Weaving in Role-Based Aspect-Oriented Design Models

Shin NAKAJIMA
NII and PRESTO, JST
2-1-2 Hitotsubashi, Chiyoda-Ku
Tokyo, 101-8430 JAPAN
EMAIL: nkjm@nii.ac.jp

Tetsuo TAMAI
The University of Tokyo
3-8-1 Komaba, Meguro-Ku
Tokyo, 153-8902 JAPAN
EMAIL: tamai@acm.org

Abstract

Aspect-oriented modeling deals with cross-cutting concerns at early stages of the software development. Weaving involves highly abstract aspect descriptions, and it is necessary to take into account application specific constraints. We adopt a role-based aspect-oriented modeling method and define a notion of the aspect weaving in a systematic way. We further discuss how we use Alloy, a lightweight formal specification language and analysis tool, for the precise model descriptions and verification.

1 Introduction

Aspect-oriented modeling is an important approach to reducing complexity of software design [5]. Primary concerns, identified by means of some criteria, are not always orthogonal to each other. Some *residuals* are left spread over the primary ones. Ways to deal with such cross-cutting concerns are needed [11]. Object-oriented programming has been successful to represent primary concerns in the form of class definitions, but other concerns cross multiple classes. And aspect-oriented programming provides an idea of an aspect being a first-class element to describe such concerns scattered over classes. AOP compiler or other tool provides mechanisms for weaving aspects with primary concerns automatically [10][12][23].

Aspect is quite important in early stages of the software development. It is a common exercise to analyze a complex system from multiple viewpoints [15] to identify aspects that are (mostly) independent with each other. Further, the traceability becomes clear when we keep track of how a specific aspect identified at the early stage is implemented. Aspect-oriented modeling covers many activities at the early stages. A. Rashid et al [16] focus on identifying non-functional properties as aspects in software requirements and propose a set of informal composition rules. E. Baniassad and S. Clarke [1] discuss a method on how concerns are identified and separated in the requirement analysis phase. G. Georg et al [6] adopt the idea of the aspect to have clear design dealing with cross-cutting concerns.

Since aspect-oriented modeling (AOM) is a method at the early stages of the software development, model descriptions are more abstract than programs. Thus, it is hard to have an automatic weaver in AOM as compared with the case of AOP. Weaving involves highly abstract aspect descriptions, and it is necessary to take into account application specific constraints. The model transformation can only be done by human designers who know the appliation semantics. Since the model transformation is sometimes done with the human intervention, some verification is necessary between the model descriptions before and after weaving. A rigorous method for specifying the weaving operation as well as writing aspect model descriptions is called for.

In this paper, we first introduce an idea of applying a role-based modeling method [17][22] to AOM, and define a notion of the aspect weaving in a systematic way. We then propose to use Alloy [8][9], a lightweight formal specification language and analysis tool, for the precise model descriptions and verification. Technically, we discuss how a role-based aspect model is described in Alloy and how the aspects are weaved. We demonstrate our idea by using an example case dealing with the security aspect.

2 Aspect-Oriented Modeling

2.1 Aspect and Role

As a start of the discussion, we show here a *Logging* aspect, an example cross-cutting concern often used in the AOP literatures [11]. When we use the Logging framework of the Java platform class library, a method call such as the one below is used.

```
logger.log(Level.SEVERE, ‘‘fatal’’, e)
```

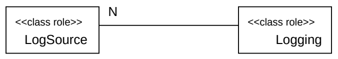

Figure 1. Logging Aspect

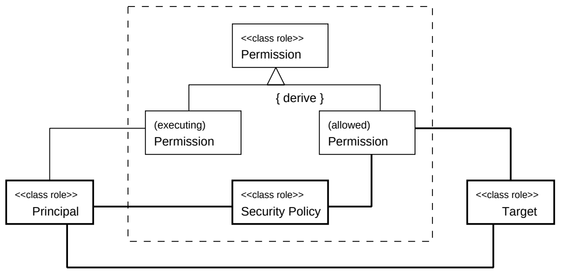

Figure 2. Access Control Aspect

Such method calls should be inserted in all the method bodies where we want to record the access logs. Although the logging is a conceptual unit focusing on a particular functionality, the actual code fragments are scattered over classes. Aspect/J [10], for example, provides a first-class language element to describe the logging aspect, which frees us from writing scattered codes.

The *Logging* aspect in Figure 1, however, is simple when we view the design in terms of the role-based modeling method [17]. All the classes to be logged have `LogSource` role, and the role is related to another role `Logging`.

We consider next an *Access Control* aspect in Figure 2. It is an aspect taken from the system security domain, a typical example cross-cutting concern [4][6]. When `Principal` makes an access of either read or write to `Target`, the aspect illustrates that each access is checked against given `SecurityPolicy`, which is basically a collection of `Permission`. The access checking is done by comparing the `ExecutingPermission` with `AllowedPermission`.

The *Access Control* aspect is a bit complicated than the *Logging* aspect. It involves several roles that have some interactions between them. We need a method focusing on the collaborative behavior of roles, and the role-based approach is a good candidate to deal with such model descriptions [17][21].

Role-based modeling and aspect-oriented modeling share some common notion [22]. When we identify a class as a primary concern of a system, the idea of the role provides alternative viewpoints to study the function and behavior of the system. A role-based modeling method is focused on finding appropriate roles for charactering an object [17]. And it can also be used as a method to identify collaborative behavior [19]. On the other hand, analyzing with roles in AOM puts emphasis on identifying aspects themselves. K. B. Graversen and K. Osterbye [7] propose a method on how an identified aspect is translated into `advice` in AspectJ. The method is supposed to be applicable to an aspect such as the `Logging`, which is rather independent. For the aspects more on the interaction, G. Georg et al [6] propose a UML-based diagram notation to incorporate `role` stereotype, and applies the role-based modeling method for the case of the security aspect.

In this paper, we adopt the role-based modeling method similar to [6] in that

Aspect = Roles and their Interaction,

and sometimes use UML-like diagrams for illustrative purposes.

Since the proposed method is based on object-oriented modeling, the design model is a description of either aspect or class [1]. And the role-based approach here sees a role to be a kind of *elementary particle* to constitute all the design models. In other words, role provides a uniform basis to represent both aspect and class. It depends on how the designer recognizes the system whether a specific set of roles and their interactions is regarded as either aspect or class.

2.2 Weaving in AOM

Aspect-orinted modelilng (AOM) is different from aspect-oriented programming (AOP) in that it should deal with model descriptions at an abstract level. Weaving in AOM is also different from the case in AOP. In AOP, an aspect is a first-class language element, and AOP compiler or other tool provides mechanisms for automatically weaving aspects with primary concerns [10][12][23].

In the role-based aspect-oriented modeling in this paper, the model description, of either aspect or class, is centered around the structural relationships between the identified roles. In other words, both aspect and class are not so much different in that they are represented in terms of the roles and their interactions. It implies that two seemingly different things, (1) weaving aspect with class and (2) weaving two aspects, can be discussed in a uniform way. In AOP, only the former weaving is considered while the latter one is equally important in AOM because we sometimes want to deal with abstract aspects at the design level without considering classes.

Weaving is essentially a model transformation and similar to *mixins* in object-oriented programming [2]. Mixin of two flavours, or classes, results in a new *fat* flavour. In a similar manner, weaving two model descriptions results in a new model description that has all the roles in the original models. And the new roles can be obtained through the two steps. Namely, first we identify a role in the first aspect with

[1]For simplicity, in this paragraph, *class* refers to a primary concern as compared with *aspect*.

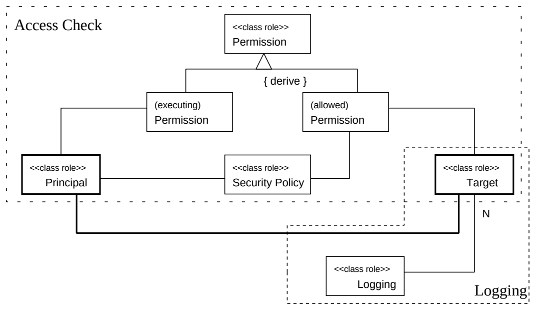

Figure 3. Weaving Two Aspects

another role in the second one, and then the identified roles are *merged* to become a *fat* role.

Since we deal with abstract design, the model transformation of weaving in AOM is done manually. Thus, there are many places to contaminate model descriptions by introducing defects unintentionally. We have to make sure that the weaving definitely produces a consistent new model description. It is desirable to exclude such role merging operations that introduce some inconsistency between them. Further, a property that an aspect originally has may be violated after weaving an aspect with a model description of another aspect or a class.

Unfortunately the diagram such as UML is not adequate for formal analysis of the model descriptions. It is necessary to have a rigorous method for specifying weaving as well as writing aspect model descriptions.

2.3 Weaving and Role Merging

Weaving in the role-based aspect-oriented modeling is essentially to make two corresponding roles merged. Since both aspect and class are described in terms of roles in our method, we present our discussion with an example of weaving two aspects.

Schematically, we obtain a model description in Figure 3 after weaving the Logging aspect (Figure 1) with the Access Control aspect (Figure 2). `Target` role in Figure 2 is merged with `LogSource` role in Figure 1 to become a new *fat* `Target` in Figure 3. We should take into account the application specific constraints. In general, roles within an aspect have various collaborative behavior with each other. Since the diagram notation only shows some structural relationships among the roles, it is not apparent from the diagram that the roles have further constraints. Actually, in UML, such constraints are supposed to be expressed by means of OCL, a textural constraint language.

We will give a definition of weaving and role merging in a systematic manner by using a simple notation. First, the simplest role merging is a case for a one-to-one mapping. Below, we show a description for a case of merging two roles R and P, where a role R is a constituent of an aspect $AspectA$ and P of $AspectB$.

OneToOneMerging at ⟨R, P⟩ is
    identify: R → P
with f(R, P)

The relationship $identify$ describes that an one-to-one mapping exists between the specified roles, R and P. The expression $f(R, P)$ specifies constraints on the arguments roles. The one-to-one mapping mentions that there is an instance of a role P for each instance of a role R. And $f(R, P)$ describes what properties the R and the corresponding P instance have in order that they satisfy the application specific semantics of the merging. The expression describes a constraint relationship on the role instances of the argument roles. It is just $true$ when the attributes do not have any significant meanings in the merging. Set-theorematic approach can be applied to the semantics of the role merging.

For the example case in Figure 3, we have what is shown below. The constraint part is written in English since its formal description needs rigorous definition of the roles involved. The fragment below is actually not a precise definition of the merging, and will be further refined afterwards. It is shown here only for an illustrative purpose [2].

OneToOneMerging at ⟨LogSource, Target⟩ is
    identify: LogSource → Target
with Logged-operation-representing-access-action.

We can generalize the above OneToOneMerging to Merging that involves more than one role in each aspect. Roles $\mathrm{R}_1 \ldots \mathrm{R}_N$ (N $\geq$ 1) are simultaneously merged with roles $\mathrm{P}_1 \ldots \mathrm{P}_M$ (M $\geq$ 1). It means that the relationships between R_i's as well as R_i's are *merged* with those for the case of P_j's. The relationship is described here as $\mathrm{R}_1 \rightarrow \ldots \rightarrow \mathrm{R}_N$.

Merging at $\langle \{\mathrm{R}_1 \ldots \mathrm{R}_N\}, \{\mathrm{P}_1 \ldots \mathrm{P}_M\} \rangle$ is
    identify: $(\mathrm{R}_1 \rightarrow \ldots \rightarrow \mathrm{R}_N) \rightarrow (\mathrm{P}_1 \rightarrow \ldots \rightarrow \mathrm{P}_M)$
with $\bigwedge_{i,j} \mathrm{f}_{i,j}(\mathrm{R}_i, \mathrm{P}_j)$

For a special case where N = 1 and M = 2, we have MergingOneAndTwo.

MergingOneAndTwo at $\langle \{\mathrm{R}_1\}, \{\mathrm{P}_1, \mathrm{P}_2\} \rangle$ is
    identify: $\mathrm{R}_1 \rightarrow (\mathrm{P}_1 \rightarrow \mathrm{P}_2)$
with $\mathrm{f}_{1,1}(\mathrm{R}_1, \mathrm{P}_1) \wedge \mathrm{f}_{1,2}(\mathrm{R}_1, \mathrm{P}_2)$

Intuitively, R_1 in AspectA is refined into P_1 and P_2 in AspectB, and thus the $identify$ relationship describes a mapping from R_1 to a relationship between P_1 and P_2 [3].

[2] Section 3.3 will show a precise description.
[3] Section 3.3 will show an example.

At this point, we can define the aspect weaving in terms of the merging just introduced. Since the role merging is an operation at a *microscopic* level, we consider that weaving two aspects consists of more than one role merging operation. Further, the weaving may introduce some extra constraints on the participant role instances. Since the merging operations are defined in a manual manner, there may be some inconsistency between the operations. In the bottomline, we have to verify that the weaving operation, consisting of more than one merging, is consistent.

Weaving (AspectA, AspectB) is
 Merging at $\langle\{R_1^1 \ldots R_N^1\}, \{P_1^1 \ldots P_M^1\}\rangle$
 $\vdots$
 Merging at $\langle\{R_1^K \ldots R_N^K\}, \{P_1^K \ldots P_M^K\}\rangle$
with Further-Constraints-Added

In summary, in the role-based aspect-oriented modeling method, the aspect weaving is basically a set of role merging operations, and it involves changes in the structural relationships in the model description.

3 Lightweight Analysis with Alloy

We present a case of using Alloy for the precise descriptions and verification of the model obtained by means of the role-based aspect-oriented modeling method.

3.1 Logging Aspect

In order to have rigorous Alloy description, we have to add some element roles to the model description illustrated in Figure 1. `LogData` is a logged data. Since each logged data is distinctive with each other, `sig LogData` has a unique `Id` value as its attribute. The first `fact` represents the uniqueness constraint that the two data are the same if their `id` attributes are equal. `AccessAction` represents an access event, and has similar `fact` for the uniqueness constraint. Further, it specifies that `LogData` is a faithful log of `AccessAction` if the `id` are the same.

```
sig LogData { id: Id }
fact{ all d1,d2: LogData |
            d1.id = d2.id => d1 = d2 }
sig AccessAction {  id: Id }
fact{
   all d: LogData |
      one a: AccessAction |
          a.id = d.id &&
          (some t: LogSource | a in t.done)
}
```

We introduce `LogState` for keeping track of the snapshot of the logging system. `LogSource` has two attributes: `tobe` maintains a set of `AccessAction` to be executed and `done` is a set of those having been executed. We introduce the two attributes in order to directly refer to the two states, a state before executing a specific `AccessAction` and another state after the execution. Clearly, the two attributes are exclusive, which is specified as the conditions in `fact`.

```
sig LogState {
  target:  set LogSource,
  state: Logging
}
sig Logging { logset : set LogData }
sig LogSource {
  done: set AccessAction,
  tobe: set AccessAction
}
fact { all t: LogSource | no(t.done & t.tobe) }
```

Then, the main logging function is a *state-transformer* on `LogState`, and shown in `fun execute` below. It states that the whole system is moving to `s2 LogState` after executing an `AccessAction` on the `s1 LogState`. Some book-keeping is necessary, but essentially `execute` adds a new `LogData` having a correspondance with the `AccessAction` to the system state.

```
fun execute(s1: LogState, a: AccessAction): LogState
{
  some t1: LogSource |
    t1 in s1.target &&  a in t1.tobe &&
    (one t2: LogSource |
        t2.done = t1.done + a &&
        t2.tobe = t1.tobe - a &&
        result.target = s1.target - t1 + t2) &&
    (one d: LogData |
        d.id = a.id &&
        (d not in s1.state.logset) &&
        result.state.logset = s1.state.logset + d)
}

run execute for 2
```

The `run` command ensures that the description is consistent and the intended behavior is satisfied.

3.2 Access Control Aspect

The example is shown in Figure 2, and we first introduce `Principal` and `Target`. The uniqueness constraint is imposed on the `name` attribute. Only the definition of `Principal` is shown here since `Target` can be defined similary.

```
sig Principal { name: PName }
fact{ all p1,p2: Principal |
          p1.name = p2.name => p1 = p2 }
```

`Permission` represents an access permission that consists of `Mode`, either `ReadMode` or `WriteMode`, and `Target`.

```
sig Permission { mode: Mode, target: Target }
sig Mode { }
static disj sig ReadMode extends Mode {}
static disj sig WriteMode extends Mode {}
```

Then, `Policy` is a mapping from `Principal` to `Permission`, which represents that specified accesses are allowed for the `Principal`. In particular, a system may have only one `Policy` called `SecurityPolicy`.

```
sig Policy {
   principals : set Principal,
   grant: principals ->+ Permission
}
static disj sig SecurityPolicy extends Policy {}
```

As shown in Figure 2, `Permission` has two subtypes. `AllowedPermission` represents `Permission` that is allowed by the `SecurityPolicy`. `ExecutingPermission` is meant to be possessed by a specific `Principal` when it tries to make an access. `ExecutingPermission`, representing the access action, is checked against the set of `AllowedPermission` in order for the access to be granted. Actually, the description says that the access is granted if (1) the `Principal` is controlled by the `SecurityPolicy`, and (2) `ExecutingPermission` is included in the set of `AllowedPermission`.

```
sig AllowedPermission extends Permission {}
fact{ all p: AllowedPermission |
          p in allowedPermission() }
fun allowedPermission(): set Permission
{
  result =
    { p : Permission |
          all ps in ran(SecurityPolicy.grant) |
              p in ps }
}
sig ExecutingPermission extends Permission {}
```

Last, `isPrincipalAccessAllowed` describes how to check whether an access with `Mode` by a specific `Principal` to a `Target` is allowed or not. It ensures that the `SecurityPolicy` of the current system refers to the `Principal`, and its `grant` relationship includes the necessary access permission. Actually, `isTargetAccessAllowed` uses `AccessPermission` and `ExecutingPermission` to implement the checking. Its detailed definition is omitted.

```
fun isPrincipalAccessAllowed
        (u: Principal, t: Target, m: Mode)
{
   (u in SecurityPolicy.principals) &&
   isTargetAccessAllowed(u.(SecurityPolicy.grant),
                      t,m)
}

run isPrincipalAccessAllowed for 3 but 1 Policy,
                                        2 Mode
```

The above `run` command specifies that the analysis is done with one `Policy` (i.e. `SecurityPolicy`) and two `Mode` (both `ReadMode` and `WriteMode`).

3.3 Weaving

We consider here the example shown in Figure 3, and use the model of the weaving as discussed in Section 2.3. First, we need a one-to-one mapping between the two roles to be merged. Below specifies that `LogSource` in the Logging aspect is merged with `Target` in the Access Control aspect.

Merging at $\langle\{\text{LogSource}\}, \{\text{Target}\}\rangle$ is
    identify: LogSource $\rightarrow$ Target
with $true$

The case for `AccessAction`, however, is a bit complicated because it has two relating roles to be merged in the Access Control aspect. It can be said that `AccessAction` in the Logging aspect is refined to a mapping of `Principal` to `ExecutingPermission` in the Access Control aspect. The description below specifies that (1) there is a mapping from `Principal` to `ExecutingPermission`, and (2) there is another mapping from `AccessAction` to the mapping of (1). The mapping (1) is defined within the Access Control aspect, and the latter one (2) corresponds to the one-to-one mapping for expressing the role merging. In addition, the constraint part has to specify two things; (a) `AccessAction` should faithfully represent an access event, and (b) the event describes that `Principal` makes an access of `ExecutingPermission`.

Merging at $\langle\{\text{AccessAction}\}$,
        $\{\text{Principal, ExecutingPermission}\}\rangle$ is
    identify: AccessAction $\rightarrow$
        (Principal $\rightarrow$ ExecutingPermission)
with $a : AccessAction, u : Principal,$
    $ep : ExecutingPermission \mid$
        a represents $\langle u, ep\rangle$

Then, we have a definition of the aspect weaving by means of the two role merging descriptions. The weaving operation has further constraints describing (1) that the `target` attribute of `ExecutingPermission` corresponds to `Target` having a mapping with `LogSource` and (2) that `AccessAction` is stored in the `tobe` attribute of `LogSource` indicating it to be executed.

Weaving (Logging, AccessControl) is
    Merging at $\langle\{\text{LogSource}\}, \{\text{Target}\}\rangle$
    Merging at $\langle\{\text{AccessAction}\}$,
        $\{\text{Principal, ExecutingPermission}\}\rangle$

with $(ep : ExecutingPermission, t : Target \mid ep.target = t)$
$\wedge\ (a : AccessAction, l : LogSource \mid a\ in\ l.tobe)$

In Alloy, the one-to-one-mapping relationships, a set of $identify$ relationship, would be encoded compactly by introducing a new `sig`, and all the constraints are collected to be an invariant expression in `fact`. We have `WeavingOne` with appropriate constraints as below.

```
sig WeavingOne {
  id1: LogSource -> Target,
  id2: AccessAction ->
       (Principal -> ExecutingPermission)
}
fact{
   all g: WeavingOne |
     one l: LogSource,  a: AccessAction,
         u: Principal, ep: ExecutingPermission |
         ep = u.(a.(g.id2)) &&
         ep.target = l.(g.id1) && a in l.tobe
}
```

An application property can be encoded as `fun P1` for the weaved model description in Figure 3 by using `WeavingOne` explicitly.

```
fun P1(s1,s2: LogState, u: Principal,
       ep: ExecutingPermission, g:WeavingOne)
{
  all a: AccessAction |
      a in dom2(g.id2) && execute(s1,s2,a)
}

run P1 for 3 but 1 Policy, 1 Principal
```

The `run` command searches for a solution that all `AccessAction` is successfully logged. Actually, we can say that the weaving `WeavingOne` is consistent because Alloy responds back at least one solution. If the weaving is inconsistent, Alloy cannot find any solution at all.

4 Discussions

We introduced a role-based aspect-oriented modeling method and defined a notion of the aspect weaving in a systematic way. Essentially, weaving is considered as a collection of role merging with auxiliary constraints arising from the application specific functionalities. The case study in [14] is the motivation of the aspect weaving model in the present paper.

In the proposed modeling approach, role provides a uniform basis to represent design in that any design model consists of a set of roles and their interactions. It depends on how the designer recognizes the system whether a specific set of roles and their interactions is regarded as either a primary concern (classes) or cross-cutting concern (aspect). We have chosen a view in that the two role-based models explained in the paper were supposed to represent aspects. In addition, since role is a sole primitive, we could define the aspect weaving as a collection of role merging in a clear manner.

We further discussed how we used Alloy, a lightweight formal specification language and analysis tool, for the verification as well as precise model descriptions. Technically, we discussed how a role-based aspect model was formally described in Alloy, and introduced how roles were weaved. In particular, we showed that weaving in the models could be formalized as role merging, which was compactly represented in a declarative manner in Alloy. The constraint-based Alloy analyzer was easy to use for reasoning about whether a given property was satisfied before and after weaving. Alloy has its own theoretical basis on first-order relational logic, and provides high-level syntactic constructs to express object-oriented concepts [8][9]. It is not surprising that the role-based design models and Alloy are in harmony.

Here we give a brief survey on related work. Since security is one of the most interesting cross-cutting concerns, the aspect-oriented approaches have been reported [4][6]. Some work on the role-based and aspect-oriented modeling are mentioned in Section 2.1. Since the weaving in this paper is essentially a transformation of the models in which object interactions are dominant, our approach is related to work on composing design pattern [18]. Further, the formalization of design patterns [3][13][20] share some common technical points with the formal descriptions of the role-based aspect-oriented model discussed in the paper.

Last, we enumerate a list of future work. First, it is necessary to conduct a number of case studies in order to consider whether the role-based aspect-oriented modeling is promising or not. Second, the proposed approach of the weaving does not seem scalable. It is because we have to add application-specific constraints in the definition of the weaving operation. Automatic weaver is desirable. Further, the weaved design description may become awkward and is hard to read by human designers since how the models are weaved can be understood only by studying the possibly a lot of weaving operations. Model transformation dealing with *refactoring* would help obtain a clear model description that has *no* explicit weaving operations in it.

5 Position Summary

In order to have a modeling method for aspect-oriented models at early stages of the software development, we adopt a role-based approach and define a notion of the aspect weaving in a systematic manner. The weaving is essentially a collection of role merging with takining into account the application specific constraints. Further, we use Alloy, a lightweight formal specification language and analysis tool, fo the verification as well as precise model descriptions. Thanks to the declarative styles of specification and the constraint-based analysis in Alloy, we can express the weaving relationships as well as the aspect descriptions in a compact manner. We think that Alloy is a valuable tool for writing and reasoning about the aspect-oriented models.

Acknowledgements

The authors would like to thank the anonymous referees of the Early Aspects workshop and the participants at the local technical meeting WNASC2004, especically Prof. J. Kramer, Prof. G. Kiczales, and Prof. H. Masuhara for their comments to make our ideas clearer than before.

This research has been conducted under Kumiki Project, supported as a Grant-in-Aid for Scientific Research (13224087) by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan.

References

[1] E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *Proc. ICSE 2004*, May 2004.

[2] H. Cannon. Flavors: A Non-hierarchical Approach to Object-Oriented Programming. Symbolics Inc., 1982.

[3] A. Cechich and R. Moore. A Formal Basis for Object-Oriented Patterns. In *Proc. APSEC'99*, 1999.

[4] B. De Win, B. Vanhaute, and B. De Decker. Security Through Aspect-Oriented Programming. In *Advances in Network and Distributed Systems Security*, pages 125–138, Kluwer Academic 2001.

[5] T. Elrad, R. FIlman, and A. Bader. Aspect-Oriented Programming. *Comm. ACM*, Vol.44, No.10, October 2001.

[6] G. Georg, I. Ray, and R. France. Using Aspects to Design a Secure System. In *Proc. 8th ICECCS*, December 2002.

[7] K. B. Graversen and K. Osterbye. Aspect Modeling as Role Modeling. OOPSLA 2002 Workshop on TS4AOSD , November 2002.

[8] D. Jackson, I. Shlyakhter, and M. Sridharan. A Micromodularity Mechanism. In *Proc. FSE-9*, September 2001.

[9] D. Jackson. Lightweight Analysis of Object Interactions. In *Proc. TACS 2001*.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, 1997.

[11] K. Lieberherr. Controlling the Complexity of Software Designs. In *Proc. ICSE 2004*, pages 2–11, May 2004.

[12] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In *Foundations of AOL Workshop*, 2002.

[13] T. Mikkonen. Formalizing Design Patterns. In *Proc. ICSE'98*, pages 115–124, 1998.

[14] S. Nakajima and T. Tamai. Lightweight Formal Analysis of Aspect-Oriented Models. technical note, 2004.

[15] B.Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships between Multipleviews in Requirements Specifications. *ACM TOSEM*, Vol.2, No.10, pages 760-773, October 1994.

[16] A. Rashid, A. Moreira, and J. Araujo. Modularisation and Composition of Aspectual Requirements. In *Proc. AOSD'03*, pages 11–20, 2003.

[17] T. Reenskaug, P. Wold, and O.A. Lehne. *Working with Objects: the OOram Software Engineering Method*. Manning Publications, 1996.

[18] D. Riehle. Composite Design Patterns. In *Proc. OOPSLA'97*, pages 218–228, 1997.

[19] D. Riehle and T. Gross. Role Model Based Framework Design and Integration. In *Proc. OOPSLA'98*, pages 117–133, 1998.

[20] M. Saeki. Behavioral Specification of GOF Design Patterns with LOTOS. In *Proc. APSEC 2000*, 2000.

[21] D. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML*. Addison Wesley, 1999.

[22] T. Tamai. Objects and Roles: Modelling based on the Dualistic View. *Information and Software Technology*, vol. 41, no. 14, pages 1005–1010, 1999.

[23] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-Dimentional Separation of Concerns. In *Proc. ICSE'99*, pages 107–119, May 1999.