A Smooth Combination of Role-based Language and Context Activation

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, we propose a programming language called NextEJ. NextEJ is a smooth combination of a role-based language EpsilonJ and context activation mechanisms provided by COP languages. It supports all the features of the Epsilon model such as dynamic object-role binding and unbinding, and encapsulation of collaboration of roles as a context that can be defined as a reusable module. Furthermore, NextEJ tackles typing problem of the Epsilon model by introducing the *context activation scope* inspired by COP languages. The key ideas described in this paper are formalized as a small core language FEJ that is built on top of FJ. FEJ's type system is proven to be sound.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Role model, Epsilon, Context-oriented Programming

1. INTRODUCTION

Context-awareness is becoming an increasingly important feature in many types of applications, ranging from business applications to mobile and ubiquitous computing systems. For example, in location-based systems, the behaviors of the provided services are situation-dependent or even deeply personalized [6]; thus, instance-specific behavioral changes with respect to the surrounding context are required. Unfortunately, current mainstream object-oriented languages provide little explicit support for context-awareness [23].

To explicitly support context-awareness at the programming language level, several approaches have been proposed. One of the promising methods of realizing context-awareness is to use the *Epsilon model* [41, 42]. The Epsilon model provides a clear conceptual framework of role modeling and object adaptation to collaboration fields between roles. It also provides a good basis for developing context-aware applications. A Java-based implementation language called *EpsilonJ* has also been implemented [34].

In EpsilonJ, each context is declared by a context declaration statement (Figure 1). Each context consists of a set of roles that represents collaborations performed in that context. For example, Figure 1 shows collaborations between employers and employees in a company (e.g., an employer pays salary for the employees). Each context can be instantiated by using the `new` expression, as in Java. An object can participate in a context by assuming one of the role instances belonging to that context. This can be achieved by the `newBind` predefined method (Figure 2); this method creates a role instance and binds it with the object that is passed as an argument to `newBind`. An object can also assume other role instances belonging to other contexts.

EpsilonJ supports the development of context-aware applications. We can define behavioral variations that may vary with respect to the surrounding environment by using context declarations. In each context, we can group related context-dependent behaviors. Each context and role is a first-class entity that can be explicitly referred to by its name; thus, we can explicitly invoke context-dependent behaviors. However, in EpsilonJ, dynamically acquired methods obtained by assuming roles have to be invoked by using down-casting (line 6 of Figure 2); this is an unsafe operation. This down-casting is cumbersome and error-prone, because we always have to use it whenever we want to use role methods on a case-by-case basis. In addition, to assure that the receiver object is actually bound with the role, we have to

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

Figure 1: Example of context in EpsilonJ

```
1 Person tanaka = new Person();
2 Person komiyama = new Person();
3 Company todai = new Company();
4 todai.Employer.newBind(komiyama);
5 todai.Employee.newBind(tanaka,1000);
6 ((todai.Employer)komiyama).pay();
```

Figure 2: Object-role binding in EpsilonJ

investigate the source code carefully.

One of the reasons for this problem is the fact that there is no method to control the scope of object-role binding. If we can represent a scope that ensures that the designated objects and roles are bound, this problem would be solved. This scoping mechanism is similar to the one that has been described in *Context-oriented Programming (COP)* [23, 13, 22]. However, COP languages do not provide the object adaptation mechanism supported by EpsilonJ because the binding mechanism is not object-based but class-based. Furthermore, in most the statically typed COP languages, such binding is performed statically and only the *activation* mechanism is provided. Therefore, the flexible adaptive evolution provided by EpsilonJ is not supported by COP languages.

We have proposed a smooth combination of the Epsilon model and COP languages that incorporates the advantages of both. We design a new programming language called NextEJ [28][1] that incorporates the feature of both Epsilon model and COP. NextEJ addresses the problem of the type unsafety of the Epsilon model by introducing a language feature, the concept of which is adopted from COP languages, called *context activation scope*. In the context activation scope, we can denote which role of an object that belongs to an context is bound and activated within the scope. If the designated object is not bound with the role, the role instance is implicitly created and bound with the object, and therefore it is ensured that the object always assumes the role within 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 `thisC` that refers to its enclosing context instance. In the case of multiple context activations, the reference of `thisC` is interpreted as a composite context whose behavior is determined by the order of activation.

To carefully investigate the type soundness of NextEJ, we develop FEJ, a core calculus that combines the features of EpsilonJ and COP. This formalization is built on top of Featherweight Java (FJ) [24] and its type system is proven to be sound. There have been only a few reports on the formalization of the Epsilon model and COP languages [27, 12], and there have been non on the combination of role-based languages and COP. The formalization described in this paper can also be considered as a theoretical basis for similar languages such as ObjectTeams [21]. We discuss the relationship between NextEJ and ObjectTeams in section 4.

2. NEXTEJ: SMOOTH COMBINATION OF EPSILON AND COP

This section describes how context-awareness can be easily expressed by NextEJ. A design sketch of NextEJ was presented in [28]. For the sake of simplicity, we have included this sketch in this paper.

[1]The syntax of NextEJ has been improved in this paper.

```
class Building {
  role Guest {
    void escape() { .. }
  }
  role Security {
    void notify() {
      Guest.escape(); }
  }
}
```

```
class Shop {
  role Customer {
    void buy(Item i) {
      int p = i.getPrice();
      Seller.getPaid(p);
    } }
  singleton role Shopkeeper {
    void getPaid(int price)
      { ... } } }
```

Figure 3: Context and role declarations

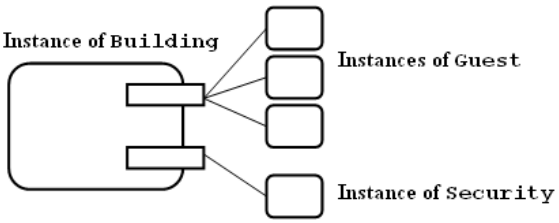

Figure 4: Structure of role instances and a context instance

2.1 An Example

To discuss the main features of NextEJ, we consider the following example. This example features two contexts, building and shop. Within a building, there exist several roles such as visitor, janitor, security agent, and owner. Similarly, within a shop there exist some roles such as customer and shopkeeper. When a person enters a building, she assumes the role of a visitor. Similarly, a person assumes the role of a customer when she enters a shop. There exist many interactions among roles; e.g., a security agent notifies all visitors in case of an emergency or a shopkeeper sells a customer an item. When a person leaves a context (e.g., building) she quits the role she assumes (e.g., visitor). Furthermore, shops may be within a building; therefore a person may simultaneously enter multiple contexts (i.e., building and shop).

2.2 Context and Role Declarations

NextEJ is an extension of Java that provides an explicit method to represent context-dependent behaviors and object adaptation to contexts. Figure 3 shows an example of context and role declarations in NextEJ. In NextEJ, each context is declared as a normal class and each role is declared within a class by using the `role` declaration statement. In Figure 3, the context `Building` consists of two roles, `Guest` and `Security`. Within roles, we can declare methods and fields. For example, the role `Guest` declares a method `escape()` that is called in the body of `notify()` declared in `Security`.

A context can be instantiated by a `new` expression because it is actually a normal class. On the other hand, an instance of a role cannot be created explicitly, as described later. The relationships among role instances and the enclosing context instance are shown in Figure 4. A role instance is always associated with an instance of its enclosing context. A set of

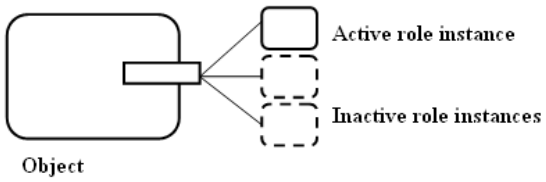

Figure 5: Structure of an object bound with role instances

instances of a role associated with the same enclosing context instance is called a role group, and it is referred by the role name. For example, the method call `Guest.escape()` is interpreted as calling the `escape()` methods of all `Guest` instances. A role declared as `singleton` is called a *singleton role*; this implies that at most one instance of the role with the same enclosing context instance can be created.

A role declaration cannot contain an `extends` clause; however, this does not mean that a role cannot extend other roles. The composition mechanism of contexts and roles is discussed in section 2.5.

2.3 Object Adaptation and Context Activation

A class instance enters a context by assuming one of its role instances. Furthermore, a class instance can be bound with multiple role instances and can activate or deactivate some of them (Figure 5). For example, assuming that we have a class `Person`, object adaptation to a context can be written as follows:

```
final 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 binds them with the corresponding class instances, if these class instances are not bound with the corresponding roles. If a class instance is already bound with the corresponding role, the role instance is not created but the existing role instance is *activated*. Within the parentheses following the role name, we specify the arguments for the constructor of the role. These arguments are used only when the class instance is not bound with the role so that the role instance is created.

After entering the execution scope, it is assumed that each class instance 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()` (implying that `sato` enters the context `midtown` as a `Security`). Within 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`. Within the context activation scope, it is considered that `sato` is a subtype of `Person` and `Building.Guest`, like multiple inheritance or mixins [9]. As described 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 have to be declared with the modifier `final`, because if a reference to a context instance changes during computation, it becomes very difficult to determine the actual type of role instances belonging to the context[2].

Note that outside the context activation scope, we cannot access methods declared in roles. This does not imply 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 reactivated if the object assumes the same role in the same context.

The idea of activation/deactivation of role instances is taken from ContextJ and is one of the major differences from EpsilonJ. Within 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. Because whether an object is bound or unbound with the role instance cannot be determined statically, this down-casting may result 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.

In NextEJ, the context activation is dynamically scoped, because role method invocation is based on role instances that are dynamically bound with a class instance, and this binding can go beyond the lexical scope (e.g., by passing the instance as an argument to a method invocation).

2.4 Multiple Context Activation

A class instance may enter multiple contexts. For example, there exists a case in which a shop is within a building; in this case, a customer of the shop is also a guest of the enclosing building. To represent such a situation (i.e., there are multiple contexts in which the person is participating), the context activation scope can be nested, as shown in the following example:

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

In this example, `tanaka` first enters the context `midtown` as a `Guest`; then, it enters the context `lawson`, located within

[2]There may be other approaches to solve this problem (e.g., using *exact types*) [10, 25]; however, we prefer to use the approach of "final context instance," because it simplifies the type system and it is easy to reason about the correctness of the program. Furthermore, we observe that the reference to a context is inherently difficult to change (like aspects in AOP); thus, we are not sure when "non-final" context instances become useful.

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

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

Figure 6: Context method combination

midtown, as a Customer; finally, it buys someItem (and pays to sato, as shown in Figure 3).

2.5 Referring the Enclosing Context and Composite Context

Another feature of NextEJ that is not provided by EpsilonJ is that the enclosing context instance and its methods can be accessed through the special field thisC that is implicitly declared in all role declarations and always refers to the enclosing context instance. Therefore, if contexts Building and Shop are declared as shown in Figure 6, the following code is allowed in NextEJ:

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

In this code, the field thisC is accessed on tanaka; this is allowed because tanaka is bound with role instances. Because the enclosing context instance declares a method currentPosition (that can be statically assured by using the information provided by the context activation scope), we can safely call currentPosition on thisC; this prints where tanaka reside on the standard output.

In addition, note that tanaka enters two contexts, midtown and lawson, both of which declare the method currentPosition. Actually, on thisC, we can access a composite context of midtown and lawson. The ordering of the composition is determined by the order of activation; the innermost context always precedes the other contexts. In Figure 6, the declaration of currentPosition contains a method call next() that is similar to inner of Beta [33, 16]. 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 first called; then, that declared in Building is called. The above code therefore prints a string " Lawson Midtown" on the standard output.

If each component context of a composite context has a role with the same name, an access to the role name is also interpreted as a composite role, and the innermost role always precedes the other contexts. This mechanism is similar to family polymorphism [15]. Furthermore, a context can also extend another context. If both of a context and its derived context declare roles with the same name, the role in the derived context implicitly inherits from the role in the super context, and no subtyping is defined between them. This mechanism is similar to lightweight family polymorphism [40]. In the case of method name conflict (i.e., a role method overrides methods provided by both its super role and another component role), the super role method precedes the others (like nested inheritance [35, 36]).

2.6 Swapping Roles

As mentioned earlier, a role instance is deactivated outside the context activation scope. This deactivated role instance can be discarded[3]. Furthermore, as in EpsilonJ, another class instance may also assume the removed role instance. We can express it by using 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 tanaka dropping the instance of role lawson.Seller and sato taking it over (if tanaka is not bound with lawson.Seller, a new instance of it is created for sato).

2.7 Other Features Taken from EpsilonJ

NextEJ also has a few other features found in EpsilonJ. For example, a role may declare a *required interface*. This is a method of defining an interface to a role and it is used at the time of binding with a class instance, requiring the class instance to supply that interface, i.e., the binding class instance should possess all the methods specified in the interface. A required interface can be declared using the requires clause as follows:

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

When a required interface is declared to a role, methods can be imported from the binding class instance. 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 the 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 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[4].

2.8 Summary

As discussed in [28], NextEJ supports all the basic requirements of COP languages. Furthermore, it provides a flexible mechanism for object adaptation that was originally proposed by the Epsilon model. In NextEJ, each context represents a concern so that separation of concerns is explicitly supported by the language. Unlike ContextJ's layer-in-class approach, contexts including roles can be units of reuse. Instance-specific context-dependent behaviors are supported

[3]This can be achieved using the unbind predefined method described in [28]

[4]A similar mechanism is also found in McJava, a Java extension with mixins [26].

$$
\begin{array}{rcl}
L & ::= & \texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\} \\
A & ::= & \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M_I}\ \}\{\bar{T}\ \bar{f};\ \bar{M}\} \\
T & ::= & C.R \mid \bar{C}.\bar{R} :: C \\
T_S & ::= & T \mid \{\ \bar{M_I};\ \} \\
M_I & ::= & T\ m(\bar{T}\ \bar{x}) \\
M & ::= & M_I\ \{\ \texttt{return}\ e;\ \} \\
e & ::= & x \mid e.f \mid e.m(\bar{e}) \mid \texttt{new}\ C(\bar{e}) \oplus \bar{r} \mid \\
 & & \texttt{bind}\ \bar{x}\ \texttt{with}\ \bar{r}\ \texttt{from}\ \bar{y}\ \{\ \bar{x}\bar{y}.e_0\ \} \\
v & ::= & \texttt{new}\ C(\bar{v}) \oplus \bar{r} \\
r & ::= & v.R(\bar{v})
\end{array}
$$

Figure 7: Syntax

$$T_S <: T_S \qquad \text{(S-REFL)}$$

$$\frac{C <: D \qquad D <: E}{C <: E} \qquad \text{(S-TRANS)}$$

$$\frac{T\ m(\bar{T}\ \bar{x}); \in \bar{M_I} \Rightarrow mtype(m, C) = \bar{T} \to T}{C <: \{\bar{M_I}\}} \qquad \text{(S-STRUCT)}$$

$$C.R :: T <: T \qquad \text{(S-MIXINC)}$$

$$C.R :: T <: C.R \qquad \text{(S-MIXINR)}$$

Figure 8: Subtyping

so that within the context activation scope, objects that are not bound with the roles are not affected by the activation. As in EpsilonJ, the role-class binding is performed at run-time. While developing a context, NextEJ does not assume any existing code (i.e., we can design contexts independently), because the `requires` interface only imposes structural subtyping on roles and classes.

Furthermore, unlike EpsilonJ, NextEJ provides a mechanism for clearly defining the scope where the context-dependent behaviors are activated. This scoping mechanism ensures type-soundness. In the following section, we provide the formalization of key ideas presented in this paper.

3. FEJ: CORE CALCULUS OF NEXTEJ

In this section, we formalize the core features of NextEJ described in the previous sections as a small calculus called FEJ. NextEJ provides a number of interesting features. In this paper, we focus on the most relevant features for object adaptation and context-activation mechanisms. FEJ is built on top of Featherweight Java (FJ) [24], a functional core of class-based object-oriented languages such as Java.

3.1 Syntax

The abstract syntax of FEJ is shown in Figure 7. The metavariables C and D range over classes; S, T, U, and V range over named types; M ranges over method declarations; L ranges over class declarations; Q and R range over roles; A ranges over role declarations; M_I ranges over method signatures; f and g range over fields; m and n range over method names; T_S ranges over types (including interface types); b, c, d, and e range over expressions; x and y range over variables; r and s range over role instances; and v and w range over values.

We write $\bar{M}$ as a shorthand for a possibly empty sequence $M_1 \cdots M_n$, $\bar{A}$ as a shorthand for $A_1 \cdots A_n$, and $\bar{e}$ as a shorthand for $e_1, \cdots, e_n$. We also abbreviate pairs of sequences in a similar manner, writing $\bar{T}\ \bar{f}$; as a shorthand for $T_1\ f_1; \cdots T_n\ f_n;$, $\bar{T}\ \bar{x}$ as a shorthand for $T_1\ x_1, \cdots, T_n\ x_n$, and $\bar{C}.\bar{R}$ as a shorthand for $C_1.R_1 :: \cdots :: C_n.R_n$. We denote an empty sequence as $\cdot$ and the length of sequence $\bar{e}$ as $\#(\bar{e})$. Sequences are assumed to contain no duplicate names.

In FEJ, the body of class declaration consists of field declarations, followed by method declarations and role declarations. Similarly, the body of role declaration consists of field declarations followed by method declarations. The inheritance of classes is omitted in this calculus, implying that the family-polymorphism-like features of NextEJ is totally excluded on FEJ. Although this feature is interesting and very useful, it is not technically a new feature. To realize a clear understanding of our new features provided by NextEJ, FEJ is designed to be concentrated on the most relevant features of object adaptation and context activation.

A named type is a class name, a role name (prefixed by a class name), or a composite type of a sequence of roles and a class. Named types appear in field declarations, signature of method declarations and constructor declarations, and return type of method declarations. On the other hand, interface types $\{\ \bar{M_I}\}$ can only appear in the `requires` clause of role declarations. The special variable `super` within the role declaration is assumed to have an interface type appearing in the `requires` clause.

A method declaration consists of the method signature and a return type, followed by its body. The body of the method declaration consists of just one `return` statement, implying that FEJ is a purely functional calculus.

There are five kinds of expressions in FEJ: variables, field accesses, method invocations, class instance creations, and bind expressions. A class instance creation consists of normal Java's `new` syntax and its associated role instances, which may vary during computation. The body of a bind expression only consists of one expression. Within the body of the bind expression, variables appearing in the `bind` clause and `from` clause are not considered as free variables. To explicitly denote this fact, we write $\bar{x}\bar{y}.e_0$ to imply that $\bar{x}$ and $\bar{y}$ are bound in e_0. We assume that `this` and `super` are special variables that are implicitly declared in method declarations.

An FEJ program is a pair of a class table CT and an expression e. A class table is a map from class names to class declarations. The expression e may be considered as the `main` method of the real NextEJ program. The class table is assumed to satisfy the following conditions: (1) $CT(C) =$ `class` $C \cdots$ for every $C \in dom(CT)$ and (2) $C \in dom(CT)$ for every class name appearing in the range of CT. In the derivation hypothesis shown below, we abbreviate $CT(C) =$ `class` $C \cdots$ as `class` $C \cdots$.

Subtyping rules of FEJ are shown in Figure 8. In FEJ, subtyping is a reflective and transitive closure of the mixin composition (::) relation. Furthermore, a class is a subtype of an interface if the class implements all the methods declared in the interface; thus, there exists structural subtyping between classes and interfaces.

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\}}{\mathit{fields}(C) = \bar{T}\ \bar{f}}$$

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\ \}\{\ \bar{S}\ \bar{g};\ \cdots\} \in \bar{A}}{\mathit{fields}(C.R) = \bar{S}\ \bar{g}}$$

$$\frac{\mathit{fields}(C) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, C) = T_i}$$

$$\frac{\mathit{fields}(C.R) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, C.R) = T_i}$$

$$\frac{\mathit{ftype}(f, C.R) = S}{\mathit{ftype}(f, C.R :: T) = S}$$

$$\frac{\mathit{fields}(C.R) = \bar{T}\ \bar{f} \qquad f \notin \bar{f}}{\mathit{ftype}(f, C.R :: T) = \mathit{ftype}(f, T)}$$

Figure 9: Field lookup

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\ \}}{\mathit{fvalue}(f_i, \texttt{new}\ C(\bar{e}) \oplus \cdot) = e_i} \quad \text{(FV-CLASS)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\}\{\ \bar{T}\ \bar{f}\ \cdots\ \} \in \bar{A}}{\mathit{fvalue}(f_i, \texttt{new}\ D(\cdots) \oplus \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = d_i} \quad \text{(FV-ROLE)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \qquad f \notin \bar{f} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\}\{\ \bar{T}\ \bar{f}; \cdots\ \} \in \bar{A}}{\mathit{fvalue}(f, \texttt{new}\ D(\cdots) \oplus \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = \mathit{fvalue}(f, \texttt{new}\ D(\cdots) \oplus \bar{r})} \quad \text{(FV-ROLE1)}$$

Figure 10: Field value lookup

3.2 Auxiliary Definitions

For the typing and reduction rules, we require a few auxiliary definitions. Field lookup functions are defined in Figure 9. The function $\mathit{fields}(\_)$, where $\_$ is either C or $C.R$, is a sequence $\bar{T}\ \bar{f}$ of field types and names declared in C or $C.R$. The function $\mathit{ftype}(f, T)$ returns the type S of field f, if f can be accessed on T. We write $f \notin \bar{f}$ to imply that the field f is not included in $\bar{f}$. The definitions of both *fields* and *ftype* are straightforward.

Figure 10 shows field value lookup rules. The function $\mathit{fvalue}(f, v)$ returns the value w of field f, if f can be accessed on either $\bar{r}$ or v. It first searches for f on the sequence of role instances $\bar{r}$. In this searching, the fields declared in the role instance of the left-most side of $\bar{r}$ are searched, followed by the next role instance. If f is not found on $\bar{r}$, fields declared in the class instance $\texttt{new}\ C(\bar{e})$ are searched.

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\} \quad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, C) = \bar{T} \to T}$$

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{A}\ \} \quad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\cdots\ \bar{M}\} \in \bar{A} \quad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, C.R) = \bar{T} \to T}$$

$$\frac{\texttt{class}\ C\ \{\bar{T}\ \bar{f}; \bar{N}\ \bar{A}\} \quad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\cdots\ \bar{M}\} \in \bar{A} \quad m \notin \bar{M} \qquad T\ m(\bar{T}\ \bar{x}) \in \bar{M}_I}{\mathit{mtype}(m, C.R) = \bar{T} \to T}$$

$$\frac{T\ m(\bar{T}\ \bar{x}) \in \bar{M}_I}{\mathit{mtype}(m, \{\ \bar{M}_I\ \}) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, C.R) = \bar{T} \to T}{\mathit{mtype}(m, C.R :: S) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, C.R) \text{ is undefined}}{\mathit{mtype}(m, C.R :: T) = \mathit{mtype}(m, T)}$$

Figure 11: Method type lookup

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\ \} \quad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, \texttt{new}\ C(\bar{e})) = \bar{x}.e} \quad \text{(MB-CLASS)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{A} \quad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = \bar{x}.e, \texttt{new}\ C(\bar{e}).R(\bar{d})} \quad \text{(MB-ROLE)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \qquad m \notin \bar{M} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{A}}{\mathit{mbody}(m, \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = \mathit{mbody}(m, \bar{r})} \quad \text{(MB-MIXIN)}$$

Figure 12: Method body lookup

Method lookup functions are defined in Figures 11 and 12. The type of method invocation m at T_S, written $\mathit{mtype}(m, T_S)$, is a pair of a sequence $\bar{T}$ of the formal parameter types and a return type T, written as $\bar{T} \to T$. We write $m \notin \bar{M}$ to imply that the method definition of the name m is not included in $\bar{M}$. The definition is also quite similar to field lookup functions; if T_S matches a mixin composition type $C.R :: T$, method declarations on $C.R$ are searched first. The body of the method invocation m on the sequence of role instances $\bar{r}$, written as $\mathit{mbody}(m, \bar{r})$, is a triple, written as $\bar{x}.e, r$, of the sequence of parameters $\bar{x}$, body e, and role instance r indicating where the method m is found from $\bar{r}$. The body

Bindability checking:

$$\frac{C \texttt{ <: } \{\bar{M}_I\} \qquad \texttt{class } D\ \{\cdots\ \bar{A}\} \qquad \texttt{role } R \texttt{ requires } \{\bar{M}_I\}\{\ \cdots\ \} \in \bar{A}}{\mathit{bindable}(D.R, C)}$$

Allowed unbinding:

$$\frac{T = \bar{D}.\bar{Q} :: C \qquad \forall D_i.Q_i \in \bar{D}.\bar{Q}, D_i.Q_i \in \bar{C}.\bar{Q}}{\mathit{unbindAllowed}(T, \bar{C}.\bar{Q})}$$

Figure 13: Other auxiliary functions

Well-formed role instance:

$$\frac{\mathit{bindable}(D.R, C) \qquad \mathit{fields}(D.R) = \bar{T}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U}\texttt{<:}\bar{T}}{\Gamma \vdash \mathit{roleOK}(D, R, \bar{e}, C)}$$

Expression typing:

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad \mathit{ftype}(f, S) = T}{\Gamma \vdash e_0.f : T} \qquad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash e_0 : T_S \qquad \Gamma \vdash \bar{e} : \bar{S} \qquad \mathit{mtype}(m, T_S) = \bar{T} \to T \qquad \bar{S}\texttt{<:}\bar{T}}{\Gamma \vdash e_0.m(\bar{e}) : T} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(C) = \bar{T}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{S} \qquad \bar{S}\texttt{<:}\bar{T} \qquad r_i = d_i.R_i(\bar{c}_i) \qquad \Gamma \vdash d_i : U_i \qquad U_i\texttt{<:}C_i \qquad \Gamma \vdash \mathit{roleOK}(C_i, R_i, \bar{c}_i, C)}{\Gamma \vdash \texttt{new } C(\bar{e}) \oplus \bar{r} : \bar{C}.\bar{R} :: C} \qquad \text{(T-NEW)}$$

$$\frac{\Gamma(\bar{x} : \bar{C}.\bar{R} :: \Gamma(\bar{x}), \bar{y} : \Gamma(\bar{y})/\bar{C}.\bar{R}) \vdash e_0 : T \qquad r_i = d_i.R_i(\bar{c}_i) \qquad \Gamma \vdash \bar{x} : \bar{S} \qquad \Gamma \vdash \bar{d} : \bar{U} \qquad \bar{U} \texttt{ <: } \bar{C} \qquad \Gamma \vdash \mathit{roleOK}(C_i, R_i, \bar{c}_i, S_i) \qquad \Gamma \vdash \bar{y} : \bar{V} \qquad \Gamma \vdash \mathit{unbindAllowed}(V_i, \bar{C}.\bar{R})}{\Gamma \vdash \texttt{bind } \bar{x} \texttt{ with } \bar{r} \texttt{ from } \bar{y}\ \{\ \bar{x}\bar{y}.e_0\ \} : T} \qquad \text{(T-BIND)}$$

Figure 14: Expression typing

of the method invocation m on a class instance $\texttt{new } C(\bar{e})$, written as $\mathit{mbody}(m, \texttt{new } C(\bar{e}))$, is also defined in a similar manner.

Other auxiliary definitions regarding binding operations are shown in Figure 13. The predicate $\mathit{bindable}(D.R, C)$ checks whether or not an instance of C can be bound with an instance of $D.R$. This predicate returns true if C is a subtype of $D.R$'s required interface $\{\ \bar{M}_I\ \}$. Finally, the predicate $\mathit{unbindAllowed}(T, \bar{C}.\bar{R})$ checks whether all the role types contained in T are also members of $\bar{C}.\bar{Q}$.

3.3 Typing

$$\frac{\bar{x} : \bar{T}, \texttt{this} : C \vdash e_0 : T_0 \qquad \texttt{class } C\ \{\ \cdots\ \} \qquad \bar{C} \in \mathit{dom}(CT) \qquad \bar{R} \texttt{ OK IN } \bar{C}}{T_0\ m(\bar{T}\ \bar{x})\ \{\ \texttt{return } e_0;\ \}\ \texttt{OK IN } C} \qquad \text{(T-METHOD)}$$

$$\frac{\bar{x} : \bar{T}, \texttt{thisC} : C, \texttt{this} : C.R, \texttt{super} : \{\ \bar{M}_I\ \} \vdash e_0 : T_0 \qquad \texttt{class } C\ \{\ \cdots\ \bar{A}\ \} \qquad \bar{C} \in \mathit{dom}(CT) \qquad \bar{R} \texttt{ OK IN } \bar{C} \qquad \texttt{role } R\ \{\ \bar{M}_I;\ \}\{\ \cdots\ \} \in \bar{A}}{T_0\ m(\bar{T}\ \bar{x})\ \{\ \texttt{return } e_0;\ \}\ \texttt{OK IN } C.R} \qquad \text{(T-RMETHOD)}$$

$$\frac{\bar{M} \texttt{ OK IN } C.R}{\texttt{role } R \texttt{ requires } \{\ \bar{M}_I\ \}\{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\ \texttt{OK IN } C} \qquad \text{(T-ROLE)}$$

$$\frac{\bar{M} \texttt{ OK IN } C \qquad \bar{A} \texttt{ OK IN } C}{\texttt{class } C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\ \}\ \texttt{OK}} \qquad \text{(T-CLASS)}$$

Figure 15: Well-formed definitions

The typing rules for FEJ expressions are shown in Figure 14. An environment Γ is a finite mapping from variables to types, written as $\bar{x} : \bar{T}$. The typing judgment for expressions has the form $\Gamma \vdash e : T$, read as "in the environment Γ, expression e has type T."

The rules are syntax directed, with one rule for each form of expressions. The typing rules for method invocations and class instance creations check whether each actual parameter has a type of the corresponding formal parameter. The typing rule for class instance creations also checks that each role instance bound with the class instance is well-formed (i.e., each actual parameter for the role instance creation has a type of the corresponding formal parameter) and the class C is a subtype of each type of role instance.

The rule T-BIND is complicated. By $\Gamma(\bar{x} : \bar{T})$, we imply an environment that can be obtained by updating all the types of $\bar{x}$ contained in Γ to the corresponding types $\bar{T}$, respectively. By $T/C.R$, we imply a type generated by removing $C.R$ from T, if T is a type of the form $\bar{C}.\bar{R} :: C$ and $C.R$ is contained in $\bar{C}.\bar{R}$. We write $\bar{x} : \bar{C}.\bar{R} :: \Gamma(\bar{x})$ as a shorthand for $x_1 : C_1.R_1 :: \Gamma(x_1), \cdots, x_n : C_n.R_n :: \Gamma(x_n)$, and $\bar{y} : \Gamma(\bar{y})/\bar{C}.\bar{R}$ as a shorthand for $y_1 : \Gamma(y_1)/C_1.R_1, \cdots, y_n : \Gamma(y_n)/C_n.R_n$. Thus, the first hypothesis of T-BIND indicates that under the updated type environment, the body of `bind` expression e_0 has type T. Then, this rule inspects the role names of $\bar{r}$ and checks whether the types of $\bar{x}$ are compatible with those roles. It also checks whether role unbinding of $\bar{C}.\bar{R}$ is allowed for $\bar{y}$.

Typing rules for method declarations, role declarations, and class declarations are shown in Figure 15. The type of the body of a method declaration is a subtype of the return type. The special variable `this` is bound in every method declaration, and for every method declaration in roles, variables `thisC` and `super` are also bound; the type of `thisC` is its enclosing class, and the type of `super` is its required interface. A role declaration is well-formed if all the methods declared in that role are well-formed, and a class declaration is well-formed if all the methods and roles

$$\frac{\mathit{fvalue}(f, \texttt{new}\ C(\bar{v}) \oplus \bar{r}) = w}{(\texttt{new}\ C(\bar{v}) \oplus \bar{r}).f \longrightarrow w} \quad \text{(R-FIELD)}$$

$$\frac{v = \texttt{new}\ C(\bar{v}') \oplus \bar{r} \qquad \mathit{mbody}(m, \bar{r}) \text{ is undefined} \qquad \mathit{mbody}(m, \texttt{new}\ C(\bar{v}')) = x.e}{v.m(\bar{v}) \longrightarrow [\bar{v}/\bar{x}, v/\texttt{this}]e} \quad \text{(R-INVK)}$$

$$\frac{v = \texttt{new}\ C(\bar{v}') \oplus \bar{r} \qquad \bar{r} = \bar{r}_1, w.R(\bar{w}), \bar{r}_2 \qquad \mathit{mbody}(m, \bar{r}) = \bar{x}.e, w.R(\bar{w}) \qquad \mathit{cp}(v) = \texttt{new}\ C(\bar{v}') \oplus \bar{r}_2}{v.m(\bar{v}) \longrightarrow [\bar{v}/\bar{x}, v/\texttt{this}, w/\texttt{thisC}, \mathit{cp}(v)/\texttt{super}]e} \quad \text{(R-RINVK)}$$

$$\texttt{bind}\ \bar{v}\ \texttt{with}\ \bar{r}\ \texttt{from}\ \bar{w}\ \{\bar{x}\bar{y}.e_0\} \longrightarrow [(\bar{v} \oplus \bar{r})/\bar{x}, (\bar{w} - \bar{r})/\bar{y}]e_0 \quad \text{(R-BIND)}$$

$$\frac{e \longrightarrow e'}{e.f \longrightarrow e'.f} \quad \text{(CR-FIELD)}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.m(\bar{e}) \longrightarrow e_0'.m(\bar{e})} \quad \text{(CR-INVK)}$$

$$\frac{e_i \longrightarrow e_i'}{v.m(\cdots, e_i, \cdots) \longrightarrow v.m(\cdots, e_i', \cdots)} \quad \text{(CR-INVK-ARG)}$$

$$\frac{e_i \longrightarrow e_i'}{\texttt{new}\ C(\cdots, e_i, \cdots) \longrightarrow \texttt{new}\ C(\cdots, e_i', \cdots)} \quad \text{(CR-NEW)}$$

Figure 16: Dynamic semantics

declared in that class are well-formed.

3.4 Dynamic semantics

The reduction rules of FEJ are shown in Figure 16. We use a standard call-by-value operational semantics. There exist four computation rules: R-FIELD, R-INVK, R-RINVK, and R-BIND. The rest of the rules formalize the call-by-value strategy. The reduction relation is of the form $e \longrightarrow e'$, read "expression e reduces to expression e' in one step."

For the rule R-FIELD, the field f is searched in all the role instances $\bar{r}$ bound with the class instance $\texttt{new}\ C(\bar{v})$. There exist two rules for method invocation: one is for method invocation declared in a class instance (rule R-INVK), and the other is for method invocation declared in a role instance (rule R-RINVK). The method invocation reduces to the expression of the method body, substituting all the formal parameters $\bar{x}$ with the argument values $\bar{v}$ and the special variable `this` with the receiver of method invocation. Furthermore, in R-RINVK, the special variables `thisC` and `super` are also substituted with corresponding values; `thisC` is substituted with the enclosing class instance returned by *mbody*. By $\mathit{cp}(v)$, we imply a fresh value whose class name and arguments for its instance creation are identical to v. In R-RINVK, a new value copied from the receiver of method invocation is created and replaced with `super`. The binding role instances for $\mathit{cp}(v)$ are also changed so that $\mathit{cp}(v)$ is bound with role instances that exist on right-hand side of $w.R(\bar{w})$ (role instance returned by *mbody*), which formalizes the method combination mechanism for role instance compositions.

The `bind` expression reduces to its body. It substitutes the free variables $\bar{x}$ and $\bar{y}$ with values appearing in the `bind` clause and `from` clause, respectively. By $v \oplus r$, we imply a value obtained by adding r to the left-most side of v's binding roles $\bar{r}$. By $v - r$, we imply a value obtained by removing r from the v's binding roles. We write $(\bar{v} \oplus \bar{r})/\bar{x}$ as a shorthand of $(v_1 \oplus r_1)/x_1, \cdots, (v_n \oplus r_n)/x_n$. Similarly, we write $(\bar{w} - \bar{r})/\bar{x}$ as a shorthand of $(v_1 - r_1)/x_1, \cdots, (v_n - r_n)/x_n$. Thus, in the resulting expression, roles $\bar{r}$ are removed from $\bar{w}$ and added to $\bar{v}$. Because FEJ is a purely functional calculus, both binding operations (creating a new role instance) and context activation (activating the role instance) are expressed in one reduction rule.

3.5 Property

We show the property of FEJ, namely, every well-typed expression evaluates to a value. In this section, we only present a series of theorems indicating FEJ type soundness. We provide proofs in the full version of this paper[5].

THEOREM 3.1 (SUBJECT REDUCTION). *If* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : S$ *for some* $S <: T$.

THEOREM 3.2 (PROGRESS). *If* $\Gamma \vdash e : T$ *and there exist no* e' *such that* $e \longrightarrow e'$, *then* e *is a value.*

THEOREM 3.3 (FEJ TYPE SOUNDNESS). *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow^* e'$ *with* e' *a normal form, then* e' *is a value* v *with* $\emptyset \vdash v : S$ *and* $S <: T$.

4. RELATED WORK

We have overviewed the main features of NextEJ and compare it with those of EpsilonJ. Besides the object adaptation mechanism of EpsilonJ, NextEJ provides the feature of context activation inspired by COP languages. However, context activation in NextEJ is slightly different from that of COP languages. While COP languages provide methods for realizing behavioral variability with respect to some contextual information, NextEJ puts more emphasis on acquiring and activating *new* behaviors that are not provided by the original class.

The Epsilon model, on which this work is based, is related to aspect-oriented programming (AOP). AOP has a useful feature in that it enables one to add aspects dynamically as well as statically [29]. One of the main AOP languages is AspectJ [30], a Java-based AOP language. The main objective of writing aspects is to deal with cross-cutting concerns. This implies that there 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 for 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.

[5] `http://www.l.u-tokyo.ac.jp/~kamina/nej-full.pdf`

The Epsilon model is also related to feature-oriented programming (FOP) in that both approaches provide a method to modularize and compose features. FeatureC++ is an FOP extension for the C++ programming language that provides a method for composing features statically as well as dynamically [3, 39]. In FeatureC++, a developer can select a composition method from static binding or dynamic binding when composing a product. Context-activation mechanism is not considered in FeatureC++.

In CaesarJ, a Java-based AOP language, an aspect can be deployed and undeployed at any time [4]. This feature is similar to context activation scope in NextEJ. However, in NextEJ, a context is activated at the binding time. On the other hand, in CaesarJ, the binding is specified in the *binding class*. Although CaesarJ's binding class and methods for aspect deployment provide much flexibility for aspect composition and activation, NextEJ provides a more simple and flexible basis for object adaptation and context activation mechanism by specifying objects and contexts at the binding time.

ObjectTeams [21] also has a similar mechanism for 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* (opposite 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-class binding is declared at the class declaring time. NextEJ and EpsilonJ provide a more flexible method to express the relationship between roles and classes by using `requires` clause. Currently there exist no formalizations of ObjectTeams.

Delegation Layers [37] provide flexible object-based composition of collaborations. They combine the mechanism of delegation [31, 38] and virtual classes [32, 10], or Family Polymorphism [15]; roles may be represented by virtual classes, and a composition is instance-based using the delegation mechanism. However, this approach do not successfully represent the object adaptation described in this paper. For example, in NextEJ, the object can assume and discard a role dynamically, and even the discarded role may be assumed by another object and the state held in the role instance is taken over by the latter object.

powerJava [5] is a language similar to 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, a powerful mechanism for obtaining 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.

There are pieces of literature that formalize the feature of extending objects at run-time. Ghelli presented foundations for extensible objects with roles based on Abadi-Cardelli's object calculi [1], where coexistence of different methods introduced by incompatible extensions is considered [19]. Gianantonio et al. presented a calculus $\lambda Obj+$[20], an extension of λObj[17] with a type assignment system that allows a self-inflicted object extension that still statically catches the "message not found" errors. Drossopoulou et al. proposed a type-safe core language *Fickle*[14] that allows reclassification of objects, a mechanism for dynamically changing object's belonging classes that share the same "root" superclass. On the other hand, FEJ focuses on a foundation of object adaptation and context activation for Java-like languages (based on FJ). FEJ supports a notable feature of unbinding of role instances (removing role instances from values in `from` clause of `bind` expression).

Mixins [9] are related to roles in NextEJ in that they form partial definitions that can be reused with a number of classes that conform to the requirements of mixins. Several extensions of Java with mixins have been proposed [18, 2, 26]. Although mixin composition is originally performed at compile time, dynamic composition of mixins is also studied in a core calculus [7], and such kind of object-level inheritance is also studied as *wrappers* [11, 8].

5. CONCLUDING REMARKS

We have presented NextEJ, a type-safe alternative to Epsilon model programming language with the features of COP. It provides a method for naturally representing context-awareness at 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 for activation by the context activation scope. Furthermore, in NextEJ, multiple contexts can be activated at a time, and the behavior of the composite context generated by such multiple context activations is determined by the order of activations. A small core calculus FEJ provides a solid basis for assuring its type soundness.

Acknowledgments.

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

6. REFERENCES

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

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

[3] S. Apel, T. Leich, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE'05*, volume 3676 of *LNCS*, pages 125–140, 2005.

[4] Ivia Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173, 2006.

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

[6] Masahiro Bessho, Shinsuke Kobayashi, Noboru Koshizuka, and Ken Sakamura. A space-identifying ubiquitous infrastructure and its application for tour-guiding services. In *SAC 2008*, pages 1516–1621, 2009.

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

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

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

[10] 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.

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

[12] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *COP'09*, page No.10, 2009.

[13] 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.

[14] Sophia Drossopoulou, Ferruccio Damiani, and Mariangiola Dezani-Ciancaglini. Fickle: Dynamic object re-classification. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 130–149, 2001.

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

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

[17] K. Fisher, F. Honsell, and J.C̃. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

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

[19] Giorgio Ghelli. Foundations for extensible objects with roles. *Information and Computation*, (175):50–75, 2002.

[20] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In *OOPSLA'98*, pages 166–178, 1998.

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

[22] 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.

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

[24] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[25] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *OOPSLA'07*, pages 113–132, 2007.

[26] 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.

[27] 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.

[28] Tetsuo Kamina and Tetsuo Tamai. Towards safe and flexible object adaptation. In *COP'09*, page No.4, 2009.

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

[30] 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.

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

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

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

[34] Supasit Monpratarnchai and Tetsuo Tamai. The implementation and execution framework of a role model based language, EpsilonJ. In *SNPD'08*, pages 269–276, 2008.

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

[36] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. *J&*: Nested intersection for scalable composition. In *OOPSLA'06*, pages 21–35, 2006.

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

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

[39] Marko Rosenmüller and Norbert Siegmund. Code generation to support static and dynamic composition of software product lines. In *GPCE'08*, pages 3–12, 2008.

[40] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(3):285–331, 2008.

[41] 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.

[42] 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.