

3.4. Results

We have succeeded in separating Bit and Equality; they are totally independent and reusable. As the context Equality must be instantiated, its context variable `busy` is an instance variable created for each context, which is convenient for implementing the infinite loop prevention mechanism. The method replacement operation of `import & export` in this case, together with the method renaming capability, was powerful enough to allow a very concise description. The design scales to new `node(Bit)` introduction and new types of nodes introduction as well as new relation instance introduction and new type relation introduction.

Besides the integrated system, we have written various kinds of examples, including the mediator pattern, the observer pattern, the visitor pattern, Kendall's bureaucracy structure, a rental shop business, the contract net protocol([26]) and the dining philosophers problem.

4. Implementation

An early version of Epsilon was implemented on ABCL/R3, a reflective concurrent object-oriented language [17] but the syntax and semantics at the time was considerably different from the language explained in this paper [32].

A preliminary implementation of EpsilonJ was done on Ruby by the third author. Ruby, created by Y. Matsumoto, is a full object-oriented language like Smalltalk with the feature of scripting languages like Perl [31]. It has such a nice feature as adding methods to a class and even to an object instance at runtime, which is quite convenient for implementing a language like EpsilonJ.

As it was implemented on Ruby, its syntax is different from EpsilonJ and so we gave a different name, Bunraku³, to this implementation. The implementation of Bunraku was made simple by sacrificing static type checking. As the platform Ruby was a type-less language, this design decision was natural.

Currently, we are also implementing EpsilonJ on Java. The basic idea is to use the annotation feature of Java2 5.0 so that it is implemented totally within the scope of standard Java. Context and Role's are declared like:

```

class @Context class Company {
  @StaticRole abstract class Employer
    extends RoleBase<Employer> {
    ...
  }
  @Role abstract class Employee
    extends RoleBase<Employee> {
    ...
  }
}

```

Context is defined as a class and Role's are defined as inner classes of the Context class but they are annotated by `@Context` and `@StaticRole` (in the static case) or `@Role`, respectively.

³Bunraku is the Japanese traditional and most refined puppet show performed by the collaboration of puppet manipulation, joruri recitation and shamisen music with roles played by puppets. Moreover, the name "Bunraku" shares three letters, "b", "u", and "r", with Ruby.

Role classes are declared `abstract`, because some method bodies are supplied at runtime when the binding of a role and an object is executed. A set of basic role methods, including `bind`, `new-bind`, and `unbind`, are defined in `RoleBase` class and every role class has to inherit it. The `requires` phrase is actually designated by the standard interface `implements` phrase. Creating a new instance of contexts and roles is executed by a special factory method and thus the use of `new` operator explained in the preceding subsections is modified here. Besides this point, the syntax explained in Section 2.3 is partly modified in this implementation, but the features are essentially the same.

Some annotation types can be read at runtime and with the reflective APIs they can change the program behaviors. This mechanism is employed for implementing dynamic features of `EpsilonJ`, including binding and unbinding.

The current implementation does not yet support some features like method name replacement and access to a *super* method but they will be realized in short time.

5. Related Work

Besides the work referred so far, all work of Aspect-Oriented Software Development, i.e. Aspect-Oriented Programming, MDSOC, Composition Filters and Demeter, are naturally related to our approach. As discussed in Section 2, MDSOC is relatively closer to our approach among them. A clear difference between MDSOC (or more concretely `Hyper/J`) and `EpsilonJ` is that composition of “features” in MDSOC is on the class-to-class basis at compile time, while composition of roles with objects in `EpsilonJ` is on the instance-to-instance basis at runtime. Since the composition in `EpsilonJ` is dynamic, it is possible and natural to realize decomposition (unbinding).

Aspect-oriented programming with `AspectJ` has a feature of adding aspects dynamically as well as statically [14]. The main objective of writing aspects is to deal with cross-cutting concerns. It implies that there already exists some structure of module decomposition but in adding a new type of concern, related pieces of code are distributed among modules, cross-cutting the existing structure. Our intention is that each concern can be encapsulated in a collaboration context that has a clear meaning and can be comprehended independently. They are related through objects that participate in multiple collaborations and so there should be no problem of “cross-cutting”.

Although there have been efforts of designing software from the beginning based on the AOP method under the name of “early aspects” [20], the normal framework of mind for thinking aspects assumes the existing program code as a target of inserting advices to join points. On the other hand, `Epsilon`’s way of thinking assumes no existing code and designs collaboration contexts independently. The work corresponding to designating pointcuts and attaching advices is executed by binding objects to roles. `AspectJ` provides features of specifying sophisticated pointcut conditions, whereas `EpsilonJ` only allows replacement on the method-to-method basis. Our experience of writing a number of examples in `EpsilonJ` lets us believe that this limitation of `EpsilonJ` practically brings no problems while it enhances the level of resulting programs’ behavioral comprehension but we have to accumulate

more experiences, developing large application programs to really endorse this claim.

Some research attempts at describing aspects in terms of role models have been reported (e.g. [6]). Among them, the Caesar model [18] is particularly related with our work, which will be discussed in Section 6.

In addition to the literature referred in Section 2, we introduce some more work of role modelling that are worth noting.

Contracts, proposed by R. Helm et al. [8], is a construct for the explicit specification of behavioral compositions. A contract defines a set of communicating participants and their contractual obligations. This notion of *participants* correspond to roles but participants are actually slots whose required attributes and methods have to be supplied by objects that commit to the contract.

D. Riehle extended the approach of role modelling to deal with object migration [36] and to design composite patterns [22] and frameworks [23]. B. Kristensen et al. also presented a conceptual framework of role modelling [15, 16]. They listed up the characteristics of roles: visibility, dependency, identity, dynamicity, multiplicity, and abstractivity. They proposed a graphical notation for illustrating roles and did some experiments on supporting roles by programming languages. Our model and language are more rigorously defined and concrete but it is interesting to characterize them from their conceptual viewpoint.

Gottlob et al. [5] deals with dynamic change of objects using the concept of roles. Since their main concern is data base, objects are more like data base schemas. They claim that inheritance is class based and thus inconvenient for handling dynamic changes. Instead, they propose a role hierarchy and realize specialization and inheritance at the instance level.

As we emphasize the dynamic feature of our model, the delegation mechanism should naturally be associated. Actually, since messages sent to objects bound to roles are possibly dispatched to role methods, it can be regarded as a kind of delegation. Delegation is a relatively primitive mechanism and it may be convenient to implement `Epsilon` in a prototype-based language like *Self* [27]. There are attempts to give structure in delegation mechanism like collaboration. For example, Bardou & Dony [2] proposed a notion of “split object” that is composed of a number of pieces and arriving messages are delegated to proper pieces. Roles in our model, in a way, correspond to the “pieces” but the meaning of roles in `Epsilon` are clarified by the collaboration context.

One of the promising applications of our model is mobile agent systems. When an object (or an agent) moves to a new site, it adapts to the new environment by assuming a role in that context. This situation can be appropriately described by the `Epsilon` model. Our early result is reported in [33].

Multiple inheritance is certainly related to the notion of object and role binding. Specifically, an elegant construct of *mixin* [1] that essentially realizes multiple inheritance has inspired the design of `EpsilonJ`. The idea and the syntactic phrase of *required interface* were borrowed from `McJava` [12], being developed in parallel to the work presented in this paper under the same research project called `Kumiki`. Mixins in `McJava` can be composed with objects that supply their required interface, thus the role and object binding can be effectively simulated. However, the *mixin-object* composition is static on the class basis, which differs from the dynamic characteristics of binding/unbinding operations in `EpsilonJ`.

McJava rather puts emphasis on assuring the type safety property of programs.

6. Discussions and Conclusion

Here, we will discuss the comparison of Caesar and EpsilonJ. The goal of Caesar is to decouple aspect interface, aspect implementation and aspect binding [18]. The aspect interface is called ACI (Aspect Collaboration Interface) with multiple mutually recursive nested types, which roughly corresponds to the context of EpsilonJ, where a role corresponds to a nested type.

For example, the observer pattern is written in Caesar by a pair of ACI and its implementation as follows.

```
interface ObserverProtocol {
    interface Subject {
        provided void add(Observer(Observer o);
        provided void removeObserver(Observer o);
        provided void changed();
        expected String getState();
    }
    interface Observer
        {expected void notify(Subject s);}
}
class ObserverProtocolImpl
    implements ObserverProtocol {
    class Subject {
        List observers = new LinkedList();
        void addObserver(Observer o) {
            observers.add(o);
        }
        void removeObserver(Observer o) {
            observers.remove(o);
        }
        void changed() {
            Iterator it = observers.iterator();
            while (iter.hasNext())
                ((Observer)iter.next()).notify(this);
        }
    }
}
```

The same pattern can be written in EpsilonJ as:

```
context ObserverPattern {
    static role Subject
        requires{void changed();} {
        void changed() {
            super.changed();
            Observer.notify(this);
        }
    }
    role Observer
        requires{void notify(Subject);} {}
}
```

This is more concise, partly because the interface and the implementation is not separated in EpsilonJ. But the essential point of the observer pattern is the interaction that when the subject's state is changed, it notify's the observers. This crucial behavior is not expressed in the interface of ObserverProtocol in Caesar and only given by the "implementation". The other reason the description in EpsilonJ is shorter is that the operations of adding and removing observers can be omitted, because they are taken care of by the innate binding and unbinding mechanism of EpsilonJ.

A more important difference is the way of aspect binding in Caesar and the binding in EpsilonJ. Aspect binding in Caesar is on the class basis, employing the wrapping mechanism. This design results in introducing many constructs such as wrapper recycling to prevent multiple wrapping of the same object, most specific wrappers to handle polymorphism and static and dynamic deployment statements. All these features are realized without any specific constructs in EpsilonJ owing to the dynamic instance-based binding with type constraint (given by the `requires` interface).

Detailed comparison using the observer pattern example is very interesting but due to the space limitation, it will be deferred to another paper.

In the introduction, we mentioned that an environment changes over time and objects in that environment should change accordingly. However, our model does not directly support changes of environments. It is possible to simulate the situation where objects are affected by environment change by preparing multiple environments and letting objects leave one environment to enter another.

This is based on our design decision. If we allow flexibility at all levels, be it context/role or object, it is hard to design a model based on distinction between what is relatively stable and what is not. We confine the adaptation feature to objects' dynamic participation into and separation from collaboration contexts, which we assume makes modelling simpler. So our intended programming style is to accumulate typical collaboration contexts as a reuse library, build domain objects according to applications, create concrete collaboration contexts using the library components and enter objects to appropriate collaborations.

In summary, our approach of binding objects and roles have the following characteristics:

1. Composition takes place when an object instance and a role instance are bound together;
2. An object instance can be bound to multiple role instances residing in different contexts;
3. The state of an object and that of a role construct a Cartesian product state after composition;
4. Through interface, a role may import methods of the binding object and/or export methods to the binding object. This will affect the combined states and the combined behavior of the role-object composite and thus realize an organic union.

Acknowledgments

The authors would like to thank other members of our research group, especially Hidehiko Masuhara, Daisuke Nishimura, and Tetsuo Kamina for their programming experiments and fruitful discussions. This research has been conducted under Kumi Project, supported as a Grant-in-Aid for Scientific Research (13224087) by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan.

References

- [1] G. B. and William Cook. Mixin-based inheritance. In *OOP-SLA 1990*, pages 303–311, 1990.

- [2] D. Bardou and C. Dony. Split objects: a disciplined use of delegation within objects. In *OOPSLA '96*, pages 122–137, San Jose, California, USA, Oct. 1996.
- [3] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *OOPSLA '89*, pages 1–6, 1989.
- [4] M. Fowler. Dealing with roles. <http://www2.awl.com/cseng/titles/0-201-89542-0/apsupp/>. supplemental information to *Analysis Pattern*, Addison-Wesley, 1997.
- [5] G. Gottlob, M. Schrefl, and Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.
- [6] K. B. Graverson and K. Osterbye. Aspect modelling as role modelling. In *OOPSLA 2002 Workshop on TS4AOSD*, Seattle, Nov. 2002.
- [7] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA '93*, pages 411–428, 1993.
- [8] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ECOOP/OOPSLA '90*, pages 169–180, Oct. 1990.
- [9] Y. Honda, S. Watari, and M. Tokoro. Compositional adaptation: A new method for constructing software for open-ended systems. *Computer Software*, 9(2):122–136, 1992. in Japanese.
- [10] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, 1999.
- [11] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM press, 1992.
- [12] T. Kamina and T. Tamai. A core calculus for mixin-types. In *Foundations of Object-Oriented Languages (FOOL11)*, Venice, Italy, Jan. 2004. In conjunction with POPL 2004.
- [13] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In *OOPSLA '99*, pages 353–369, Nov. 1999.
- [14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*. Springer-Verlag, June 1997.
- [15] B. B. Kristensen. Object-oriented modeling with roles. In *Proceedings of the 2nd International Conference on Object-Oriented Information Systems (OOIS'95)*, pages 57–71, Dublin, Ireland, 1995.
- [16] B. B. Kristensen and K. Osterbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.
- [17] H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing parallel language constructs using a reflective object-oriented language. In *Reflection Symposium '96*, pages 79–91, Apr. 1996.
- [18] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 90–99, Boston, Mar. 2003.
- [19] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *CACM*, 44(10):43–50, Oct. 2001.
- [20] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: a model for aspect-oriented requirements engineering. In *Proceedings of the International Conference on Requirements Engineering (RE 2002)*, pages 9–13, Essen, Germany, Sept. 2002. IEEE.
- [21] T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects: the OOram Software Engineering Method*. Manning Publications, Greenwich, 1996.
- [22] D. Riehle. Composite design patterns. In *OOPSLA '97*, pages 218–228, oct. 1997.
- [23] D. Riehle and T. Gross. Role model based framework design and integration. In *OOPSLA '98*, pages 117–133, Vancouver, oct. 1998.
- [24] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspect. In *Proceedings of 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, Mar. 2004.
- [25] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.
- [26] D. R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. on Computers*, 29(12):1104–1113, 1980.
- [27] R. B. Smith and D. Ungar. Programming as an experience: The inspiration for self. In *ECOOP '95*, pages 303–330, Oct. 1995.
- [28] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for AspectJ. In *1st Proceedings of 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 19–26, Enschede, Holland, Apr. 2002.
- [29] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transaction on Software Engineering and Methodology*, 1(3):229–268, 1992.
- [30] T. Tamai. Objects and roles: modeling based on the dualistic view. *Information and Software Technology*, 41(14):1005–1010, 1999.
- [31] D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.
- [32] N. Ubayashi and T. Tamai. An evolutionary cooperative computation based on adaptation to environment. In *Proc. Asia Pacific Software Engineering Conference '99*, pages 334–341, Takamatsu, Japan, Dec. 1999. IEEE Computer Society.
- [33] N. Ubayashi and T. Tamai. Separation of concerns in mobile agent applications. In *Proceedings of the 3rd International Conference REFLECTION 2001, LNCS 2192*, pages 89–109, Kyoto, Sept. 2001. Springer.
- [34] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.
- [35] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *OOPSLA '96*, pages 359–369, 1996.
- [36] R. Wieringa, W. de Jonge, and P. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.
- [37] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, 1990.