Model-driven development of access control aspects

Manuel Koch,
Karl Pauls

Freie Universität Berlin,
Fachbereich Mathematik und Informatik,
Takustr. 9, D-14195 Berlin, Germany,
{mkoch,pauls}@inf.fu-berlin.de

Abstract: In distributed system development security is a major design criteria. Security and more specific access-control can be seen as an aspect in terms of Aspect-oriented Programming. We present a Model Driven Development (MDD) approach to the development of access control aspects which permits their generation from UML models. The contribution of this paper is threefold. First, we present the integration of access control requirements into the software development process. Second, we introduce an access control specification language for distributed systems that is capable to express the access control aspect of a system. Third, we define an interceptor-based approach for the integration of aspects into the application logic during runtime instead of code weaving. Applying our technique to the design process of a distributed system allows to generate the access control aspects of the system in an specification language that subsequently can be enforced by an interceptor enabled platform.

1 Introduction

Model Driven Development (MDD)[NE00] has the advantage that the many specification documents developed in the software development process are related and that existing dependencies are documented. A documentation of dependencies helps to take into account every change of a model or a relation necessary to guarantee consistency. One of the main drawbacks of model-driven development, however, is the consideration of security requirements in the development process which is not yet sufficiently supported. Security aspects, however, are inherent in any modern software system that is not used in completely trusted environments. The lack of a systematic support for software engineers who need to produce secure software is based on the fact, that security requirements are generally difficult to analyze and model [NE00, DS00] and because security policies are generally specified in terms of highly specialized security models that are not integrated with general software engineering models. Recent research concerns the integration of security engineering into the software development process [Jue02, LBD02].

Aspect-oriented programming separates application code from application independent code [EFB01, Lop04]. Aspects represent usually non-functional concerns as logging, se-

curity. etc. More specific in adaptive programming implicitly the Law of Demeter for Concerns (LoDC) is used i.e. “talk only to your friends that contribute to a common set of concerns or that share the same concerns” [Lie04]. This has the advantage that application logic can be developed independently and aspects can be added when needed without changing application code. Current approaches to AOP weave aspect code into application code, for example AspectJ.

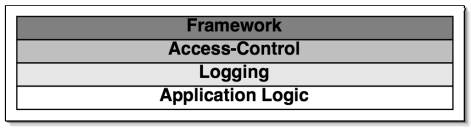
We present in this paper a model-driven approach to the development of access control aspects. We consider a software development process containing the common stages for functional requirement analysis and system design, but also UML models for security requirements and security design. Functional and security models are modeled separately. The functional models are developed independent of any security aspect and serve as a basis for the security models. Parts of the security models can be automatically generated from the functional models. A designer may extend these generated models by access control information if necessary. The access control UML models are finally used to generate access control policies which can be deployed into the enforcement infrastructure. A generated access control policy can be seen as security aspect. In contrast to approaches as AspectJ, there is no code weaving of application code and aspect code. The application code remains unmodified and the security logic interferes with the application only over special mediators. An implementation of a mediator may be an invocation interceptor (e.g., CORBA interceptors for intercepting CORBA remote calls or Axis handlers for intercepting SOAP messages).

2 Aspect-oriented Programming

Aspect-oriented programming (AOP) separates concerns into single units called aspects [EFB01]. Concerns can range from notions such as security and quality of services to buffering, caching, and logging. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable modules. With AOP, each aspect can be expressed in a separate and natural form, and can then be automatically combined together into a final executable form by an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of procedures, modules, or objects, increasing reusability of the codes.

Aspects surround the system kernel, i.e., the core functionality of the system. Several aspects can be used in one application, e.g., logging, security and monitoring etc. Figure 1 shows this general idea of a system kernel and several surrounding aspect layers. Figure 2 shows a concrete application in which the system kernel contains the application logic. Aspect layers are access control and logging. This model permits to change functionally equal components without changing the higher aspect layers. Especially, components can come and go at any time. Furthermore, aspects can come and go, as well, without affecting the component. This view permits the construction of components which are designed only for its functionality. Component design does not considers the non-functional aspects. They can be added at any time by an aspect layer.

This approach is different from the aspect weaving approach in which aspects are implemented separately from the application logic and is then compiled together (e.g., AspectJ, JBOSS4.0). In our approach aspects are both specified and deployed separately. This is done by assuming an interceptor facility as a mediator between the aspect layers and the system kernel (CORBA interceptor, RACCOON interceptor implementation, Axis handler concept).



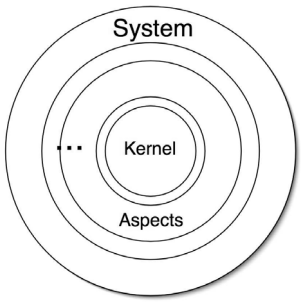


Abbildung 1: Aspect Oriented Programming.

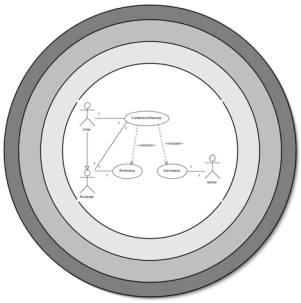


Abbildung 2: Concrete Aspect Oriented Programming.

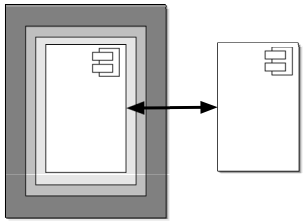


Abbildung 3: Concrete Aspect Oriented Programming.

The aspect of access control is specified by the *View Policy Language.*

2.1 View Policy Language

View-based access control is an access control model specifically designed to support the design and management of access control policies in object-oriented systems [Bro01, Bro02]. The principal feature of VBAC is that of a *view* for the description of fine-grained access rights, which are permissions or denials for operations of distributed objects. Views on objects are assigned to principals, i.e., to individual subjects or roles, and a principal has access to an operation of an object if (s)he has a view on the object with a permission to call the operation. The principal has no access if the operation is explicitly denied in another view on that object that is available to the role, or if no permission is found.

For defining views, we use the *View Policy Language* (VPL)[Bro01] as part of a policy design document, which is a product of the design stage in the development process. In addition to the usual features described above, VPL also supports view extension, so that an extending view inherits all access rights of the base view. Views can statically be re-

stricted such that they can only be assigned to specific roles and views can be declared to be *virtual*. Virtual views have empty bodies. To specify automatic changes in the security state, VPL defines *schemas*. A schema defines triggers for the automatic assignment and removal of views to principals.

View-based access policies are delivered in descriptor files and deployed together with applications in the target environments, similar to approaches like EJB [Sun00] or the CORBA Component Model [OMG99].

3 Model-driven development of Access Control Aspects

This section presents the model-driven approach to develop the access control aspects. The various design documents are combined in order to generate an access-control related model of the system. This can be done automatically and presents a starting point for designing more complicated security concerns. The approach allows to model systems without taking access-control into account which are specified later on based on the informations generated from the model.

As an example we use a conference management system originally introduced in [Bro01, Bro02]. Figure 4 shows the use case for the conference management application. A PC

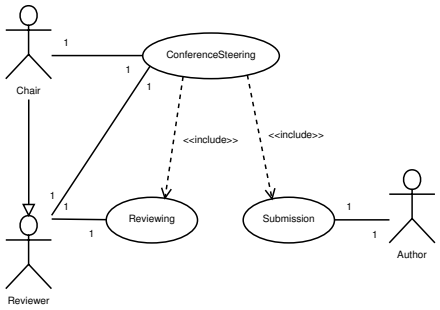


Abbildung 4: The use case for the conference application.

chair can issue a call for papers to open a submission phase for a conference, so that authors may submit papers. The chair is responsible for the declaration of the submission deadline, which terminates the submission phase and starts the reviewing phase. Reviewers write and submit reviews for their assigned papers. The reviewing phase is terminated by the chair calling for a final decision.

The class diagram developed on the basis of the use case is shown in Figure 5. The roles are derived from the UML actors of the use case diagram and model the presence of users of different types. Strictly speaking, roles are already related to access-control but at this stage refer to the subjects behind the roles hence, serve as a grouping mechanism only.

This is important since it is explicitly allowed to model roles with named properties (i.e, attributes). In a high level view named properties are something provided by a specific subject. The fact that a certain role may have attributes denotes that any subject acting through the role will have to provide a value for the key that consists out of the attribute name. Following the example we get the roles `Chair`, `Reviewer` and `Author`. Reviewers

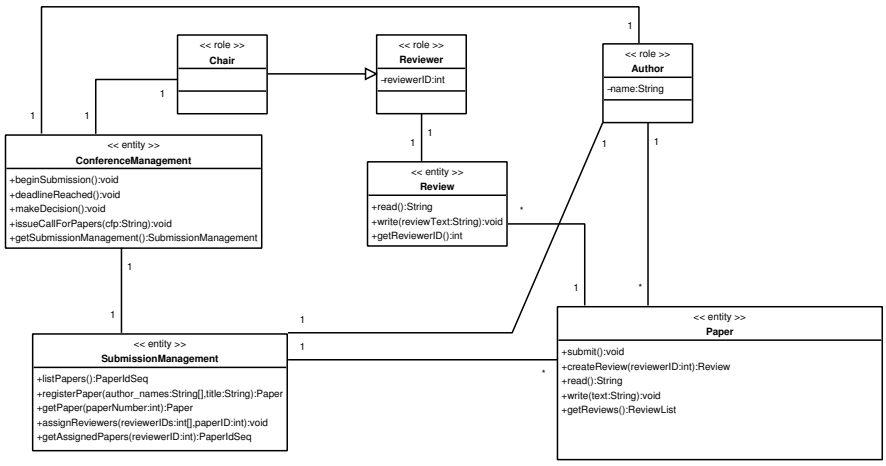


Abbildung 5: The class diagram for the conference application.

are identified by a *reviewerID*, authors by the property *name*. Subjects which want to play a role must have these role properties. A specific role property is assigned and set either by the system administrator who is responsible for the user–role assignment or in the security aspect. The entities model the core functionality of the system.

3.1 Generation of Access Control Views

We describe the views on the application from the viewpoint of the different actors in a sequence diagram. The sequence diagram in Figure 6 describes the view of the actor `Chair`. The chair is responsible for the management of the conference procedure ranging from issuing the call for papers, assigning reviewers to submitted papers to the final decision of accepted papers.

The sequence diagram for the actor `Author` is given in Figure 7. After an author has got the `SubmissionManagement` object for the conference, s(he) can write and submit a paper.

The sequence diagram for the actor `Reviewer` is given in Figure 8. A reviewer gets the papers for which s(he) is responsible from the `SubmissionManagement`. The reviewers reads each of the papers and writes a review.

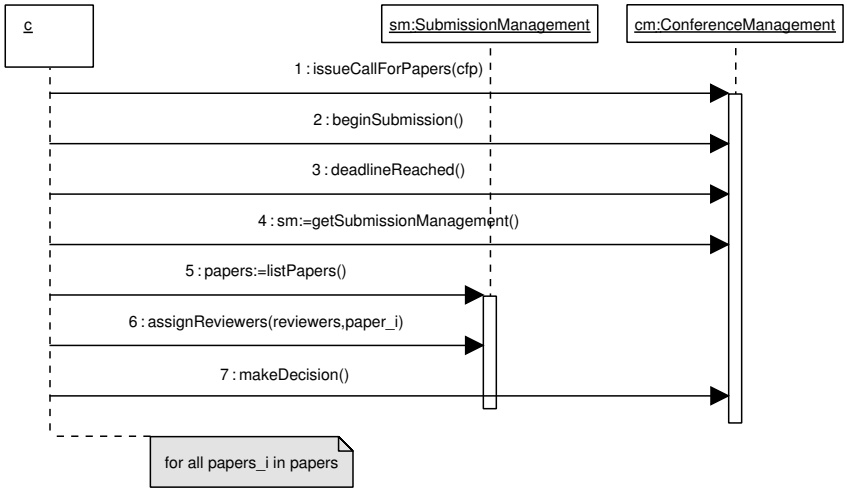


Abbildung 6: The sequence diagram for the Chair's view.

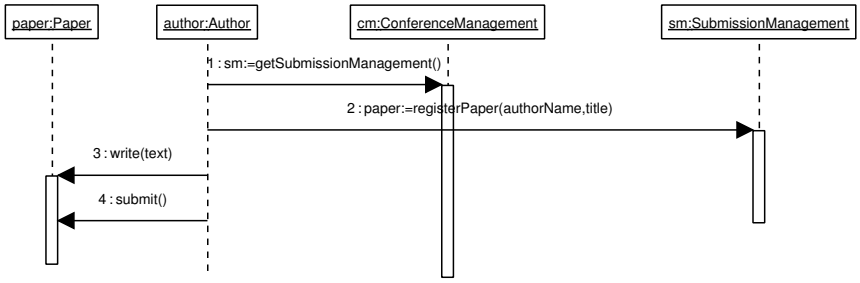


Abbildung 7: The sequence diagram for the Author's view.

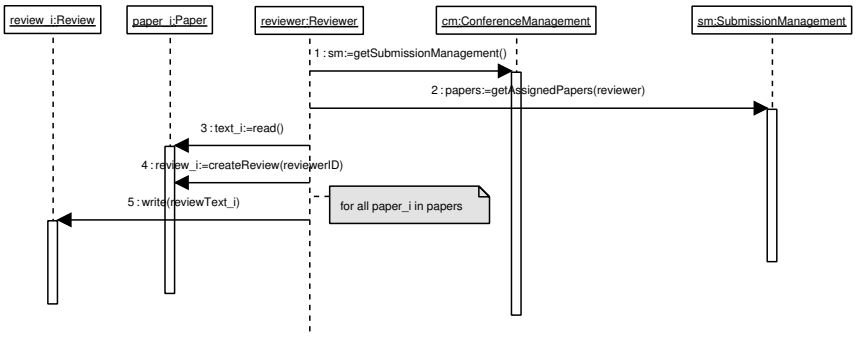


Abbildung 8: The sequence diagram for the Reviewer's view.

Out of the sequence diagrams the access control views can be automatically generated. In each sequence diagram each of the occurring objects are considered. For each object a view is generated which contains the operation calls on this object. Consider as an example the sequence diagram for the author in Fig. 7. The diagram has the three objects `paper:Paper`, `cm:ConferenceManagement` and `sm:SubmissionManagement`. On the paper object there are the operation calls `write()` and `submit`. Therefore, a view on class `Paper` is defined which contains the two access rights *write* and *submit*. On the conference management object `cm` there is only one call `getSubmissionManagement()`. This gives a view on class `ConferenceManagement` consisting of one right *getSubmissionManagement.* On class `SubmissionManagement` results a view containing the right *registerPaper*, since this operation is called by the author in the sequence diagram.

Analog define the sequence diagrams for the chair (Fig. 6) and the reviewer (Fig. 8) views. These views can be automatically generated from the sequence diagrams. The result is shown in Fig. 9.

3.2 Refinement and Completion of Access Control Views

The views generated from sequence diagrams are generally incomplete in the sense that they do not give a complete access control specification. This is due to the fact that sequence diagrams show only scenarios the designer is interested in. On the other hand, the views may be redundant in the sense that the same views are generated from different sequence diagrams (e.g., view *ConfMgmt2* and *ConfMgmt3*). To sum up, not all of the access control information can be generated and the designer uses the generated views as a basis which is refined to the final access control specification.

The refinement includes also the introduction of access control roles and their initial assignment to views. Actors give a good hint for the specification of access control roles, since in many cases the actors of the use case diagram correspond to access control roles. Howe-

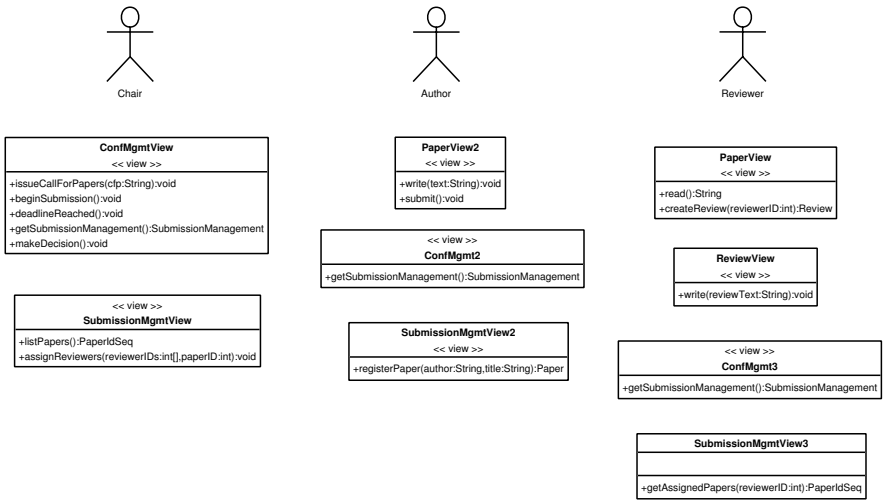


Abbildung 9: The generated access control views.

ver, this has not to be always the case, since in some applications it may be more natural to integrate several actors in one access control role. In our conference management example, however, we have a direct correspondence between actors and access control roles, i.e., we have the roles `Chair`, `Author` and `Reviewer`.

Figure 10 shows the views and roles created by the designer on the basis of the views in Fig. 9. One of the redundant views *ConfMgmt2* and *ConfMgmt3* in Fig. 9 is removed (here *ConfMgmt3*) and a view *PaperBaseView* is introduced which has only operation `read`. The view serves as base view for the views *PaperView* and *PaperView2*. In addition, two virtual views (stereotype `<<virtual view>>`) are added, namely *SubmissionPhase* and *ReviewingPhase*. Virtual views do not contain operations and are used in the example as prerequisites for other views. For example, all submission management views require the virtual view *SubmissionPhase* which is specified by the association from the submission management views to the virtual view. The intended meaning of such a requirement association is, that a role which has a submission management view can call the operations of that view only if the role has the virtual view at the same time.

Views are assigned to roles by associations. For example, role `Actor` is assigned to the views *PaperView2* and *SubmissionMgmtView2*. The cardinality at the association end of the view specifies whether the view is initially assigned (value 1) or whether the view is not assigned in the initial state but can be assigned later (value 0..1). For example, the views *SubmissionMgmtView2* and *PaperView2* are not initially assigned to the role `Author`, but can be assigned dynamically during runtime.

The dynamic assignment or removal of views to and from roles, respectively, is modeled in an activity diagram. The operation call which triggers a view change and the actu-

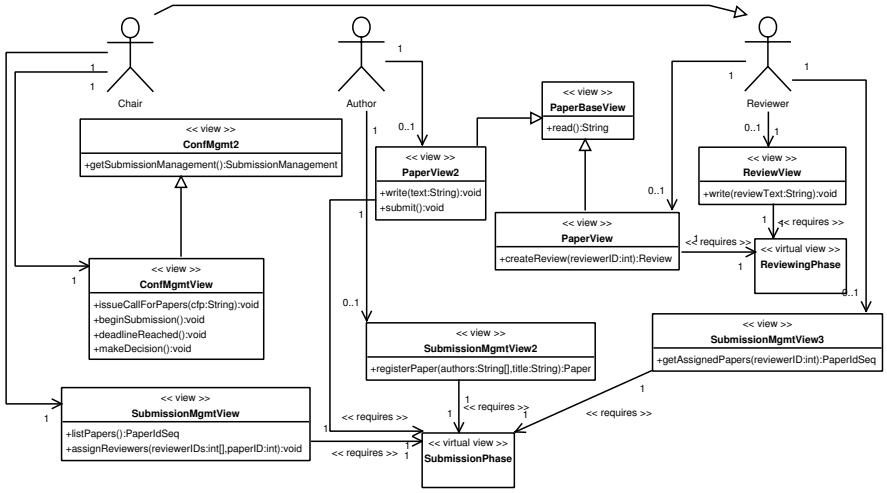


Abbildung 10: Extension and refinement of the generated views.

al view-role relation modification are specified as edge labels in the diagram. Figure 11 shows the activity diagram of our example. The initial state is given by the assignment of views as specified in Figure 10. The protection state changes if the chair opens the submission phase by calling operation `beginSubmission`. Therefore, the trigger is a call of operation `beginSubmission`, its effect is the assignment of the view *SubmissionPhase* (which is a view on class `ConferenceManagement`) to role `Author`. The new protection state is called *SubmissionPhase* in which authors are permitted to submit their articles. Authors can register papers by calling operation `registerPaper(authorNames,title)`. The effect of this operation call is that all authors of the paper get the view *PaperView2* on the registered paper. That authors have only access to their paper ensures the condition `where Author.name in authornames`. The attribute *name* of the role `Author` conveys the caller's name which must coincide with one of the authors of the paper. Calling operation `deadlineReached` ends the submission phase and starts the reviewing phase. The activity diagram specifies the view removal and assignment effects. In the reviewing phase, the operations `assignReviewers` and `createReview` cause view assignment changes. In the former case, a set of reviewers get the right to work on the assigned paper. In the second case, a reviewer gets the right to create a review for a paper. Calling operation `makeDecision` changes into the final state.

3.3 Generation of VPL

We present in this section the generation of the VPL policy from the UML diagrams. The access control roles and the role hierarchy are given in Figure 10. The attributes of roles

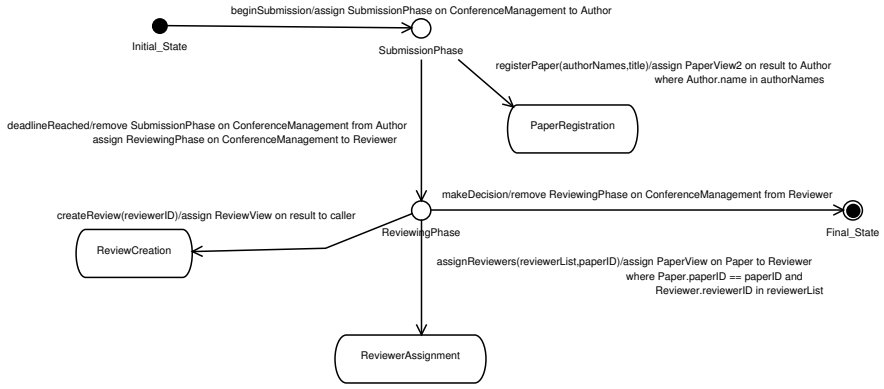


Abbildung 11: The dynamic changes of the protection state.

are given in the class diagram (for the example in Fig. 5). The VPL policy starts with the keyword `policy` and a policy name and contains a list of roles, views and schemas.

```
policy Conference {
   roles
      Reviewer
         property int reviewerID
      Chair: Reviewer
         holds ConfMgmtView, SubmissionMgmtView
      Author
         property String name
}
```

The intended notion of `Chair:Reviewer` is that role `Chair` is an extension of `Reviewer`. Only the role `Chair` initially holds views, since the assigned views for `Reviewer` and `Author` in Figure 10 have a cardinality value 0..1. The properties for `Reviewer` and `Author` are generated from diagram 5.

The views and their extension hierarchy are generated from the diagram in Figure 10. As an example we give the VPL specification for the views on paper objects. A view is specified by the keyword `view` (`virtual view` in the case of a virtual view) followed by the view name and a list of operations which the view allows a caller to call. The permitted operations are directly taken from the UML diagram in Figure 10.

```
view PaperBaseView controls Paper {
   allow read
}

view PaperView:PaperBaseView {
   allow createReview
}
```

```
view PaperView2:PaperBaseView {
   allow write,submit
}
```

The activity diagram is used for the generation of VPL schemas. For each trigger (i.e., operation call) in the activity diagram there is an entry in a schema which observes the class the operation belongs to. For example, the trigger *beginSubmission* creates an entry in the schema which observes `ConferenceManagemennt` since the operation `beginSubmission` belongs to this class. The schema entry is given by the assign or remove effect specified in the edge label belonging to the trigger. A VPL schema starts with the keyword `schema`, the schema name and the class to which the operations of the schema belong.

```
schema InitialState observes ConferenceManagement
{
   beginSubmission
      assign SubmissionPhase on ConferenceManagement to Author
   deadlineReached
      remove SubmissionPhase on ConferenceManagement from Author
      assign ReviewingPhase on ConferenceManagement to Reviewer
   makeDecision
      remove ReviewingPhase on ConferenceManagement from Reviewer
}

schema SubmissionPhase observes SubmissionManagement
{
   registerPaper(author_names, title)
      assign PaperView2 on result to Author
             where Author.name in author_names
   assignReviewers(reviewerList,paperID)
      assign PaperView on Paper to Reviewer
             where Paper.paperID == paperID and
                   Reviewer.reviewerID in reviewerList
}
```

The VPL presented above is an extension of the original VPL presented in [Bro02]. We added the possibility to specify role properties. This extension becomes necessary if the VPL (i.e., the aspect of access control) shall be totally independent from the application logic. As a side-effect the VPL can be easily used in non-object-oriented platforms, as well [FKP04]. Furthermore, the VPL presented in this article can consider also the parameters of operations which allows the designer a more fine-grained specification of access control. This level of fine-grained specification is necessary in any of our case studies [FKO03, KKOB03].

4 Conclusions

We have presented a model-driven approach to the specification of access control aspects. Access control aspects are generated from sequence diagrams and the access control policy is modeled in UML class and activity diagrams. The presented VPL policy is an extension of the VPL used in [Bro02]. The UML diagrams are used to generate the VPL policy. Our approach of aspect orientation separates application from security code during develop-

ment and runtime. The enforcement of a generated VPL policy needs a mediator between the application and access control logic. In the project Raccoon a mediator is implemented by a CORBA interceptor. Future work concerns the implementation of a mediator for other platforms, e.g. Web Services or OSGi, and the formal definition of the VPL extensions.

Literatur

[Bro01] Gerald Brose. *Access Control Management in Distributed Object Systems.* Dissertation, Freie Universität Berlin, 2001.

[Bro02] G. Brose. Manageable Access Control for CORBA. *Journal of Computer Security*, 4:301–337, 2002.

[DS00] Premkumar T. Devanbu und Stuart Stubblebine. Software Engineering for Security: A Roadmap. In Anthony Finkelstein, Hrsg., *The Future of Software Engineering.* ACM Press, 2000.

[EFB01] T. Elrad, R. Filman und A. Bader. Aspect-Oriented Programming. In *Communications of the ACM*, Jgg. 44, Seiten 28–97, 2001.

[FKO03] T. Fink, M. Koch und C. Oancea. Specification and Enforcement of Access Control in Heterogeneous Distributed Applications. In *Proc. of International Conference on Web Services - Europe 2003 (ICWS-Europe'03)*, 2003.

[FKP04] T. Fink, M. Koch und K. Pauls. An MDA approach to Access Control Specifications Using MOF and UML Profiles. In *Proc. of First International Workshop on Views On Designing Complex Architectures*, Seiten 165–181, 2004.

[Jue02] J. Juerjens. UMLsec: Extending UML for Secure Systems Development. In *Proc. of UML 2002*, number 2460 in LNCS, Seiten 412–425. Springer, 2002.

[KKOB03] M. Koch, R. Kober, C. Oancea und J. Bernarding. Zugriffsschutz für Web Services - eine Krankenhausfallstudie. In *Proc. 8. Telematik im Gesundheitswesen (TELEMED2003)*, 2003.

[LBD02] T. Lodderstedt, D. Basin und J. Doser. SecureUML:A UML-Based Modeling Language for Model-Driven Security. In *Proc. of 5th Int. Conf. on the Unified Modeling Language*, number 2460 in LNCS. Springer, 2002.

[Lie04] Karl J. Lieberherr. Controlling the Complexity of Software Designs. In *Proc. of 26th International Conference in Software Engineering*, Seiten 2–11, 2004 2004.

[Lop04] C. Lopes. *Aspect-Oriented Software Development*, Kapitel AOP: A Historical Perspective. Addison Wesley, 2004.

[NE00] Bashar Nuseibeh und Steve Easterbrook. Requirements Engineering: A Roadmap. In Anthony Finkelstein, Hrsg., *The Future of Software Engineering.* ACM Press, 2000.

[OMG99] OMG. *CORBA 3.0 New Components Chapters, TC Document ptc/99-10-04.* OMG, Oktober 1999.

[Sun00] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0, Final Draft*, Oktober 2000.