Towards Safe and Flexible Object Adaptation

Tetsuo Kamina
The University of Tokyo
7-3-1 Hongo Bunkyo-ku Tokyo
113-0033 Japan
kamina@acm.org

Tetsuo Tamai
The University of Tokyo
3-8-1 Komaba Meguro-ku Tokyo
153-8902 Japan
tamai@acm.org

ABSTRACT

In this paper, a programming language NextEJ is proposed. NextEJ is based on *Epsilon model*, which realizes object adaptation to contexts. The novelty of Epsilon model is its ability to make objects be able to freely enter or leave contexts dynamically and belong to multiple contexts at a time. However, such kind of flexibility also easily brings type-unsafety. NextEJ tackles this problem by introducing a new feature called *context activation scope.* Inside a context activation scope, it is assured that an object is always bound with the role activated so that no method-not-understood errors occur at run-time. Furthermore, context activation scope can be nested so that multiple contexts can be activated at a time. A role instance has a pre-defined field `thisContext` which refers to its enclosing context instance. In the case of multiple context activations, the reference of `thisContext` is interpreted as a composite context whose behavior is determined by the order of activations.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Role model, Epsilon, NextEJ

1. INTRODUCTION

Context-awareness is becoming an increasingly important feature of many kinds of applications, ranging from business applications to mobile and ubiquitous computing systems. To explicitly support context-awareness in programming language level, a new programming approach, called *Context-oriented Programming (COP)*, and its implementing languages have been proposed[14, 8, 13].

There are many challenges in realizing COP. For example, to achieve context-awareness, a context (or layer) should be a first-class entity explicitly referred at run-time. To realize context-dependent behavioral changes of objects, the relationship between objects and contexts should be changed at any time, and multiple contexts may affect the behavior of an object. To enhance reuse of code, each context should be implemented as a reusable module. To realize all of these requirements is actually a challenging issue; for example, ContextJ and its friends (such as ContextL and ContextS) provide a flexible mechanism of activating/deactivating some pieces of code at run-time, but allow that only for whole classes at once. Thus, they does not support instance-specific context-dependent behavior, where the context-dependent behavior is activated only for a single instance.

One of the promising way of realizing these requirements is to adopt *Epsilon model* [28, 29] in COP. In Epsilon, a context is defined as a field of collaboration between *roles* and an object adapts to the context assuming one of the roles. Objects can freely enter or leave contexts and belong to multiple contexts at a time. Contexts and roles are the first class constructs at run-time referred by their names. Furthermore, contexts are independent reusable components to be deployed separately from objects. A Java-based implementation language *EpsilonJ* is also developed [22].

The flexibility provided by Epsilon and EpsilonJ, however, easily brings type-unsafety. Even though our previous work reveals that where execution of EpsilonJ program gets stuck in a core language level [16] so that we can insert dynamic type-checking in the compiled code, the underlying calculus shows its essential type unsafety including the need of downcasting.

In this paper, we propose a programming language NextEJ. NextEJ tackles the problem of type unsafety of Epsilon model by introducing a new language feature called *context activation scope.* In the context activation scope, we can denote which role of objects belonging to an context is bound and activated inside the scope. If the designated object is not bound with the role, the role instance is implicitly created and bound with the object, so it is assured that the object always assumes the role inside the scope and no method-not-understood errors occur at run-time. Furthermore, context activation scopes can be nested so that multiple contexts can be activated at a time. A role instance has a pre-defined field `thisContext` which refers to its enclosing context instance. In the case of multiple context activations, the reference of

```
context Building {                  context Shop {
  role Guest {                        role Customer {
    void escape() { .. }                void buy(Item i) {
  }                                       int p = i.getPrice();
  role Security {                         Seller.getPaid(p);
    void notify() {                     } }
      Guest.escape();                 static role Seller {
    }                                   void getPaid(int price)
  }                                       { ... } }
}                                   }
```

Figure 1: Context and role declarations

thisContext is interpreted as a composite context whose behavior is determined by the order of activations.

Contributions of this paper are three folds:

- Designing a role-based COP language NextEJ, a type safe variant of EpsilonJ satisfying the requirements of COP.
- Introducing the feature of multiple context activation into Epsilon model, realizing a more natural and flexible way for representing context-awareness.
- The mechanism of composite contexts and ordering of multiple context activations.

This paper reports the design concept of NextEJ using a simple example. Formal definition, proof of type soundness and other properties, language implementation, and more case studies remain as future work.

2. NEXTEJ: A SAFE AND FLEXIBLE COP LANGUAGE

2.1 An Example

To show how our proposal supports COP, we consider the following example. This example features two contexts, building and shop. Inside a building, there are several roles such as guest, administrator, security agent, and owner. Similarly, there are some roles inside a shop such as customer and seller. When a person enters a building, she assumes a role of guest. Similarly, she assumes a role of customer when she enters a shop. There are many interactions among roles; e.g., a security agent notifies all the guest in the case of emergency, or a seller sells the customer an item. When she leaves from a context (e.g. a building) she quits the role she assumes (e.g. a guest). Furthermore, shops may be inside a building, thus a person may enter multiple contexts (i.e. a building and a shop) at a time.

2.2 Context and Role Declarations

Figure 1 shows an example of context and role declarations in NextEJ. Each of structure of contexts and roles is the same as that of EpsilonJ's contexts and roles, respectively. The context `Building` consists of two roles, `Guest` and `Security`. Inside contexts and roles, we can declare methods and fields, just as classes. For example, the role `Guest` declares a method `escape()`, which is called in the body of `notify()` declared in `Security`.

A context can be instantiated by a `new` expression. On the other hand, an instance of role cannot be created explicitly, as we will see later. The connections of role instances and

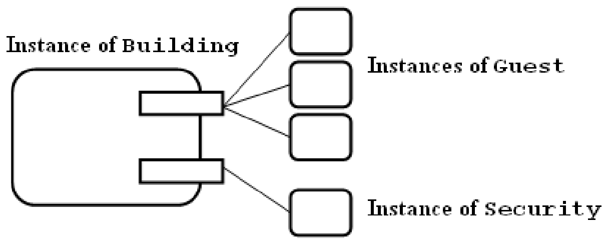

Figure 2: Structure of role instances and a context instance

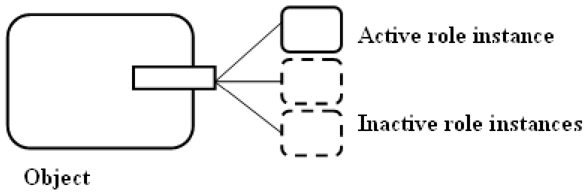

Figure 3: Structure of an object bound with role instances

the enclosing context instance are shown in Figure 2. A role instance is always associated with an instance of its enclosing context. A set of instances of a role associated with the same enclosing context instance is called a role group, which is referred by the role name. For example, the method call `Guest.escape()` is interpreted as calling methods `escape()` of all the `Guest` instances. A role declared as `static` is called a *singleton role*, which means that at most one instance of the role with the same enclosing context instance can be created.

2.3 Object Adaptation and Context Activation

An object entering contexts is created as a class instance, just as in Java. An object enters a context by assuming one of its role instances. Furthermore, an object can be bound with multiple role instances and can activate or deactivate some of them (Figure 3). For example, assuming that we have a class `Person`, object adaptation to a context can be written as follows:

```
Building midtown = new Building();
Person tanaka = new Person();
Person suzuki = new Person();
Person sato = new Person();
bind tanaka with midtown.Guest(),
     suzuki with midtown.Guest(),
     sato with midtown.Security() {
  ...
  sato.notify();
}
```

The sentence beginning from the keyword `bind` is called a *context activation scope*. Before entering the execution scope (enclosed between braces), it creates role instances and bind them with the corresponding objects, if these objects are not bound with the corresponding roles. If an object is already bound with the corresponding role, the role instance is not created but the existing role instance is *activated.* Inside the parentheses following the role name, we put the arguments for the constructor of the role. These arguments are used only when the object is not bound with the role so that the role instance is created.

After entering the execution scope, it is assumed that each object declared in the `bind` clause is bound with the corresponding role instance. For example, in the above code, `sato` is bound with a role `midtown.Security()` (which means `sato` enters the context `midtown` as a `Security`). Inside the following brace, `sato` acquires the behavior (and states) declared in `Building.Guest`, thus we can safely call the method `notify()` declared in `Building.Guest` on `sato`. Inside the context activation scope, it is considered that `sato` is a subtype of `Person` and `midtown.Guest`, like multiple inheritance or mixins [5][1]. As we will see later, a context can also be composed with another context, and a subtyping relation exists between a context and the composite context. To ensure type safety, all the variables referring to a context instance are considered `final`.

Note that outside the context activation scope, we cannot access methods declared in roles. It does not mean that the acquired role is discarded outside the scope. Instead, the role instance and its states are retained but *deactivated*, recovering the original behavior of the object. The retained role instance will be activated again if the object assumes the same role of the same context.

The idea of activation/deactivation of role instances is taken from ContextJ and one of the major differences from EpsilonJ[2]. Inside the context activation scope, it is always assumed that the object is bound with the corresponding role instance, thus we can safely access the role instance method. In EpsilonJ, on the other hand, once an object is bound with a role instance, this role instance is activated only through down-cast expressions. Since where an object is bound or unbound with the role instance cannot be determined statically, this down-casting may results in a cast-exception. Once the object is unbound with the role, the role instance becomes garbage. Instead, in NextEJ, the deactivated role instance may be activated again, preserving its states.

An object may also be unbound with the role instance, which is explained in section 2.6.

2.4 Multiple Context Activation

An object may enter multiple contexts. For example, there is a case where a shop is inside a building; in this case, a customer of the shop is also a guest of the enclosing building. To represent such a situation, context activation scope can be nested, as shown in the following example:

```
Building midtown = new Building();
Person tanaka = new Person();
bind tanaka with midtown.Guest() {
  ...
  Shop lawson = new Shop();
  Person sato = new Person();
  bind tanaka with lawson.Customer(),
       sato with lawson.Seller() {
    tanaka.buy(someItem);
  }
  ...
}
```

[1]`midtown.Guest` is a dependent type [24, 23, 25].

[2]As in ContextJ, the context activation scope is dynamically scoped.

```
context Building {               context Shop {
  String name;                     String name;
  Building(String name) {          Shop(String name) {
    this.name = name; }              this.name = name; }
  void currentPosition() {         void currentPosition() {
    System.out.println(              System.out.println(
      " "+name);                       " "+name);
    next();                          next();
  }                                }
  role Guest { ... }               role Customer { ... }
  role Security { ... }            role Seller { ... }
}                                }
```

Figure 4: Context method combination

In this example, `tanaka` firstly enters the context `midtown` as a `Guest`; then it enters the context `lawson`, which is located inside `midtown`, as a `Customer`; finally it buys `someItem` (and pays to `sato`, as described in Figure 1).

Note that multiple context activation is not supported by EpsilonJ, since in EpsilonJ any compositions among roles are not considered.

2.5 Referring the Enclosing Context and Composite Context

Another feature of NextEJ that is not provided by EpsilonJ is that, in NextEJ, the enclosing context instance and its methods can be accessed through the special field `thisContext` which is implicitly declared in all the role declarations and always refers to the enclosing context instance. Therefore, if contexts `Building` and `Shop` are declared as shown in Figure 4, the following code is allowed in NextEJ:

```
Building midtown = new Building("Midtown");
Person tanaka = new Person();
bind tanaka with midtown.Guest() {
  Shop lawson = new Shop("Lawson");
  bind tanaka with lawson.Customer() {
    tanaka.thisContext.currentPosition();
  }
}
```

In this code, the field `thisContext` is accessed on `tanaka`, which is allowed since `tanaka` is bound with role instances. Since the enclosing context instance declares a method `currentPosition` (which can be statically assured by using the information provided by the context activation scope), we can safely call `currentPosition` on `thisContext`, which prints where `tanaka` reside on the standard output. Note that the `thisContext` field is considered `final`, since it refers to a context instance (recall that all the variables referring to a context instance are considered `final`).

Note also that `tanaka` enters two contexts, `midtown` and `lawson`, and both of them declares method `currentPosition`. Actually, on `thisContext` we can access a composite context of `midtown` and `lawson`. The ordering of composition is determined by the order of activation; the inner most context always precedes the other contexts. In Figure 4, the declaration of `currentPosition` contains a method call `next()`, which is similar to `inner` of Beta [21, 10]. It calls the next method if it exists. If the next method does not exist, calling `next()` has no effects. Therefore, `currentPosition` declared in `Shop` is firstly called; then that declared in `Building` is called. The above code therefore prints a string `" Lawson Midtown"` on the standard output.

Currently inheritance relations between contexts at the definition level, which may complicate the method combination rule, are not considered.

2.6 Unbinding and Swapping Roles

As mentioned earlier, a role instance is deactivated outside the context activation scope. This deactivated role instance can be discarded, and we can bind the object with a new (fresh) instance of the role:

```
// All the states of midtown.Guest
// bound with tanaka is discarded.
tanaka.unbind(midtown.Guest);

// A fresh instance of midtown.Guest
// is bound with tanaka again.
bind tanaka with midtown.Guest() {
  ...
}
```

The `unbind()` method takes a name of role as an argument. If the receiver object is bound with the argument role, the argument role is removed from the receiver and becomes a garbage (If the role is not bound with the object, nothing happens).

Furthermore, as in EpsilonJ, another object may also assume the removed role instance. We can express it by the `bind` statement (context activation scope) followed by the `from` clause:

```
Person sato = new Person();
bind sato with lawson.Seller() from tanaka {
  ...
}
```

The above code results in that `tanaka` drops the instance of role `lawson.Seller` and `sato` takes over it (if `tanaka` is not bound with `lawson.Seller`, a new instance of it is created for `sato`).

To ensure that the role bound with the receiver of `unbind` (or successor of `from` clause) is actually deactivated, the method containing `unbind` operation has to declare the following `unbind` clause:

```
void unbindGuest(Person p)
     unbind Building.Guest {
  ...
  // midtown is an instance of Building
  p.unbind(midtown.Guest);
}
```

Calling `unbindGuest` method inside an context activation scope that activates `Guest` on any instance of `Building` is prohibited:

```
bind tanaka with midtown.Guest {
  unbindGuest(tanaka) // compile error !!
}
```

2.7 Other Features Taken from EpsilonJ

NextEJ also has a couple of features found in EpsilonJ. For example, a role may declare a *required interface*. This 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` clause as follows:

```
context Building {
  role Guest requires { String name(); } {
    ... }
}
```

When a required interface is declared to a role, methods can be imported from the binding object. For example, supposing that `Person` has a method `name()`, in the aforementioned `bind` statements the method `name()` of `tanaka` is imported to the `Guest` role instance through the interface.

The imported method can be used in the body of role declaration. Furthermore, the role may override the imported method, and in the overriding method, we may call the original (overridden) method by calling the method with the same signature on `super`.

For type-checking of this binding, it is only necessary for the class to have a method that has the same name and the same signature required by the role. In other words, the class has to be a *structural subtype* of the `requires` interface[3].

2.8 Properties

NextEJ has the following features (Each of section number in parentheses indicates where the feature is explained):

Behavioral variations. In NextEJ, we may specialize the behavior of object by binding a role that overrides the object's method (section 2.7). Furthermore, the context and role methods can also be specialized by composing contexts (section 2.4).

Layers. Related context-dependent behavioral variations are grouped as a context, which is a first-class entity that can be explicitly referred by its name (section 2.2).

Activation. Contexts can be activated and deactivated dynamically at run-time. In NextEJ, this activation is performed on a single, particular object so that instance-specific context-dependent behavioral variations are supported (section 2.3).

Context. The special field `thisContext` provides the way to access the enclosing context instance and its methods (section 2.5).

Scoping. The scope within which contexts are activated or deactivated is explicitly controlled by the context activation scope (section 2.3). Context activation scope can be nested so that multiple contexts can be activated at a time (section 2.4).

Therefore, it can be said that NextEJ satisfies all the requirements of COP mentioned in [14]. Furthermore, NextEJ has the features provided by Epsilon model including encapsulation of collaboration between roles, unbinding and swapping mechanisms, required interfaces and structural subtyping. NextEJ resolves type safety problems of EpsilonJ such as dynamic cast errors.

[3]A similar mechanism is also found in McJava, a Java extension with mixins [15].

3. RELATED WORK

We have overviewed the main features of NextEJ and compare it with EpsilonJ. We also briefly mention about the relationship between ContextJ and this work in section 1. In this section, we discuss relationship between NextEJ and other related work.

Epsilon model, on which this work is based, is related to aspect-oriented programming (AOP). AOP has a feature of adding aspects dynamically as well as statically [17]. One of the most major AOP language is AspectJ [18], which is an AOP language based on Java. The main objective of writing aspects is to deal with cross-cutting concerns. It implies that there are already exists some structure of module decomposition. Although efforts have been made to design software based on the AOP principle from the beginning, the normal framework of mind for thinking aspects assumes the existing program code as a target of inserting advices to join points. Instead, Epsilon does not assume any existing code and designs collaboration contexts independently. The work corresponding to designating pointcuts and attaching advices is executed by binding objects to roles.

Delegation Layers [26] provide flexible object based composition of collaborations. They combine the mechanism of delegation [19, 27] and virtual classes [20, 6], or Family Polymorphism [9]; roles may be represented by virtual classes, and composition is instance-based using delegation mechanism. This approach, however, do not successfully represent object adaptation described in this paper. For example, in NextEJ the object after assuming a role may dynamically throw the role away, and even the thrown role may be assumed by another object and states held in the role instance are take over by the latter object.

ObjectTeams [12] also has a similar mechanism of role binding. In ObjectTeams, each instance of a bound role class internally stores a reference to its base object. This reference cannot be changed during its lifetime. By *lowering* (retrieving the base object from a role object) and *lifting* (the reverse translation of lowering), we can safely change the behavior of the object at run-time. As in NextEJ, a *team* (a construct of ObjectTeams corresponds to a context in NextEJ) can be activated and deactivated. However, in ObjectTeams, the role binding is class-based and declared at the class declaring time. Thus the instance-based binding and activation provided by NextEJ are not supported by ObjectTeams.

powerJava [2] is also a similar language with NextEJ, in that roles and collaboration fields are the first class constructs, interaction between roles are encapsulated, and objects can participate in the interaction by assuming one of its roles. As in NextEJ, the type of role depends on the enclosing context instance. However, powerJava lacks the feature of role groups that is a powerful mechanism of getting role instances associated with the context instance reflectively. Role unbinding and swapping, and explicit ordering of context activation that affects method combination are also unconsidered.

Mixins [5] are related to roles in NextEJ in that mixins form partial definitions that can be reused with a number of classes that conform the requirements of mixins. Several extensions of Java with mixins have been proposed [11, 1, 15]. Even though mixin composition is originally performed at compile time, dynamic composition of mixins is also studied in a core calculus [3], and such kind of object level inheritance is also studied as *wrappers* [7, 4].

4. CONCLUSION AND FUTURE WORK

We have presented NextEJ, a safe and flexible role-based COP language. It provides a way of naturally representing context-awareness in the programming language level. Based on the object adaptation mechanism provided by Epsilon model, NextEJ supports a convenient COP feature of activating/deactivating contexts or roles. While such activation is only supported by down-casting in EpsilonJ, NextEJ provides a safe way of activation by context activation scope. Furthermore, in NextEJ multiple contexts can be activated at a time, and the behavior of composite context generated by such multiple context activations is determined by the order of activations.

This work, however, still in its early stage. We are planning to formalize the ideas presented in this paper to provide a solid basis for language processor implementation. We are also going to implement a prototypical compiler to study how this approach can be used in more realistic examples. Implementing a practical language processor or IDE will make a big progress on the research on COP.

Acknowledgments.

This work is supported in part by Grant-in-Aid for Scientific Research No.18200001 and Grant-in-Aid for Young Scientists (B) No.20700022 from NEXT of Japan.

5. REFERENCES

[1] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.

[2] M. Baldoni, G. Boella, and L. van der Torre. Interaction between objects in powerJava. *Journal of Object Technology*, 6(2):5–30, 2007.

[3] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. Safe and flexible objects with subtyping. *Journal of Object Technology*, 4(10):5–29, 2005.

[4] Lorenzo Bettini, Sara Capecchi, and Elena Giachino. Weatherweight Wrap Java. In *SAC'07*, pages 1094–1100, 2007.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

[6] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, volume 1445 of *LNCS*, pages 523–549, 1998.

[7] Martin Buchi and Wolfgang Weck. Generic wrappers. In *ECOOP 2000*, volume 1850 of *LNCS*, pages 201–225, 2000.

[8] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *Dynamic Language Symposium (DLS) '05*, pages 1–10, 2005.

[9] Eric Ernst. Family polymorphism. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 303–327, 2001.

[10] Erik Ernst. Propagating class and method combination. In *ECOOP'99*, volume 1628 of *LNCS*, pages 67–91. Springer-Verlag, 1999.

[11] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 98*, pages 171–183, 1998.

[12] Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[13] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE 2007*, volume 5235 of *LNCS*, pages 396–407, 2008.

[14] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[15] Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of Java with mixin-types. In *2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 398–414. Springer, 2004.

[16] Tetsuo Kamina and Tetsuo Tamai. Flexible object adaptation for Java-like languages. In *Proceedings of the 10th Workshop on Formal Techniques for Java-like Programs (FTfJP 2008)*, pages 63–76, 2008.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, 1997.

[18] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, M ik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[19] Gunter Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP'99*, volume 1628 of *LNCS*, pages 351–366, 1999.

[20] Ole Lehrmann Madsen and Birger Moller-Pdersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA'89*, pages 397–406, 1989.

[21] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[22] Supasit Monpratarnchai and Tetsuo Tamai. The implementation and execution framework of a role model based language, EpsilonJ. In *Proceedings of the 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'08)*, pages 269–276, 2008.

[23] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA'04*, pages 99–115, 2004.

[24] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 201–224, 2003.

[25] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA'05*, pages 41–57, 2005.

[26] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002*, volume 2374 of *LNCS*, pages 89–110, 2002.

[27] Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In *OOPSLA'01*, pages 283–299, 2001.

[28] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *International Conference on Software Engineering (ICSE 2005)*, pages 166–175, 2005.

[29] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. Objects as actors assuming roles in the environment. In *Software Engineering for Multi-Agent Systems V*, volume 4408 of *LNCS*, pages 185–203, 2007.