The Design and Implementation of a Role Model Based Language, *EpsilonJ*

Supasit Monpratarnchai
Tamai Tetsuo
The University of Tokyo
{supasit,tamai}@graco.c.u-tokyo.ac.jp

In the social reality, objects communicate with each other by means of assuming roles to establish collaboration, and then can adaptively change their roles to obtain other interaction possibilities. To achieve the goal of supporting and realizing such object collaboration and adaptation in the object-oriented technology, especially in Java, a new adaptive role-based model *Epsilon* and a corresponding language *EpsilonJ* have been proposed. In this paper, we present the background of adaptive role-based models, and then focus on the design of this Epsilon model and its language. A program written in EpsilonJ must be translated into executable code to execute. We propose a translation scheme of mapping EpsilonJ syntax to the standard Java. With this translation scheme, we implemented a practical syntax translator as a preprocessor of EpsilonJ program, through lexical analysis and parsing. Evaluation shows that our translator can effectively perform transformation in high accuracy, and translated programs can be executed more efficiently than the existing implementation of EpsilonJ.

I. INTRODUCTION

Object orientation is a software engineering approach that models a system as a group of interacting objects. Each object represents some entities of interest and is characterized by its class, its states, and its behaviors. Several concepts are featured in object orientation such as modularity, encapsulation, inheritance, and polymorphism. With these concepts, many benefits are associated; e.g. reusability, modular architecture, increased quality, client/server applicability, etc., which drive object orientation to become a leading paradigm in programming languages, knowledge representation, design and modeling, and database.

The data abstraction principle of object orientation can be compared with the interaction of objects in the real world; e.g. the same operation of turning-on a device is implemented in different manners inside different kinds of devices, depending on their functionalities. The philosophy behind object orientation however rests on the assumption in which the attributes and operations of objects are 1) *objective*, i.e. they are the same whatever the object interacting with them is, unless this interacting object is passed as an explicit parameter, and 2) *independent* from the interaction with another object (*session-less*) [1]. Accordingly, this view sometimes limits the usefulness and potentiality of object orientation in several ways [1, 2].

To alleviate those limitations at the level of programming constructs, particularly for security, usability, and adaptability reason, different operations should be offered to different callers by means of *access control*, and the state of the interaction with each caller should also be kept track by means of *sessions*. In the role-based access control model (RBAC) [3], *access rights* are associated with *roles* and callers, i.e. the callers of operations are made members of appropriate roles, thereby acquiring the roles' permissions. Moreover, sessions are designed as the mappings between a caller and an activated subset of roles which are assigned to the caller. With this effort to solve those limitations of object orientation, new general concept about *roles* was originally introduced, and the corresponding role-based models were also proposed.

II. BACKGROUND

A. Adaptive Role Model

In addition to the introduction of the role concept previously mentioned, let us consider the role from another point of view. In general, objects in social reality communicate with each other by means of assuming roles, i.e. the way an object can interact with other objects is provided by properties and abilities of its associating role. This concept is considered as *collaboration* between objects via roles. With this scheme, to obtain other interaction capabilities, objects also adaptively change their roles without losing their own identities, which is called the object *adaptation*.

Although the current object-oriented technology can efficiently represent objects in software modeling and programming languages, it is still difficult to describe such behavior based on object collaboration and adaptation scheme. This is because object orientation does not support the construction in which objects adaptively participate in or leave collaboration. Moreover, the current widely-used object-oriented modeling and programming languages do not directly support such flexibility and adaptability [4, 5]. Based on this motivation to conveniently support and realize the object adaptation and collaboration between objects, several *adaptive role-based models* have been proposed as computational models satisfying those requirements and overcoming limitations.

The notion of collaboration is well accepted in object-oriented design, represented by a set of objects together with interactions among them. In collaboration, a group of objects cooperate to perform a task or to maintain an invariant property,

and a role is a part of an object that fulfills its responsibilities in the collaboration [7]. The main objective of most role models is to support the description of this kind of collaboration, not just at the design level but also at the programming level. Thus, role-based model can be considered as *collaboration-based design*, in the sense that the model is designed by composing several roles collaborating with each other to achieve organizational goals.

B. Epsilon Model and EpsilonJ

Epsilon model is one of those adaptive role-based models sharing the same objectives in our scope of interest. This model aims to support description of collaboration between objects not only at the design level, but also at the object-oriented programming level, and also to devise a mechanism for object adaptation to environments [5].

In this model, collaboration is represented by a collaboration field called a *context*, featuring several roles. An object outside the context called a *player* can participate in this collaboration field or interact with other objects, by assuming one of the roles. With these model components, an interaction between players in the context can be performed only via role to achieve collaboration [6]. A *dynamic role binding* mechanism is used to let a player assume an appropriate role. The interacting player then acquires *affordances* [1] (properties and operations) provided by its associating role, after the binding by means of *role casting* and *delegation* mechanism, which will be described in the next section.

Figure 1 depicts the static structure and dynamic behavior of Epsilon model as described. Since each context represents its own independent concern, a separation of concerns (SoC) is explicitly supported by the model. The interactions between concerns are realized through players simultaneously assuming roles of different contexts.

Recently, role-based models have been realized and implemented as an extension of the existing programming languages, typically Java [2, 5]. Similarly, to realize the Epsilon model at the programming level, the corresponding language *EpsilonJ* was defined as the extension of Java with some new constructs. With EpsilonJ, both static structure and dynamic behavior in the Epsilon model can be explicitly represented. In the next section, we present an EpsilonJ language specification, illustrated with some examples to show how the Epsilon model components can be declared in the EpsilonJ syntax and be executed dynamically.

III. Language Syntax

Model Components Declaration

Contexts and players are declared like Java classes using the `context` and `player` keyword, respectively. Declaration of role is placed within the context similar to an inner class, using `role` keyword. Qualifier `static` is declared before the keyword `role` when there is exactly one instance of this role in the containing context instance. Although the context and roles can be declared like a class and inner classes in the traditional Java, roles in our model are more concrete than inner classes, and a coupling between a context and its roles is stronger than that of an outer and inner classes. The following code shows an example of context, roles, and player declaration of a business company.

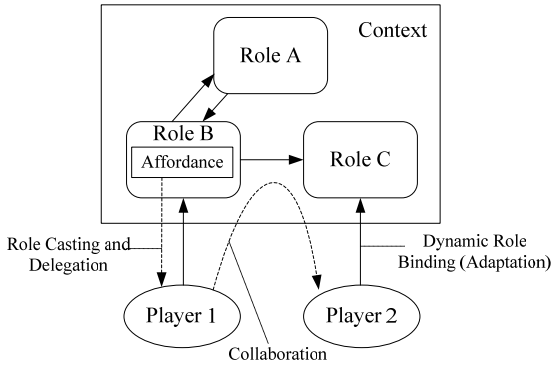

Figure 1. Static Structure and Dynamic Behavior of Epsilon Model.

```
 1 context Company {
 2   static role Employer{
 3     void pay(){ Employee.getPaid(); }
 4   }
 5   Role Employee requires { void deposit(int i); }{
 6     int save, salary;
 7     void getPaid(){
 8       save += salary; deposit(salary);
 9     }
10   }
11 }
12 player Person {
13   int money;
14   void deposit(int s){ money += s; }
15 }
```

Dynamic Role Binding

At a first time, a player instance can be dynamically bound to any role of a context by being sent as an argument to the `bind` for static role or `newBind` for non-static role of the targeting role, qualified with the context instance reference, as shown in line 19-20 of the following code. After that, a player instance can be re-bound to another role later, by just being sent to `bind` instance method, invoked by a new role as shown in line 21.

```
16 Company todai = new Company();
17 Person sasaki = new Person();
18 Person tanaka = new Person();
19 todai.Employee.newBind(sasaki);
20 todai.Employee.newBind(tanaka);
21 todai.Employer.bind(sasaki);
22 ((todai.Employer)sasaki).pay();
23 ((todai.Employee)tanaka).getPaid(100);
```

Role Casting and Delegation

Once a player is bound to a role, the player acquires an access to the role instance and thus can get the attributes or invoke the operations of the role, by being cast to the corresponding role as shown in line 22-23. This mechanism is called a *Role Casting*, and the mechanism of role method

invocation through the binding can be regarded as a kind of delegation, and is called *Role Delegation*. An explicit role casting notation is required to resolve ambiguity, because a player can be bound to multiple roles. Moreover, by indicating role casting, a static type checking gets possible.

Role Requirement Interface

There should be some interaction or coupling between the player and the role that are bound together, so that both state and behavior of the player can be affected by the binding. For this purpose, a way of defining an *interface* to a role is introduced, and it is used at the time of binding with a player, requiring the player to supply that interface. This mechanism is regarded as a *Role Requirement*, and can be declared using `requires` phrase as shown in line 5. With this requirement interface, role can be considered as a *double-faced interface*, which allows the connection of a player to a context [2]. The interface is double in which it specifies the methods that a player must offer for playing the role (role requirement) and the methods offered to the player playing the role (affordances).

IV. Proposed Translation Scheme

The source code written in EpsilonJ (*EpsilonJ program*) following the proposed syntax cannot be compiled and executed directly by a Java compiler in the standard JRE, because of the introduction of some new constructs. Thus, before the compilation and execution process, a translation is required as a preprocessor, to syntactically verify the EpsilonJ program and translate it into the pure standard Java. In the current version of EpsilonJ, the Java annotation feature is used to implement the EpsilonJ constructs [4], and so the external syntax is different from what we have described above, resulting in the significant runtime overhead. From the background introduced in the previous sections, to obtain the translation without encoding by annotations, in this paper we propose another approach for EpsilonJ implementation, by designing the translation scheme and compiling it as a practical syntax translator.

A. Dynamic Role Binding in the Role Implementation

Because a context, role, and player are equivalent to a common Java class, they are simply translated to `class`. To implement a dynamic binding, additional method `bind` is added to any role implementation, to bind a player to itself and bind itself to the *book-keeping structure* of a player [2] (which will be described later), as shown in the following code.

```
class Employer {
  EpsilonJ$Object _super;
  void bind(EpsilonJ$Object o){
    _super = o;
    o._setRoleInstance("Employer",this); }
  ... }
```

B. Role Re-Binding and Role Instance Repository

Furthermore, additional `newBind` method is also added to the context corresponding to each role, to implement re-binding feature, by creating a new role instance and binding a player to it (by invoking `bind`). To keep track of binding players, a context provides the repository of each role instance, i.e. a field for static roles, and a vector for non-static roles.

```
public class Company {
  Employer roles$Employer = null;
  Vector roles$Employee = new Vector();
  void newBind$Employer(EpsilonJ$Object o) {
    Employer tmp = new Employer();
    tmp.bind(o); roles$Employer = tmp; }
  void newBind$Employee(Employee$Super o) {
    Employee tmp = new Employee();
    tmp.bind(o); roles$Employee.add(tmp); }
...
```

C. The Iteration of Role Group

Since non-static role instances are translated into vectors (which are considered as *role groups*), whenever they invoke their own methods, the method invocation must be translated into `Iterator` abstract list interface, to iterate the method on all role instances. The following code shows a translation of `pay` in line 3 of the previous code.

```
void pay(){
  for(Iterator<Employee> i=roles$Employee.iterator();
    i.hasNext(); ) { i.next().getPaid(); }
}
```

D. Role Requirement Interface

For the requirement interface, it will be extracted from the role declaration, and translated into a separate interface in the context.

```
interface Employee$Super extends EpsilonJ$Object {
   void deposit(int s);
}
```

E. Player with a Book-Keeping Structure

Each person instance should possess its own *book-keeping structure* to keep track of the current role being bound, and also the history of its role binding. Thus, we add a new class extending the corresponding player class as follows, with the `HashMap` data structure as well as the operations to `set` and `get` role instances.

```
Class Person$EpsilonJ extends Person
  implements EpsilonJ$Object {
    HashMap _roles = new HashMap();
    public Object _getRoleInstance(String key){
      return _roles.get(key);
    }
    public void _setRoleInstance(String k, Object v){
      return _roles.put(k,v);
    }
}
```

To make this inheritance effective for all types of players, another corresponding top-most interface is also declared as follows.

```
Interface EpsilonJ$Object{
  public Object _getRoleInstance(String key);
  public void _setRoleInstance(String k, Object o);
}
```

F. *Role Binding, Casting, and Delegation*

Based on the translation scheme of EpsilonJ components described above, dynamic role binding, role casting, and role delegation are explicitly translated as the binding mechanism (implemented within the context) and book-keeping structure (implemented within the player) as the following example shows.

```
todai.newBind$Employer(sasaki);
todai.newBind$Employee(tanaka);
((Company.Employer)sasaki._getRoleInstance
                      ("Employer")).pay();
((Company.Employee)tanaka._getRoleInstance
                      ("Employee")).getPaid(100);
```

V. Translator Implementation

From the language syntax described in section III, we also derived the structure of concrete syntax which is expressed in the Extended Backus-Naur Form (EBNF) of the Context-Free Grammars (CFG), by modifying and extending that of Java syntax. Based on this syntax structure in EBNF together with the translation scheme introduced in the last section, we developed the practical EpsilonJ translator straightforwardly through lexical analysis and parsing using Java-Compiler Compiler tool (JavaCC) provided by Sun Micro-systems [8].

VI. Evaluation

To determine the powerfulness and potentiality of our approach, we conducted the evaluation in two aspects; the accuracy of translation and the execution efficiency of translated program. The accuracy of our translator is evaluated based on several test cases containing the EpsilonJ program generated in random sizes and patterns following the proposed EpsilonJ syntax. As an experimental result, 92.5% of these test cases were successfully translated by the EpsilonJ translator, i.e. their syntax conforms to that specified within the translator. By investigation throughout the program source code, we have found that several failed cases are due to the error of some Java syntax such as the repetition of identifiers, but there is no error related with the structure of EpsilonJ syntax. This failure is intended to be improved in future work.

In fact, any role-based programs can also be written in a traditional object-oriented programming language, without using specific object collaboration and adaptation constructs. Thus, to evaluate the efficiency of our EpsilonJ programs compared to those implemented in pure Java programming, another evaluation is also conducted. We designed some case studies, implemented into EpsilonJ programs, and get translated by our EpsilonJ translator. For each case, we also hand-coded a Java program which performs the identical functions. Both programs are then compiled and executed, measuring the overhead used for compilation and execution. Table I shows the compilation and execution time in milliseconds for both translated EpsilonJ programs and the traditional Java programs of three case studies.

TABLE I
Compilation and Execution Time

Case Study	EpsilonJ Program	Traditional Program
Business Company	4.11 ms	2.14 ms
Integrated System	4.92 ms	2.64 ms
Basic Printer	6.39 ms	3.08 ms

Although the result indicates that the translated EpsilonJ programs require more execution time than those of traditional Java, by approximately 2 times; however, these programs are more efficient than the current version of EpsilonJ implementation proposed in [4, 5], in which the execution overhead can be reduced by 5-10 times. By comparing to this execution time, the overhead of translation process in the first evaluation is not significant, as almost all test cases were translated in very short time, regardless of the input size.

VII. Discussion and Conclusion

We present our role-based model called Epsilon, and the corresponding language EpsilonJ with syntax specification. Then, we propose another approach to compile EpsilonJ programs by means of translation, performed by our newly developed EpsilonJ translator with high accuracy. The translation follows our proposed translation scheme for mapping the EpsilonJ program to the executable Java program. The evaluation shows that the EpsilonJ programs translated by our proposed approach of translation scheme, were executed more efficiently than the existing implementation approach of EpsilonJ.

Acknowledgment

The authors would like to thank *Tetsuo Kamina*, a member of our research group, for the inspiring idea and useful discussion on the model translation programming.

References

[1] M. Baldoni, G. Boella, and L. van der Torre, "Interaction among Objects via Roles: Sessions and Affordances in Java," *Symposium on Principles and practice of programming in Java, Vol.178*, pp. 188-193, 2006.

[2] M. Baldoni, G. Boella, and L. van der Torre, "Interaction between Objects in powerJava," *Journal of Object Technology, Vol.6, No.2, Special Issue OOPS Track at SAC 2006*, pp. 5-30, 2007.

[3] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," *IEEE Computer*, pp. 38-47, 1997.

[4] T. Tamai, N. Ubayashi, and R. Ichiyama, "An Adaptive Object Model with Dynamic Role Binding," *Proceedings of ICSE'05*, pp. 166-175, Missouri, USA, May, 2005.

[5] T. Tamai, N. Ubayashi, and R. Ichiyama, "Objects as Actors Assuming Roles in the Environment," *LNCS4408: Software Engineering for Multi-Agent Systems V*, Springer-Verlag, pp. 185-203, 2007.

[6] G. Boella and L. van der Torre, "A foundational ontology of organizations and roles," *Proceedings of DALT'06 workshop at AAMAS'06*, 2006.

[7] M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-based Designs," *Proceedings of ACM Conference OOPSLA'96*, pp. 359-369, 1996.

[8] JavaCC homepage by Sun Microsystems: https://javacc.dev.java.net/