# SecTOOL – Supporting Requirements Engineering for Access Control

Steffen Kolarczyk, Manuel Koch, Klaus-Peter Löhr, and Karl Pauls

Freie Universität Berlin
Institut für Informatik
Takustr. 9, D–14195 Berlin, Germany
{kolarczy, mkoch, lohr, pauls}@inf.fu-berlin.de

**Abstract.** SecTOOL is a case tool for security engineering. It comes as an extension to traditional UML tools, taking into account *access control requirements*. In particular, it supports the developer in eliciting access control information from UML diagrams for the early phases, starting with *requirements analysis* and use case diagrams. *Access control policies* coded in VPL or XACML are generated from the diagrams; vice versa, textually coded policies can be *visualized* in UML diagrams. Design and usage of the tool are described, emphasizing its *platform independence* through XACML.

## 1 Introduction

For a long time, security has been seen as a possible add-on to software systems, a non-functional property to be considered in the late phases of system development, if at all. A long series of security disasters, often continuing after "refitting" systems with security patches, has led to a rethinking and has given rise to the notion of *security engineering*: security aspects are now seen as an integral part of a software system, to be considered right from the early phases of development and to be implemented throughout the software life cycle.

This paper deals with taking into account *access control* requirements in UML-based software development. We take the view that these requirements can naturally be expressed as additions to several UML diagrams, starting with the use case diagram. Early work on this approach has been reported in [7]. Other authors have adopted similar approaches. Model-driven development has been extended to cover access control, resulting in *SecureUML* for J2EE applications [12], and methods for dealing with object-oriented access control have been applied to web services [8, 1]. Information flow and multi-level security is the subject of [10], and a comprehensive treatment of UML-based security engineering is given in [9].

While methodologies for access control engineering in the context of model-driven development are emerging, tools support and enforcement infrastructures are lagging behind. To improve this situation, we have developed SecTOOL, a plugin for the *Rational* UML case tool for supporting the early development phases. SecTOOL has been developed in the context of the Raccoon project

G. Müller (Ed.): ETRICS 2006, LNCS 3995, pp. 254–267, 2006.

[5, 6] and reflects Raccoon's approach to access control management: object-oriented access control policies are specified in a formal policy language; access protection according to a given policy is enforced through an appropriate infrastructure (originally for Corba objects, using Corba middleware); the application software has no built-in protection, and the policy to be applied can be modified without modifying any application code.

The implications for SecTOOL are as follows: both access control requirements and functional requirements *are* specified in an integrated fashion; *however*, the policies resulting from the access control requirements are not enforced by the application code derived from the diagrams but by an independent security infrastructure. In this respect, SecTOOL is similar to the SecureUML plugin for the *ArcStyler* tool [13]. There are three differences, though: first, SecTOOL covers all the *early development steps*, beginning with requirements analysis and use case diagrams; secondly, it supports the specification of *dynamic* modifications of privileges; and third, its design is *platform-independent* (*XACML* code can be generated).

The contribution of SecTOOL to security is seen in the enhanced reliability in handling all phases of access control engineering: requirements, design, implementation, management and maintenance. This is the heritage from the Raccoon approach to access control management, in particular from its *View Policy Language* (VPL) [4, 5]. A short introduction to VPL is given in section 2. How SecTOOL is used in specifying a first approximation to the access rights to be granted is described in section 3. Refining this according to the principle of least privilege is the subject of section 4. Section 5 presents a comprehensive view of the tool's features, and section 6 explains how an access control policy is actually enforced. The paper ends with a discussion of related work and a conclusion. A running example - a *conference management system* - is used throughout the paper.

## 2 VPL Revisited

### 2.1 View-Based Access Control

*View-Based Access Control (VBAC)* [6] is an object-oriented version of *Role-Based Access Control (RBAC)*, or more precisely, of a restricted version of $\text{RBAC}_3$. VBAC policies were originally introduced to overcome weaknesses in the standard Corba security model. Aiming at improved manageability of application-specific access control, VBAC uses grouping mechanisms such as *roles* (for subjects), *types* (for objects) and *views* (for operations). A *view* is basically a subset of the set of operations of an interface (originally an IDL-coded interface) and may contain additional information related to access control. (Note that there is no relation to the notion of "view" as known from database systems.)

Views on types are assigned to roles statically, but views on objects or types can also be assigned to or removed from subjects or roles in a dynamic fashion. Assigning a view on an object to a subject is tantamount to passing a capability for that object to the subject.

### 2.2 Static Policies

VBAC policies are coded in *VPL (View Policy Language)*, a simple language for specifying roles, views and view assignments/removals. VPL has no type system of its own: a VPL text refers to interfaces specified in a suitable typed language, e.g., IDL or UML. A simple text fragment should suffice for getting a first impression of VPL. The reader is referred to [4, 6] for a more detailed description of the language and its semantics.

```
policy conference {
   roles Author
            holds Submitting
         Reviewer
            holds BrowsingPapers
         Chair: Reviewer
            holds Steering
            maxcard 1
            excludes Author

  view BrowsingPapers controls Conference {
        allow listPapers, getPaper
   }
   ...
 }
```

This example alludes to the conference management system that will be introduced in section 3. Three roles are declared: `Author, Reviewer, Chair`. The role `Chair` is declared to extend, or *dominate*, the role `Reviewer`, according to the $\text{RBAC}_3$ model. This role is restricted to at most one subject, and the roles `Chair` and `Author` exclude each other.

The `holds` clauses specify the initial views held by the roles. An extended role inherits the views of the dominated role, so the role `Chair` holds two views, `BrowsingPapers` and `Steering`. Views are tied to interfaces, as mentioned before. The view `BrowsingPapers` is tied to the interface `Conference` (shown in the appendix); it includes the operations `listPapers` and `getPaper`.

### 2.3 Dynamic Policies

VPL supports the dynamic modification of the application's protection status: execution of an operation can be specified to cause *assignment or removal of views* to roles or subjects. For instance, the right to select a paper for reviewing will be granted to a PC member only when the PC chair executes the operation `submissionDeadline`. This is specified in the VPL policy by a construct known as *schema*: `assign` or `remove` clauses are attached to the relevant operations, as shown here:

```
policy conference {
   ...
   schema SteeringSchema observes Conference {
      submissionDeadline
         remove Submitting from Author
         assign ChoosingPapers to Reviewer
      decide
         remove ChoosingPapers from Reviewer
      }
   schema ReviewSchema observes Review {
      submit ...
      }
   ...
}
```

The schema `SteeringSchema` refers to the `Conference` interface which includes operations `submissionDeadline` and `decide` (see appendix).

## 3 Identifying Required Privileges

SecTOOL supports the design of VPL-coded access control policies, exploiting information inherent in several types of UML diagrams for the early phases of software development. The first step in acquiring access control information involves the use case diagram, the class diagram and several sequence diagrams. This step produces an approximation to the access privileges to be granted to roles. The privileges are then refined in a second step (to be described in section 4).

The operating mode of SecTOOL is best explained through a running example: we use a simplified version of a *conference management system*.

### 3.1 A Conference Management System

This system is to support the program committee, and in particular the PC chair, in preparing the conference program. (The organization committee's work is *not* supported.) The requirements are as follows:

- The preparation of the conference program goes through several *phases*: paper submission, reviewing, acceptance/rejection decision, submission of final versions. The end of each phase is marked by a certain *deadline*.
- An author may submit more than one paper. PC members - except the PC chair - may submit papers as well. All PC members, including the chair, act as reviewers.
- The PC members can inspect the submissions. After the submission deadline, they can choose (in FCFS fashion) the papers they want to review. Blind reviewing is put into practice: the reviewers do not learn the names of the authors. For n PC members and x papers, each PC member should choose at least 3*x/n papers (resulting in a total of 3 reviews per paper).

– A PC member may inspect all reviews for a paper *as soon as* she has submitted her own review for that paper. She may then decide to modify her review.
– When reviewing is finished, the PC decides about acceptance and rejection, and the PC chair sends notifications to the authors. The authors of accepted papers modify their papers and submit the final versions.

### 3.2 Use Case Diagrams Contain Role Information

The written requirements give rise to a UML use case diagram where authors, reviewers and the PC chair appear as actors. Use cases include the phases mentioned above, plus the *steering* done by the chair. The use case diagram is shown in Figure 1.

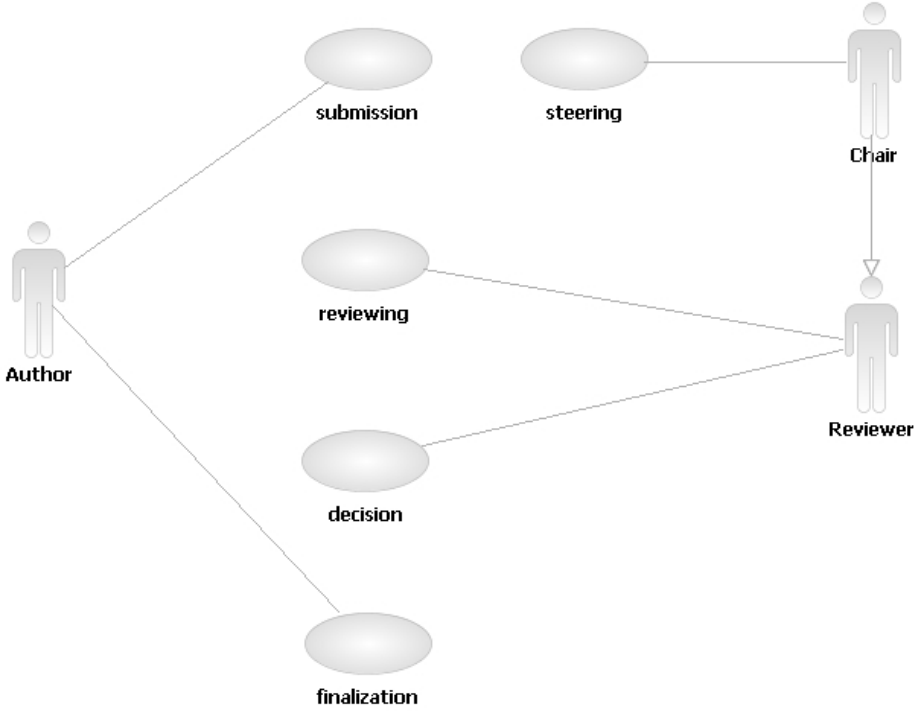


**Fig. 1.** Use case diagram

An *actor* in a use case diagram can be identified with a *role* in role-based access control. Note the inheritance relation between `Chair` and `Reviewer` - so `Chair` *dominates* `Reviewer`. Deriving an initial fragment of VPL text from the diagram can obviously be left to a tool - and this is where working with SecTOOL begins: a policy skeleton is generated that introduces role declarations as shown in section 2.2, but without mentioning any views. The views refer to the objects involved, so they cannot be derived from the use case diagram. A class diagram has to be designed.

### 3.3 Class Diagrams Contain Interface Information

According to the requirements, the system deals with objects such as *papers*, *reviews* and the singleton *conference*. The class diagram shown in Figure 2 contains the appropriate interfaces, plus empty interfaces for the roles mentioned above.

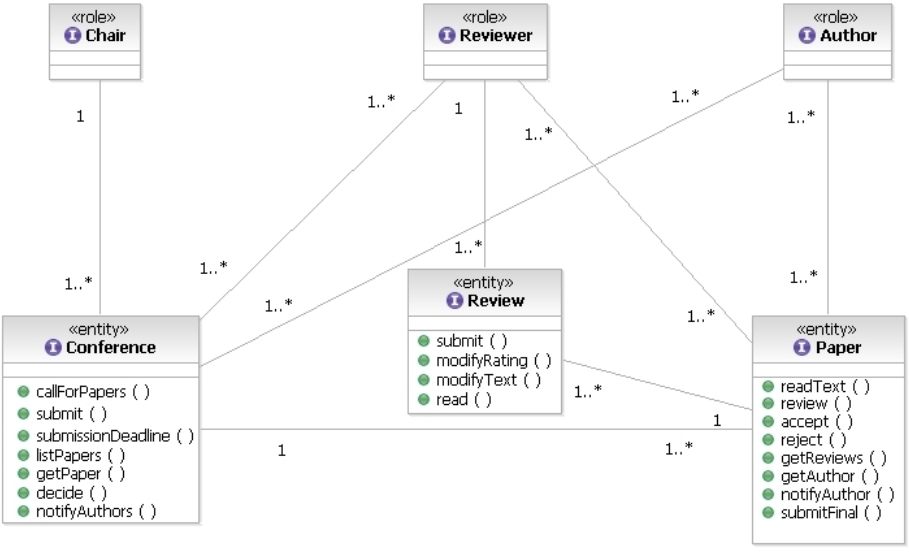


**Fig. 2.** Class diagram

The detailed specification of the operations is omitted, as the reader will be able to infer the semantics from the signatures (see Appendix A). For instance, the `readText` operation will deliver just the text of a paper, not its author and neither its status.

The access control policy has to restrict the permissions granted to roles to certain confined views on the interfaces of the objects. For instance, only the PC chair should be allowed to issue the `accept/reject` operations on `Paper` objects. So the question arises whether there is a systematic way of assigning proper views to roles or subjects.

### 3.4 Sequence Diagrams Contain View Information

A UML *sequence diagram* augments the information given in the use case diagram and the class diagram by indicating the operations actually executed for a certain use case. Figure 3 shows a diagram for the use case `Reviewing`. Similar diagrams for other use cases are not shown here.

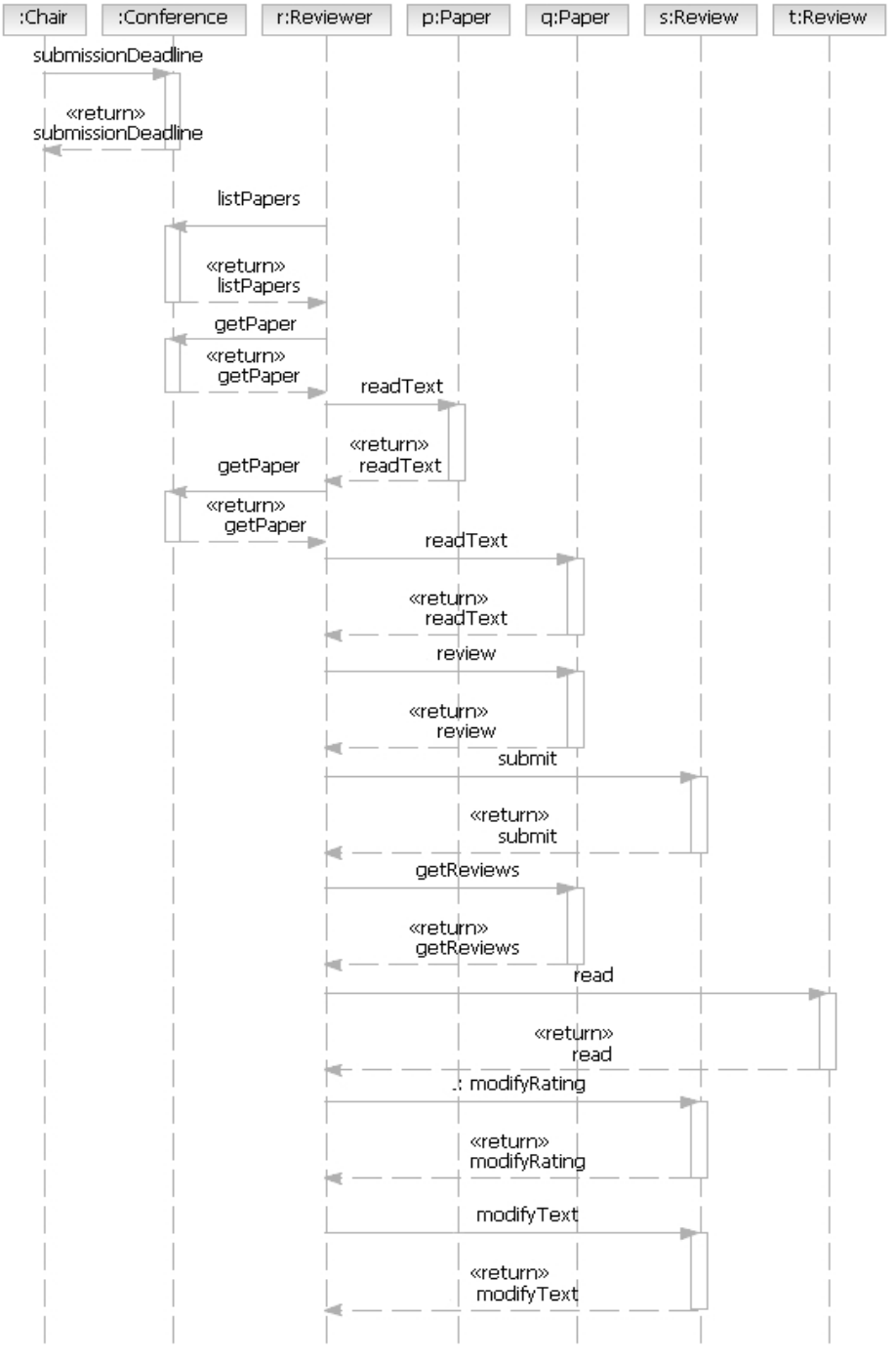


**Fig. 3.** Sequence diagram

Without SecTOOL, the human reader would derive the following views, written in VPL, from the sequence diagram:

```
view BrowsingPapers controls Conference {
      allow listPapers, getPaper
}
view HandlingPapers controls Paper {
      allow readText, review, getReviews
}
view Reviewing controls Review {
      allow submit, read, modifyRating, modifyText
}
```

SecTOOL automates this, adds the views to the VPL text *and* produces a graphical version: given the sequence diagram from Figure 3, the *view diagram* shown in Figure 4 is generated. The names of the views are chosen by the tool in a standard fashion. They are less distinctive than the names chosen above, but they do reflect the interfaces they refer to.

It is now the designer's task to decide about initial view assignment (`holds` clause) and dynamic assignment and removal (`schema` clause). For instance, the designer would append `holds BrowsingPapers` to the declaration of `Reviewer` in the VPL text. SecTOOL knows about the association between `BrowsingPapers` and `Reviewer`, and would refuse an accidental introduction of, say, `Author holds BrowsingPapers`.

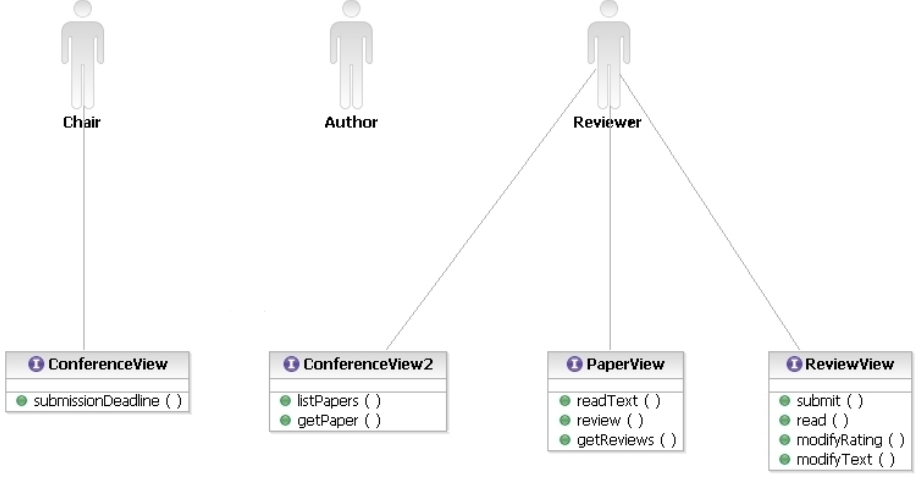


**Fig. 4.** View diagram

## 4 Refining the Privileges

The view diagram generated by SecTOOL lacks precision, in particular with respect to dynamic policies. First, the specific sequencing of invocations, as given in the sequence diagrams, is not taken into account; for instance, a PC member must get permission to choose papers for reviewing only when the chair has signalled the submission deadline. Secondly, certain permissions pertain to specific subjects and objects, not just to roles and object types; for instance, a reviewer may modify his or her own review, but not the reviews of others. And there is a third aspect that has to be considered: it must be possible to specify *denials* (negative permissions), in order to account for exceptions to general permissions. For instance, if a PC member has submitted a paper, she must not act as a reviewer *for that paper*.

### 4.1 Specifying Capability Assignment and Removal

In addition to supporting round-trip security engineering using diagrams, the ultimate goal of SecTOOL is the generation of complete access control policies, coded in VPL. So it is natural to use the *schema* construct of VPL for textual amendments to diagrams: they specify the dynamic assignment and removal of views on objects (i.e., capabilities) to and from subjects or roles.

A VPL schema for the operations of an interface can be attached as a UML pop-up *note* to the interface in the class diagram. SecTOOL understands this kind of decoration, ensures that consistency requirements are met, and integrates the assign/remove clauses into the final access control policy.

An example `schema SteeringSchema observes Conference` was given in the VPL introduction, section 2.3. Another schema would state that a reviewer gets permission to inspect all reviews for a paper as soon as she has submitted her own review. The schema refers to a view `getReviews` that has to be introduced manually:

```
view GetReviews controls Paper {
    allow getReviews
 }
 schema ReviewSchema observes Review {
    submit
       assign GetReviews on result to caller
 }
```

`result` is a reserved identifier, denoting the result of the operation `submit` (which is the associated `Paper` object). `caller` is another reserved identifier, denoting the invoking subject.

Note that singling out the `getReviews` operation from `PaperView` (see Figure 3) requires a modification of that view (viz., removal of `getReviews`). The modified view is the one that was called `ChoosingPapers` in the introductory section 2.3.

### 4.2 Negative Permissions

A VPL view may contain both positive and negative permissions. A negative permission is specified using the keyword `deny`; it overrides any related positive permissions (`allow`).

When working on a view diagram with SecTOOL, the developer can explicitly add views with negative permissions. While positive permissions are marked with a green bullet, negative permissions are marked with a red square. Figure 5 shows a variant of the earlier view diagram (Figure 4).

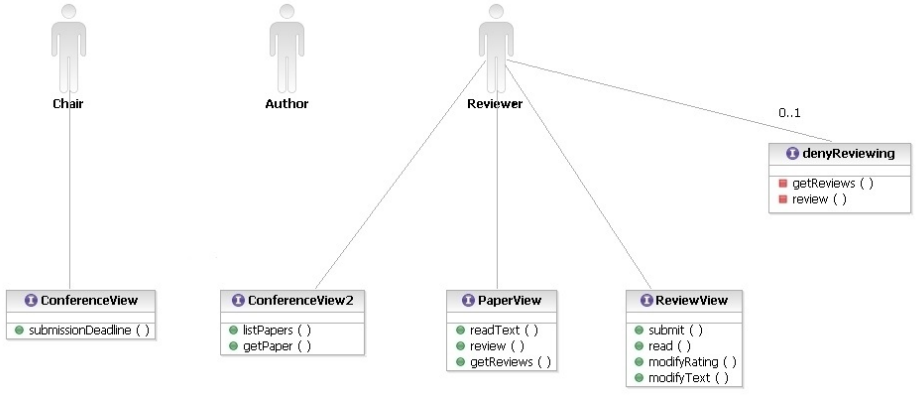


**Fig. 5.** Variant of view diagram

A negative view, `denyReviewing`, has been added to the diagram manually. This view reflects the requirement that a PC member must not review his or her own paper (if any). Note, however, that the new view diagram itself does not ensure this. The following clause can be added to the schema `SteeringSchema` given in section 2.3:

```
submit
    assign denyReviewing on result to caller
```

This overrides the general permission given by `submissionDeadline` in the original schema (assignment of `ChoosingPapers`).

## 5 SecTOOL in Action

### 5.1 Development

A graphical overview of SecTOOL-based development is given in Figure 6. SecTOOL cooperates with the typical UML tools, generating VPL policies from UML diagrams and, vice versa, visualizing VPL texts as UML diagrams. Manual modification of generated VPL text is possible as desired. So the tool supports round-trip engineering of access control policies in a comfortable manner, adding safety to the complex process of security engineering.

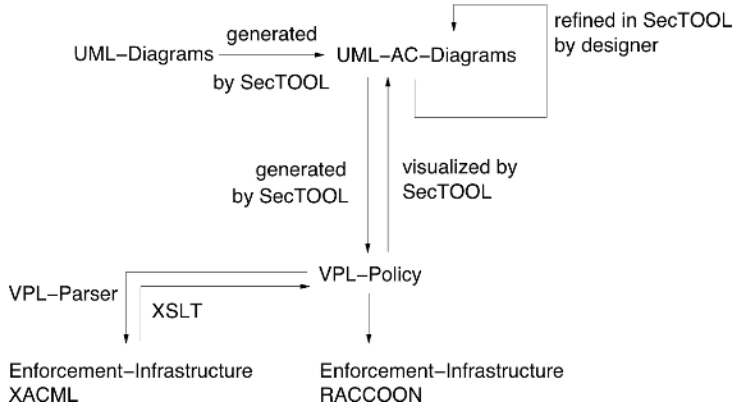


**Fig. 6.** SecTOOL in action

As explained earlier, the generation of VPL code describing a first version of a policy starts from a use case diagram, a class diagram and several sequence diagrams. A view diagram is created as an intermediate product. The developer may want to extend this diagram, e.g., to prepare for any denials that might be required. The view diagram, plus the schema information in the class diagram, are used as input to the VPL generation.

The name of the *policy* (in this case `Conference`) can be given by the designer or can be derived from the UML project name. The *roles* are derived from the actors in the use case diagram, as mentioned in section 3.2: each actor defines a role (here `Author`, `Reviewer` and `Chair`). Specialization between actors defines role inheritance; as `Chair` specializes `Reviewer` we have `Chair: Reviewer`.

*Views* are derived as follows: for each view in the view diagram with name `V` a VPL view clause `view V on I ...` is generated, where `I` is the interface name as found in the sequence diagram. The (positive) permissions, listed after the keyword `allow`, are given by the operations of the view in the view diagram. The VPL *schema* is generated by combining the schema information given in the notes in the class diagram.

The actual output of SecTOOL, as generated from the internal XML representation of VPL, is a specially formatted version of the VPL code shown earlier. Importing VPL code into SecTOOL for visualization as UML diagrams is also possible. This is useful if VPL is used for a system where UML-based documentation is not available, or in the case of round-trip engineering. Changes in the VPL specification are then reflected in the model, and consistency between model and implementation is preserved.

### 5.2 Maintenance

SecTOOL's separation of concerns – application logic vs. access control policy – is of great value during operation and maintenance of a developed system. In addi-

tion to the flexibility given by the role concept, the security administrator enjoys the freedom to modify the security policy without touching the application logic.

We have to keep in mind, of course, that the user interface will often be designed in such a way that certain operations are *a priori* impossible. For instance, an author will never encounter a Web interface that would give him the choice to inspect reviews meant for other authors. In general, however, modifying a policy will frequently make sense, in some cases even without adapting the user interface.

Consider the following example. The PC members should be allowed to check submitted papers only *after* the submission deadline. The policy is adapted to this changed requirement just by removing `holds BrowsingPapers` from the declaration of role `Reviewer` and by adapting the schema `SteeringSchema` as follows:

```
schema SteeringSchema observes Conference {
      submissionDeadline
         remove Submitting from Author
         assign BrowsingPapers, ChoosingPapers to Reviewer
      ...
 }
```

## 6 Enforcement Infrastructure

The deployment and management infrastructure designed for VPL policies is called Raccoon [4, 5]. A deployment tool processes VPL policies and stores view and role definitions in repositories that can be managed using graphical management tools. At runtime, role membership is represented by digital certificates issued by a role server. Access decisions are made by intercepting operation accesses which are forwarded in the case of a permitted access and blocked in the case of a denied access. Whether an intercepted access is permitted or denied depends on the policy information that is supplied by policy servers, which rely on the deployed policy information.

The Raccoon infrastructure is based on Corba and IDL specifications. The presented access control modeling process, however, is independent of the Raccoon infrastructure. Therefore, generated VPL policies should be available in any distributed system without requiring the Raccoon infrastructure. VPL policies should be presented in a platform-independent standard format. The Oasis has defined an XML standard for the specification of access control policies, called *eXtensible Access Control Markup Language* (XACML), together with an enforcement infrastructure specification. We have implemented a VPL parser which transforms VPL policies into XACML policies (XSL is used for transforming an XACML text back into VPL). This allows us to use SecTOOL for any platform that includes a standard XACML enforcement infrastructure.

## 7 Related Work

Work related to our approach to security engineering is presented in [2]. Basin et al. describe SecureUML, a model–driven approach to developing role-based

access control policies for J2EE applications. A formal basis allows the designer to reason about access control properties; tool support is given by an integration of SecureUML into the ArcStyler tool [13]. In contrast to our approach, however, the analysis stage of the software process is not considered. ArcStyler is a CASE tool for UML 2.0 and MDA–based modeling. The SecureUML extension is realized via plugins that allow to refine SecureUML–enhanced models towards different platform–specific security constraints (support currently includes J2EE and .NET). This is similar to our integration of SecTool as a plugin for Rational. It should be noted that the SecureUML meta–model is more expressive than the security model of the target platforms. Hence, not all parts of a model can be expressed declaratively. SecTool generates model–equivalent policies only.

Another approach to integrating security into UML has been described by Jürjens [9]. He shows how to model several security aspects by UML model elements such as, for example, stereotypes or tagged values. His approach is more general than ours since it is not restricted to access control; security protocols are considered as well. In contrast to our approach and [2], Jürjens does not provide tool support or an infrastructure to generate security policies from UML models or to enforce security policies.

In [1, 3], approaches to UML–based access control integration are given, focusing on a OCL–related workflow control language. High level security aspects as part of UML models, specified using OCL, are refined down to code or platform–independent XACML policies. As already mentioned above, we believe that visual modeling support should be provided to the developer.

## 8 Conclusion and Future Work

We have presented a tool for eliciting access control requirements from UML diagrams. Our approach is integrated into the UML software process and presents a UML representation of an access control policy. SecTOOL generates a basic access control model from UML diagrams and allows the designer to refine this model. Access control deployment files can be generated. The generated policies can be transformed into XACML policies, thus achieving platform independence.

Future work will be concerned with a more detailed investigation of the refinement of the generated view diagram: how does the designer arrive at a final access control policy? We would like to find out how this refinement process can be methodologically supported. Formal results [11] based on graph transformations look promising. It remains to be seen how they can be brought to fruition in future versions of SecTOOL.

## References

1. M. Alam, R. Breu, and M. Breu. Model–Driven Security for Web Services. In *Proceedings of the 8th Int. Multi-Topic Conference*, pages 498–505. IEEE, 2004.
2. D. Basin, J. Doser, and T. Lodderstedt. Model–Driven Security: from UML Models to Access Control Infrastructures. *Journal of ACM Transactions on Software Engineering and Methodology*, 2005.

3. R. Breu, M. Hafner, B. Weber, M. Alam, and M. Breu. Towards Model Driven Security of Inter-Organizational Workflows. In *Proceedings of the Workshops on Specification and Automated Processing of Security Requirements*, pages 255–267, 2004.
4. G. Brose. *Access Control Management in Distributed Object Systems*. PhD thesis, Freie Universität Berlin, 2001.
5. G. Brose. Raccoon — An Infrastructure for Managing Access Control in CORBA. In *Proc. Int. Conference on Distributed Applications and Interoperable Systems (DAIS)*. Kluwer, 2001.
6. G. Brose. Manageable Access Control for CORBA. *Journal of Computer Security*, 4:301–337, 2002.
7. G. Brose, M. Koch, and K.-P.Löhr. Integrating Access Control Design into the Software Development Process. In *Proc. of 6th Int. Conference on Integrated Design and Process Technology (IDPT)*, 2002.
8. T. Fink, M. Koch, and C. Oancea. Specification and Enforcement of Access Control in Heterogeneous Distributed Applications. In *Proc. of Int. Conference on Web Services - Europe 2003 (ICWS-Europe'03)*, 2003.
9. J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.
10. Jan Jürjens. Towards Development of Secure Systems Using UMLsec. In H. Hussmann, editor, *Proc. of Fundamental Approaches to Software Engineering (FASE'01)*, number 2029 in LNCS, pages 187–200. Springer, 2001.
11. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Foundations for a Graph-based Approach to the Specification of Access Control Policies. In F.Honsell and M.Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, Lect. Notes in Comp. Sci. Springer, March 2001.
12. T. Lodderstedt, D. Basin, and 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.
13. Interactive Objects. Arcstyler, 2005. `www.io-software.com`.

## A Operation Signatures

Operations of interface `Conference`:

```
callForPapers()
submit(a:Author,text:String):Paper
submissionDeadline()
listPapers():String
getPaper(number: int):Paper
decide()
notifyAuthors()
```

Operations of interface `Paper`:

```
readText():String
review(r: Reviewer):Review
accept()
reject()
getReviews():Review[]
getAuthor():Author
notifyAuthor()
submitFinal(text: String)
```

Operations of interface `Review`:

```
submit(text: String, rating: ABCD):Paper
modifyRating(rating: ABCD)
modifyText(text: String)
read():String
```