

# The Implementation and Execution Framework of a Role Model Based Language, *EpsilonJ*

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

## Abstract

*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. To utilize this translator, we also propose an interactive framework prototype for EpsilonJ program development and execution, and developed as a web-based application tool, by deploying this EpsilonJ translator as a core component. 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.*

## 1. 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. Sessions are considered with more attention in the agent-oriented paradigm, which bases communication on protocols [4]. In this model, the state of the interaction between two agents is maintained in a session, and the interaction is allowed only by agents playing certain

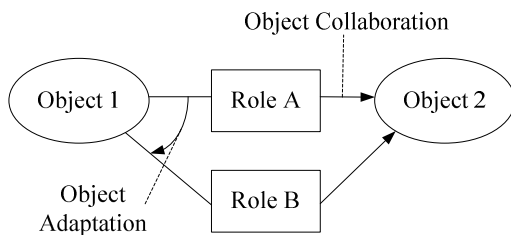
roles. However, the notion of role is rarely related with the notion of interaction session. Consequently, with the effort to solve those limitations of object orientation, new general concept about roles was introduced, and the corresponding role-based models were also proposed.

## 2. Background and Motivation

### 2.1. Adaptive Role Models

In addition to the introduction of the role concept mentioned above, 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*. Figure 1 shows the object communication with the collaboration and adaptation concept described above.

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 [5, 6]. 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.



**Figure 1. Communication between objects via role in the social reality**

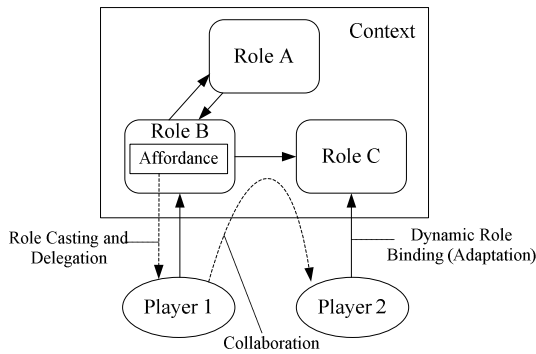
The notion of collaboration is well accepted in object-oriented design, represented by a set of objects together with interactions among them. In collaboration, a group of objects cooperate to perform a task or to maintain an invariant property, and a role is a part of an object that fulfills its responsibilities in the collaboration [8]. 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.

### 2.2. 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 [6].

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 [7]. 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 2 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 multiple players simultaneously assuming roles of different contexts. And from this separation of concerns aspect, contexts including roles can be considered as the independent reuse component. As we will see later, contexts and roles are implemented as a first class constructs, so that the collaboration patterns can be reused directly as a programming level components. Enabling separation of concerns and reusability then can be considered as some characteristics of Epsilon model.



**Figure 2. Static structure and dynamic behavior of the Epsilon model**

Recently, role-based models have been realized and implemented as an extension of the existing programming languages, typically Java [2, 6]. 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.

### 3. Language Syntax

#### 3.1. 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.

```

1 context Company {
2   static role Employer {
3     void pay() { Employee.getPaid(); }
4   }
5   Role Employee requires
6     { void deposit(int i); } {
7     int save, salary;

```

```

8     void getPaid(){
9       save += salary; deposit(salary);
10    }
11 } }
12 player Person {
13   int money;
14   void deposit(int s) { money += s; }
15 }

```

#### 3.2. 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. Note that a player may be bound to multiple roles in the same or different contexts at any time.

```

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);

```

#### 3.3. 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.

#### 3.4. 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 5-6. 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)

## 4. 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 [5], 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.

### 4.1. 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);
    }
    ...
}
```

### 4.2. Role Re-binding and Role 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);
    }
    ...
}
```

### 4.3. 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();
    }
}
```

### 4.4. 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);
}
```

### 4.5. 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);
}

```

#### 4.6. 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.

```

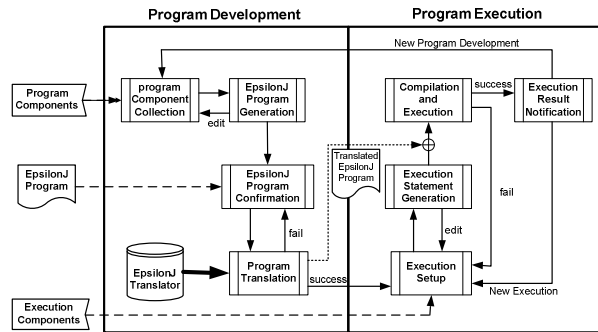
today.newBind$Employer(sasaki);
today.newBind$Employee(tanaka);
((Company.Employer)sasaki._getRoleInstance
    ("Employer")).pay();
((Company.Employee)tanaka._getRoleInstance
    ("Employee")).getPaid(100);

```

### 5. Interactive Framework

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 [16].

To much more utilize this EpsilonJ translator and assist the developers, we also designed and developed the interactive framework prototype to support the EpsilonJ program development and execution as the web-based application, deploying this translator as a



**Figure 3. Architecture and mechanism of the interactive framework for the EpsilonJ program development and execution**

core engine. Figure 3 shows the architecture and mechanism of the framework.

The framework is divided into two parts, the program development and program execution. The program development is to create the EpsilonJ program of the user-specified case, by gathering several model components from a user, automatically generating the EpsilonJ program source code, syntactically analyzing and translating it into the Java executable source code by utilizing the embedded EpsilonJ translator.

The program execution is to simulate the developed EpsilonJ program with automatic compilation and execution. First, several execution statements (similar to those in main method of Java) performing object instantiation, role binding, role casting and delegation will be collected and combined with the EpsilonJ program from program development framework. Then the complete EpsilonJ program will be automatically compiled and executed.

A work flow of our framework in Figure 3 can be described in details as follows. For the program development, first, developer specifies each program components as described in section 3 in several levels. In each level, those components will be collected and incrementally composed (based on the proposed syntax) to generate EpsilonJ program source code. The developer can edit the specified contents of each component until all components are completely defined. Once the program is generated, it will be shown to confirm its correctness and completeness. If there is any mistake found, the developer can modify this program directly. As the feature, instead of specifying each program component manually, a text file containing already written EpsilonJ program can also be loaded into an application as well. Once the program is confirmed, it will be syntactically analyzed by the EpsilonJ translator to check the syntax

correctness. If it succeeded, the EpsilonJ program will be immediately translated (parsed) into the executable Java program source code.

The program execution begins with developer assigns the execution setup components, by specifying context and player instantiation, role binding, role casting and delegation, etc. Similarly, those components will be collected and composed to generate the EpsilonJ program of the execution. If all execution statements are confirmed, they will be slightly translated into the Java program following the translation scheme for the role binding and delegation. Then it will be integrated into the translated program from the program development framework, to establish a complete executable Java program. Instantly after the executable program is established, it will be automatically compiled and executed by JRE, and the actual program execution takes place. If it is successfully compiled and executed, the execution result will be shown as an output. This terminates the program development and execution process cycle. Later, the developer can modify current program execution, create a new one for existing EpsilonJ program, or restart the application with new EpsilonJ program development.

## 6. Evaluation

To determine the powerfulness and potentiality of our implemented translator and developed application framework, 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 1 shows the compilation and execution time in milliseconds for both translated EpsilonJ programs and the traditional Java programs.

Although the result indicates that the translated EpsilonJ programs require more execution time than those of traditional Java, by approximately 2 times; these programs are more efficient than the current version of EpsilonJ implementation proposed in [5, 6], in which the execution overhead can be reduced by 5-10 times, since it requires more overhead about 10-20 times than the traditional program. 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.

However, the evaluation of the proposed EpsilonJ language is not covered in this paper since we adapted the syntax from the previous work [5, 6]. The evaluation regards main features of the EpsilonJ language itself is intended to be given in the further syntax and semantic improvement.

## 7. Related Work

Originally, the design of Epsilon model is inspired by the realization on multiple inheritances of mixin construct [10]. Mixins in McJava [9] can be composed with objects that supply their required interface, thus the role and object binding can be effectively simulated. However, the mixin-object composition is static, which differs from the dynamic characteristics of binding operation in EpsilonJ.

In Caesar model [11], an aspect interfaces are decoupled into an aspect implementation and aspect binding. The aspect interface is called ACT (Aspect Collaboration Interface) with multiple mutually recursive nested types. This ACI is identically correspondent to the context and role as in Epsilon model, where a role corresponds to a nested type.

**Table 1. 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 this model can represent the context and role as in Epsilon, the collaboration between roles cannot be expressed explicitly as it must be given only by the implementation.

Multi-Dimensional Separation of Concerns (MDSoc) model [12] is relatively closed to Epsilon, and its corresponding language Hyper/J [13] is also implemented based on Java. The general idea of MDSoC is quite similar to Epsilon as describing the collaboration fields in a separate dimensions and defining classes by consolidating roles in those dimensions. A clear difference of Hyper/J and EpsilonJ is that the composition of the features in Hyper/J is on the class-to-class basis at compile time, while the composition of roles with player objects in EpsilonJ is on the instance-to-instance basis at runtime.

ObjectTeam/Java [14] is another new language sharing similar objectives with EpsilonJ, but some basic design concepts look relatively different. For example, although there is a notion of role instance in ObjectTeam/Java, the combination of a role class and a base class (player class in EpsilonJ) is statically fixed and only attachment/detachment of a role instance to the object can be changed dynamically.

The approach of powerJava [1, 2, 15] is quite new and similar to EpsilonJ, both in the basic concept of role and the language implementation. As the limitation of current object orientation, the objects consider attributes and operations as being objective and independent from the interaction. But in powerJava, interaction with an object always passes through a role played by another object manipulating it. The advantage is that roles allow to define operations whose behavior changes depending on the role and the requirements it imposes, and to define session-aware interaction, where the role maintains the state of the interaction with an object [3]. These concepts are addressed to solve some limitations of object orientation, and also implicitly integrated in Epsilon model.

## 8. 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, which is the main contribution of this research. To utilize this translator, we also designed and developed the

interactive framework for convenient EpsilonJ program development and automatic execution as the application tool. 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.

In our EpsilonJ translator, only syntactic errors can be detected, i.e. the translated EpsilonJ program is implicitly assumed to be semantic error-free in the execution. As the future work, to eliminate this limitation while keep reducing practical runtime errors, all essential key features of Epsilon model are intended to be formalized into an  $\epsilon$  core language [9], which provides safer alternative to language design. This core language has proved to be type-sound in which a well-typed EpsilonJ program does not produce any runtime errors.

## 9. Acknowledgement

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

## 10. 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] J. Ferber, O. Gutknecht, and F. Michel, "From agents to organizations: an organizational view of multi-agent systems," *LNCS 2935: Proceedings of AOSE'03*, pp. 214-230, Springer, Berlin, 2003.
- [5] 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.
- [6] 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.
- [7] G. Boella and L. van der Torre, "A foundational ontology of organizations and roles," *Proceedings of DALI'06 workshop at AAMAS'06*, 2006.
- [8] M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-based Designs,"

- Proceedings of ACM Conference OOPSLA '96*, pp. 359-369, 1996.
- [9] T. Kamina and T. Tamai, "Flexible Object Adaptation for Java-like Languages," 2004.
- [10] G. Bracha and W. Cook, "Mixin-Based Inheritance," *Proceedings of the European Conference OOPSLA '90*, pp. 303-311, 1990.
- [11] M. Mezini and K. Ostermann, "Conquering Aspects with Caesar," *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD '03)*, pp. 90-99, Boston, March, 2003.
- [12] H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns in Hyperspace," *ECOOP '00 workshop on Aspects and Separation of Concerns*, France, June, 2000.
- [13] H. Ossher and P. Tarr, "Hyper/J<sup>TM</sup>: Multi-Dimensional Separation of Concerns for Java<sup>TM</sup>," *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, 2001.
- [14] S. Herrmann, "Programming with Roles in ObjectTeams/Java," *The German Federal Ministry for Education and Research*, 2005.
- [15] M. Baldoni and G. Boella, "Roles as a Coordination Construct: Introducing powerJava," *Proceedings of the 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord '05)*, Vol.150, No.1, pp. 9-29, March 2006.
- [16] JavaCC homepage by Sun Microsystems: <https://javacc.dev.java.net/>