# Lightweight Formal Analysis of Aspect-Oriented Models

Shin NAKAJIMA
NII and PRESTO, JST
nkjm@nii.ac.jp

Tetsuo TAMAI
The University of Tokyo
tamai@acm.org

## ABSTRACT

In aspect-oriented modeling at early stages of the software development, model descriptions with various abstraction levels are involved so that two kinds of model transformation, refinement as well as weaving, should be considered. We adapt a role-based aspect-oriented modeling method and define a notion of aspect weaving as role merging. We further adapt Alloy, a lightweight formal specification language and analysis tool, for verification as well as precise model descriptions. Thanks to the declarative style of expressing constraints in Alloy, we can formally express role merging clearly and add it incrementally.

## 1. INTRODUCTION

Separation of concerns is a promising approach to reducing the complexity of software design [17]. However, 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 [12]. For example, 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 among classes [5].

*Aspect* is quite important in early stages of the software development. It is a common exercise to analyze a complex system from multiple viewpoints [16] to identify aspects that are (mostly) independent with each other. Further, the traceability becomes clear when we keep track of how a specific aspect at the early stage is refined and weaved in the software development process. Aspect-oriented modeling covers many activities at early stages of the software development. 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] adapt the idea of aspect to have clear design relating to cross-cutting concerns.

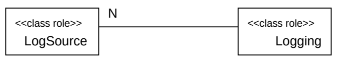

**Figure 1: Logging Aspect**

Since aspect-oriented modeling (AOM) is a method at the early stages of the software development, model descriptions with a variety of abstraction levels are involved. Two kinds of model transformation in AOM, refinement as well as weaving, should be considered. On the other hand, weaving is the only transformation of interest in AOP since the abstraction level dealt with is close to programs. AOP compiler or other tool provides mechanisms for weaving aspects with primary concerns automatically [11][13][24].

In this paper, we first introduce an idea of applying a role-based modeling method [18][23] to AOM and the method employs diagram-based model descriptions á la UML [6]. The model transformation, either refinement or weaving, is done manually, and some verification is needed between the model descriptions before and after the transformation. We then propose to use Alloy [9][10], a lightweight formal specification language and analysis tool, for the verification as well as precise model descriptions. Technically, we discuss how a role-based aspect model is described in Alloy and how roles are weaved. We demonstrate our idea by using an example case on the security aspect.

## 2. ASPECT-ORIENTED MODELING

### 2.1 Aspect and Role

As a start, we show here *Logging*, that is an example cross-cutting concern often used in AOP literatures. When we use the Logging framework of Java platform class library, a method call like below is used.

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

Such method calls should be inserted in all the method bodies where we want to record access logs. Although the logging is a conceptual unit focusing on a particular functionality, the actual code fragments are scattered over classes. Aspect/J [11], for example, provides a first-class language element to describe the logging aspect, which frees us from writing scattered codes.

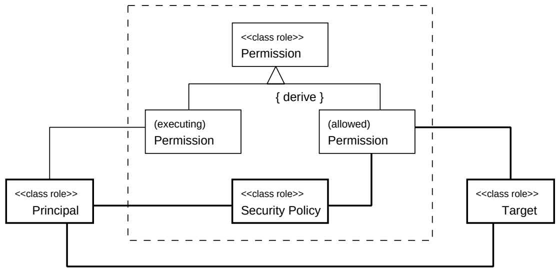

**Figure 2: Access Control Aspect**

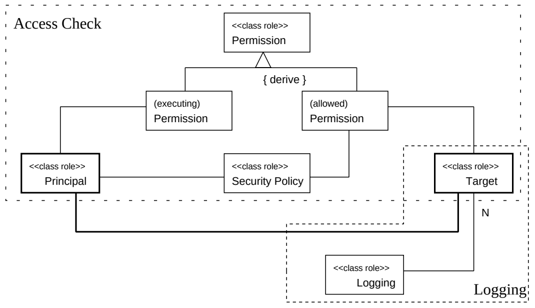

**Figure 3: Weaving Two Aspects**

The *Logging* aspect in Figure 1, however, is simple when we view the design in terms of the role-based modeling [18]. All the classes to be logged have `LogSource` role, and it is related to `Logging` role.

Next, we consider another aspect *Access Control* in Figure 2. It is an aspect taken from the system security, that is a typical example cross-cutting concern. When `Principal` makes an access of either read or write to `Target`, the aspect says that each access is checked against the pre-defined `SecurityPolicy`. It is basically a collection of `Permission`, and 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 [18][22], and the role-based approach is a good candidate to have model descriptions.

Role-based modeling and aspect-oriented modeling share some common notion [23]. When we identify a class for a primary concern, role provides alternative viewpoints to see function and behavior of the system. A role-based modeling method is focussed on finding appropriate roles for charactering an object [18]. And it also can be used as a method to identify collaborative behavior [20]. On the other hand, the aim of analyzing roles in AOM puts emphasis on identifying aspects themselves. K. B. Graversen and K. Osterbye [8] propose a method on how an identified aspect is translated into `advice` in AspectJ. The method is supposed to be applicable for an aspect such as the `Logging`, which is rather independent. For aspects more on the interaction, G. Georg et al [6] proposes 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 adapt the role-based modeling method á la [6] in that

Aspect = Roles and their Interaction,

and use UML-like diagrams for the model description.

## 2.2 Refinement and Weaving

Aspect-oriented modeling (AOM) is different from AOP in that it should deal with model descriptions at various abstraction levels. AOM may start with a very abstract aspect model description, and then refine it in several steps to an adequately concrete level. Finally, the model description is detailed enough to be implemented. Therefore, we have to consider two distinctive kinds of model transformations, refinement and weaving, in AOM.

Weaving in AOM is also different from the case in aspect-oriented programming. In AOP, an aspect is a first-class language element, and AOP compiler or other tool provides mechanisms for weaving aspects with primary concerns automatically [11][13][24].

In the role-based aspect-oriented modeling adapted in the paper, the model description is centered around the structural relationships between the identified roles. Refinement is a model transformation that adds further roles or expand some role into detailed ones. Weaving two aspects is similar to *mixins* in object-oriented programming [2], and is performed through a process of identifying a role in the first aspect with role(s) with the second one.

The model transformations, either refinement or weaving, in AOM is done manually. Thus, there are many places to contaminate model descriptions by introducing defects. For example, a property that an aspect originally has is violated after weaving the aspect with another aspect or model descriptions of primary concerns. Unfortunately the diagram such as UML is not adequate for formal analysis of the model descriptions.

## 2.3 An Example

We will present how weaving and refinement in AOM looks like by using a concrete example case.

Weaving in the role-based aspect-oriented modeling is essentially to make two corresponding roles merged. Schematically, we obtain a model description in Figure 3 after weaving the Logging aspect (Figure 1) and 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.

The model description in Figure 3 is still abstract in that it is not detailed enough to be implemented, for example, in the Java platform. The model can be refined into the one in Figure 4 when we make a decision to use the Logging framework and JAAS (Java Authentication and Authoriza-

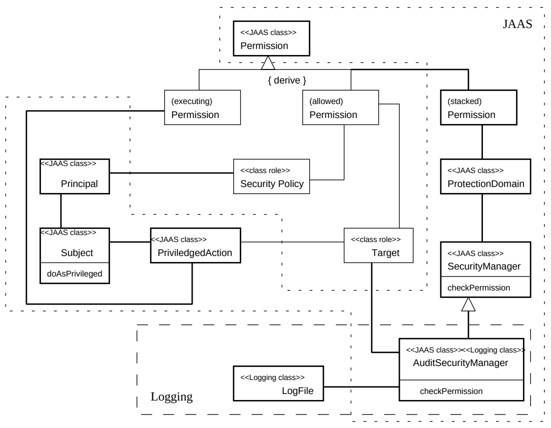
JAAS
<<JAAS class>> Permission
{ derive }
(executing) Permission
(allowed) Permission
(stacked) Permission
<<JAAS class>> Principal
<<class role>> Security Policy
<<JAAS class>> ProtectionDomain
<<JAAS class>> Subject
doAsPrivileged
<<JAAS class>> PriviledgedAction
<<class role>> Target
<<JAAS class>> SecurityManager
checkPermission
<<JAAS class>><<Logging class>> AuditSecurityManager
checkPermission
<<Logging class>> LogFile
Logging

**Figure 4: Refinement with JAAS**

tion System) [7].

Figure 4 shows a model description after the refinement. Some model elements already turn out to be concrete JAAS framework classes. Others are still left as roles so that they are to be *weaved* with application classes.

Although some of the basic vocabularies such as `Principal` and `Target` are common, the model is quite different from the one before the refinement. In particular, what is related to the access checking reflects the mechanism of *the stack inspection* provided by the JAAS framework [7].

`Principal` in Figure 3 is expanded into three of JAAS classes (`Principal`, `Subject`, `PriviledgedAction`). And classes such as `SecurityManager` to implement internal mechanism for Java security checking appear in the model. `AuditSecurityManager` is defined to over-ride `checkPermission` method to include logging as well as access checking.

In summary, in the role-based aspect-oriented modeling method, the weaving is basically a role merging and the refinement often results in a large change that affects the structural relationships in the model description.

## 3. FORMAL ANALYSIS WITH ALLOY

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

### 3.1 The Application System

#### 3.1.1 Overview

Figure 5 illustrates a sketch of a system that reads in data from `SourceFile`, and writes data to `DestinationFile` probably after some computation done in `BusinessLogic`. The system may have *Logging* aspect as well as *Access Control*.

We also enumerate two of the important properties that the whole system has.

(P1) Every access is logged regardless of the result of access checking,

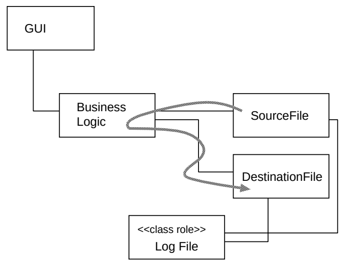

**Figure 5: Sketch of System**

(P2) Data is transferred as long as the accesses are allowed.

These two properties are formally checked against various model descriptions in the succeeding sections.

#### 3.1.2 Alloy Description

We adapt the Alloy language and analyzer [9][10] for formal description and analysis of the aspect model description. Alloy has originally been proposed as a formal specification language for *rationalized* UML class diagram with OCL.

Basically, we translate a class or role to `sig` in Alloy, a link to an attribute in `sig` with an optional `fact` constraint, and functional behavior such as data/control flow to `fun`. Below, we show a fragment of Alloy description for the simple application system in Figure 5.

Since the main functionality is *moving* data from `SourceFile` to `DestinationFile`, we introduce `Element` that moves around as well as three `sig` for the main participants. For example, `SourceFile` internally keeps `Element` data as its attribute `data`. `InitElement` is a subtype of `Element`, and is meant to be an initial value representing data of no significance. `ThisElement` represents the data that moves around between the participants.

```
sig SourceFile { data: Element }
sig DestinationFile { data: Element }
sig BusinessLogic { data: Element }
sig Element {}
static disj sig InitElement extends Element {}
static disj sig ThisElement extends Element {}
```

We, further, introduce an auxiliary but important `GlobalState`. It manages the global state to represent *snap-shot* of the system. Then, two functions, `readInFile` and `writeOutFile`, are *state-transformers* on `GlobalState`. They can express that a data originally in one place moves to another place after invoking an appropriate function.

```
sig GlobalState {
  bm : BusinessLogic,
  source : SourceFile,
  dest : DestinationFile
}
```

```
fun readInFile(s1, s2 : GlobalState) {
  some x: Element |
    s1.source.data = x &&
    s1.bm.data = InitElement &&
    s2.source.data = s1.source.data  &&
    s2.bm.data = x &&
    s2.dest = s1.dest &&
    x != InitElement
}

fun writeOutFile(s1, s2 : GlobalState) {
  some x: Element |
    s1.bm.data = x &&
    s1.dest.data = InitElement &&
    s2.dest.data = x &&
    s2.bm.data = s1.bm.data &&
    s2.source = s1.source &&
    x != InitElement
}
```

Then, the net data-flow, from `SourceFile` to `DestinationFile`, is encoded as a logical conjunction of the two `fun`. And by using the `run` command, we can ensure that the required data-flow is achieved. In the `run` command, the integer argument `3` specifies that the size of the search space used by the Alloy analyzer.

```
fun execution(s1, s3: GlobalState) {
  some s2 : GlobalState |
    readInFile(s1,s2) && writeOutFile(s2, s3)
}

run execution for 3
```

In particular, the `run` command is a checking of the property (P1) for the case of the system without access checking.

## 3.2 Abstract Aspects

We present formal Alloy descriptions of the two aspects.

### 3.2.1 Logging Aspect

In order to have rigorous Alloy description, We add some element roles to the model description 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)
}
```

Just as `GlobalState` in the example in Section 3.1.2, 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. Clearly, the two attributes are exclusive as specified 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.

```
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 being a variant of the property (P1) is satisfied.

### 3.2.2 Access Control Aspect

As a start, we introduce `Principal` and `Target` (see Figure 2). The uniqueness constraint is also imposed on the `name` attribute. The definition of `Principal` only is shown here.

```
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.

```
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, it uses `AccessPermission` and `ExecutingPermission` mentioned above.

```
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

Next, we show how weaving is encoded in Alloy.

### 3.3.1 *Weaving Two Aspects*

We consider here the example shown in Figure 3. First, we need an one-to-one mapping between the two roles to be merged. `WeavingOne` specifies that `LogSource` in the Logging aspect is merged with `Target` in the Access Control aspect. 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 be a mapping of `Principal` to `ExecutingPermission` in the Access Control aspect.

```
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)) && a in l.tobe
}
```

The one-to-one mapping in `WeavingOne` needs a further constraints on each element, whose relationships are described by the `fact` as below.

- `AccessAction` is a faithful representation of an event that `Principal` makes an access encapsulated in `ExecutingPermission`.
- `AccessAction` to be executed is stored in `tobe` attribute of `LogSource`.

The application property (P1) 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.

### 3.3.2 *Weaving with Application*

Since the property (P2) involves two `Target` as illustrated in Figure 5, we need a model description that the application (Section 3.1.2) is weaved with the two aspects (Figure 3). As for the case of the previous example, we first introduce an one-to-one mapping constraint that merges the corresponding roles.

WeavingTwo, similar to WeavingOne, is the base definition, which is accompanied with further application specific constraints in the fact. It says that SourceFile and DestinationFile are actually Target elements managed in LogState.

```
sig WeavingTwo {
  from: SourceFile -> LogState,
  to: DestinationFile -> LogState
}
fact{
  all g: WeavingTwo |
    some r1: GlobalState, s1: LogState |
      (r1.source).(g.from) in s1.target &&
      (r1.dest).(g.to) in s1.target
}
```

P21 shows that after two consecutive accesses (doubleExecute) we have appropriate SourceFile and DestinationFile both of which have the same data, and the accesses are logged. Namely, the whole system obtained after weaving satisfies the property (P2) if no access checking is involved.

```
fun P21(s1,s0: LogState,
        a1,a2: AccessAction, g: WeavingTwo)
{
  doubleExecute(s1,s0,a1,a2) &&
  (some t3: SourceFile, t4: DestinationFile |
      t3.(g.from) in s0.target &&
      t4.(g.to) in s0.target &&
      t3.data = t4.data) &&
  (some d1, d2: LogData |
      d1 in s0.state.logset &&
      d2 in s0.state.logset &&
      d1.id = a1.id && d2.id = a2.id)
}

run P21 for 3  but 1 Policy, 1 Principal
```

Then, we obtain P2 by adding P21 a fragment relating to the Logging and the Access Check aspects. Actually we introduce WeavingOne since it is the base condition that relates the two aspects.

```
fun P2(s1,s0: LogState, u: Principal,
      ep1,ep2: ExecutingPermission,
      g: WeavingTwo,  h: WeavingOne )
{
 some a1,a2: AccessAction |
     ep1 != ep2 &&
     a1 in dom2(h.id2) &&
     a2 in dom2(h.id2) &&
     P21(s1,s0,a1,a2,g)
}

run P2 for 3 but 1 Policy, 1 Principal
```

P2 can be analyzed under various conditions by defining appropriate SecuityPolicy.

## 3.4 Refinement

As discussed in Section 2.3, the model after the refinement is changed much in its structure and includes elements relating to the stack inspection mechanism, that the JAAS framework is based on, for the the access checking [7]. In other word, we have to elaborate a formal specification of the JAAS stack inspection.

The Alloy description of the stack inspection mechanism results in some 300 lines including the basic common vocabularies in the access checking domain such as Principal and Target [15]. Although the perperties (P1) and (P2) are to satisfy the refined model, the Alloy description takes qutie different forms. It is because the property fun is so defined in terms of the elements describing the stack inspection.

## 4. DISCUSSION AND RELATED WORK

We first introduced a role-based aspect-oriented modeling method and identified that two kinds of model transformation, namely refinement and weaving, were involved. Since the model transformation is not done automatically as in the case of aspect weavers in AOP, some verification is needed between the model descriptions before and after the transformation.

Our approach for the formal verification is to use Alloy, a lightweight formal specification language and analysis tool. 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.

As for the translation from the diagram-based descriptions to the Alloy descriptions, we resolved some ambiguity in the original diagrams and added further detailed specification fragments, which was done manually. We need further study in order to obtain an automatic translation tool.

It seems that our approach to the weaving, actually a role merging, is interesting in itself. Its Alloy description is essentially an one-to-one mapping between the involving roles. It, however, is important to point out here that the mapping is accompanied with further constraints that are specific to the roles.

As in the case of weaving the Access Control with the Logging aspect, a role (AccessAction) is refined to be a particular relationship between two roles (a mapping of Principal to ExecutingPermission). This implies that some weaving should take into account the refinement between the roles. Thus, weaving in AOM depends on the application semantics, and thus it is hard to have automatic weavers as in the case of AOP. However, thanks to the declarative style of expressing such constraints in Alloy, we were able to formally express such a role merging clearly and incrementally.

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 [19]. Further, the formalization of design patterns [3][14][21] share some common technical points with the formal descriptions of the role-based aspect-oriented model discussed in the paper.

Last, we are interested in an automatic weaver for aspect-oriented models. It seems mandatory to have formalization of both aspect-oriented model descriptions and the two model transformations, refinement and weaving. This paper presented how aspect weaving was formally represented and analyzed with Alloy by using a specific case, which is expected be a start of further research.

## 5. 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] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security (2ed.).* Addison Wesley 2003.

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

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

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

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

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

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

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

[15] S. Nakajima and T. Tamai. Formal Design Analysis of Changes in Security Policy. IPSJ SIG Technical Report (in Japanese), 2003-SE-143-8, July 2003.

[16] 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.

[17] D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM*, Vol.15, No.12, pages 1053-1058, December 1972.

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

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

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

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

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

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

[24] 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.