6 An Application Case Study

This chapter presents an application case study that shows how VPL can be used to express realistic, application–oriented access policies, and how the design of access policies can be integrated into the general design process. The example policy is used to demonstrate the use of a variety of policy language features in different design situations, including schemas, conditional views, and denials.

The example application is a system that supports program committees in reviewing papers for a conference and is a simplified version of the CyberChair system [van de Stadt, 1997], which is used by the ECOOP conferences. The prototype application was implemented in Java and tested on the security infrastructure presented in chapter 7. The following sections follow a simplified development process for this system.

6.1 Requirements Analysis

The pseudo analysis in this section is used to present the functionality of the example application in terms of use cases and to establish these use cases as a basis for the presentation of security requirements, the application design, and the design of an access policy.

6.1.1 Functional Requirements

The use case diagram in figure 6.1 gives a high–level overview of the main areas of system functionality. These use cases are described informally in textual form below. Only the main flow of events during each use case is sketched, exceptional situations are omitted.

Use Case ConferenceSteering

1. The use case starts when the PC chair issues a call for papers.
2. The chair enables the subordinate *Submission* use case by declaring the submission phase opened.
3. The chair declares that the deadline for submissions is reached, which terminates the *Submission* use case.

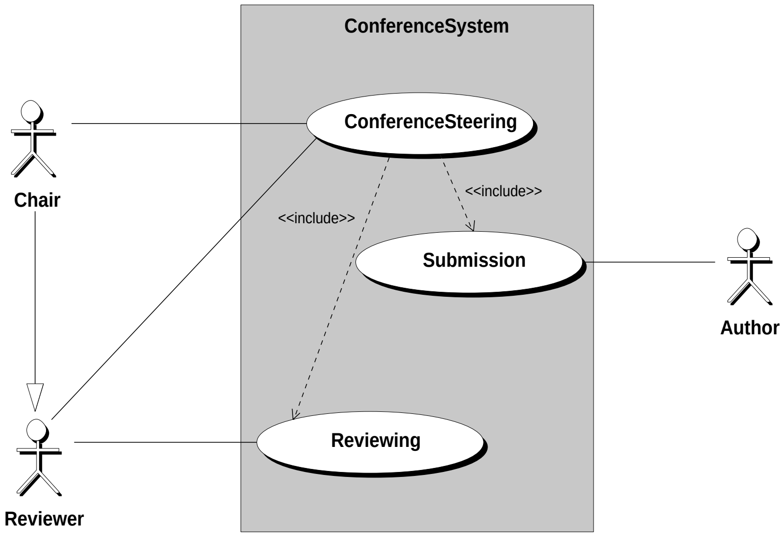


Figure 6.1: Actors and Use Cases for the Conference System.

4. Reviewers indicate their specific interests for reviewing certain papers.
5. The chair assigns reviewers to individual papers.
6. The *Reviewing* use case begins.
7. The reviewing phase is terminated by the chair calling for a final decision.
8. Potential conflicts between reviews for a single paper are resolved.
9. The final decision — approval or rejection — is made for each paper and must be unanimous.
10. Finally, authors are notified of the acceptance of their papers, at which stage this use case ends.

The indication of interest to review individual papers and the assignment of reviewers to papers is not supported by the example application. These two steps are assumed to be performed through direct communication between the chair and the reviewers, i.e., outside of the system.

Use Case Submission

1. Authors register a paper with the conference and receive a paper number.
2. Authors write and submit the paper to the conference using the paper number.

Use Case Reviewing

1. Reviewers write and submit reviews for their assigned papers.
2. After a reviewer has submitted a review for a paper, he may read other reviews for the same paper and also modify his own review.

6.1.2 Application Security Requirements

The security policy for this application is not designed to meet any environment–specific security requirements. Rather, the focus is on application security exclusively, which is defined here in terms of the *need–to–know* principle. This means that the policy is designed along the application protocols and intended to allow only those interactions that are required by the application and specified in the above use cases. Interactions between actors and the system that are not legal scenarios in these use cases are not permitted, e.g., the policy must ensure that conference steering operations are not performed by any other actor than the chair.

Other implicit restrictions are that existing reviews may only be modified by the reviewer who wrote them, or that author access to the system is restricted to operations for registering and submitting papers. An additional requirement is that there should only be one chair and that the chair is not allowed to act as an author. This last requirement is intended to avoid loyalty conflicts with reviewers.

An important property of this application are its state–based access rules. In particular, authors must not be permitted to modify papers after submitting them or to submit registered papers after the deadline. Also, reviewers must be prevented from submitting more than one review per paper or from reading other reviewers' reviews before submitting their own. The idea behind this last requirement is to shield reviewers from the influence of others to ensure independent reviewing. Reviews can then be aligned with each other to resolve conflicts before the final meeting of the program committee.

6.2 Application design

This section briefly introduces the IDL interfaces that support the functionality described in the previous section. There is no discussion of design decisions here, the presentation is mainly for the purpose of describing the object accesses that later need to be controlled by the access policy. The identified actors are not represented in IDL because communication between actors is supposed to happen through means outside the system rather than through remote invocations. For example, authors receive notifications via e–mail and reviewers discuss their decision in program committee meetings or in telephone or e–mail conversations.

Figure 6.2 lists the main interface in CORBA IDL. The complete set of interfaces for this application can be found in Appendix B.

```
interface ConferenceManagement {
    // navigation
    SubmissionManagement getSubmissionManagement();
    readonly attribute Document callForPapers;
    // use case: ConferenceSteering
    void issueCallForPapers( in string cfp );
    void beginSubmission();
    void deadlineReached();
    void makeDecision();
};
```

Figure 6.2: The ConferenceManagement interface.

In the *ConferenceSteering* use case, the chair interacts with the system using the ConferenceManagement interface to a singleton object of that type. The ConferenceManagement interface supports three operations to switch between processing phases, i.e., to start and end the other two subordinate use cases. The ConferenceManagement interface is also used to retrieve the call for papers and a SubmissionManagement object.

In the *Submission* use case, authors can create a Paper object by calling registerPaper() on the SubmissionManagement object. This operation, which is shown in figure 6.3, raises exceptions if the combination of author name and paper title is already registered or if the registration information is incomplete. The Paper interface in figure 6.4 inherits general document writing operations from the Document interface, which was introduced in chapter 4. Authors can submit using the operation submit() on their paper objects. Figure 6.5 shows a screen shot of a simple graphical user interface that allows authors to retrieve the call for papers, register papers, load a file as a means of writing a paper, and submit the paper.

In the *Reviewing* use case, reviewers can list available papers by calling listPapers() on the SubmissionManagement object. They can then retrieve submissions by calling getPaper() and giving a reference number as an argument. The interface Paper supports additional operations to list and retrieve reviews that have been submitted for this paper. Reviewers submit reviews by calling submitReview() and receive a reference to a Review object in return, which they can

```
interface SubmissionManagement {
    // use case: Submission
    Paper registerPaper( in string author_name, in string title )
        raises( AlreadyRegistered, IncompleteInformation );

    // use case: Reviewing
    PaperIdSeq listPapers();
    Paper getPaper( in long paperNumber );
};
```

Figure 6.3: The SubmissionManagement interface.

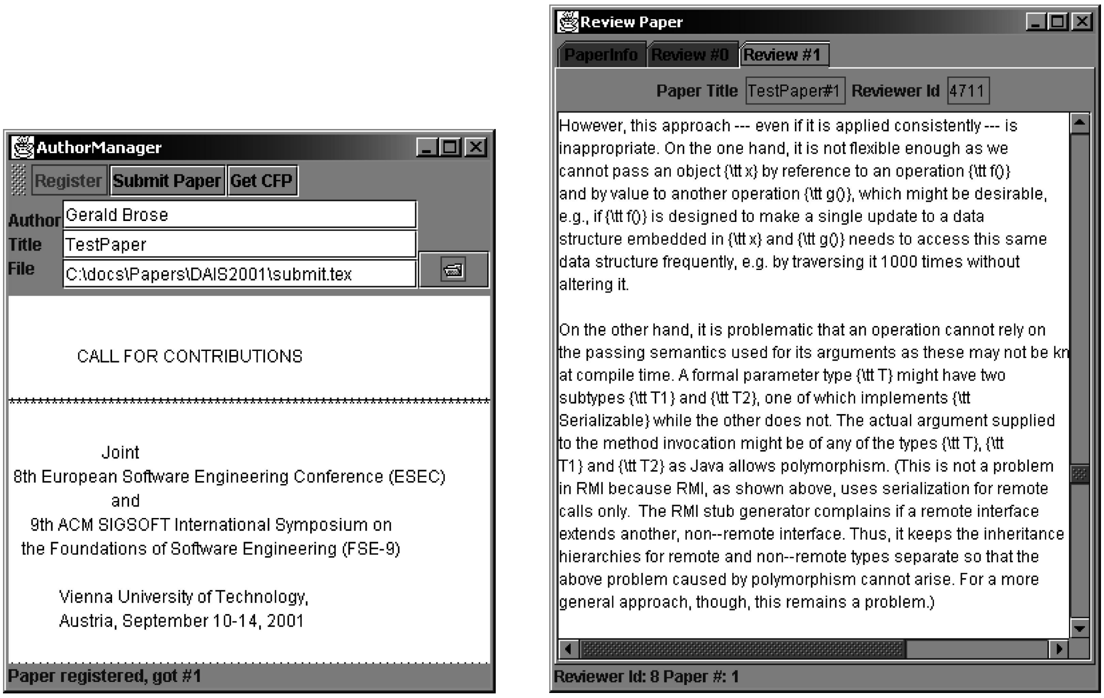
```
interface Paper: Document {
    // use case: Submission
    readonly attribute long number;
    void submit();

    // use case: Reviewing
    Review submitReview(in string review, in long reviewerNumber);
    longSequence listReviews();
    Document getReview(in long reviewerNumber);
};

interface Review: Document {
    readonly ReviewerId reviewer;
};
```

Figure 6.4: IDL Interfaces for the conference application.

use to modify their reviews. After calling submitReview() they may also retrieve the reviews of other reviewers using getReview(). An example graphical tool for reviewers that support submission of reviews is depicted in figure 6.5.



Author: Gerald Brose
Title: TestPaper
File: C:\docs\Papers\DAIS2001\submit.tex

CALL FOR CONTRIBUTIONS

Joint
8th European Software Engineering Conference (ESEC)
and
9th ACM SIGSOFT International Symposium on
the Foundations of Software Engineering (FSE-9)

Vienna University of Technology,
Austria, September 10-14, 2001

Paper registered, got #1

Paper Title: TestPaper#1 Reviewer Id: 4711

However, this approach --- even if it is applied consistently --- is
inappropriate. On the one hand, it is not flexible enough as we
cannot pass an object {\tt x} by reference to an operation {\tt f()}
and by value to another operation {\tt g()}, which might be desirable,
e.g., if {\tt f()} is designed to make a single update to a data
structure embedded in {\tt x} and {\tt g()} needs to access this same
data structure frequently, e.g. by traversing it 1000 times without
altering it.

On the other hand, it is problematic that an operation cannot rely on
the passing semantics used for its arguments as these may not be kn
at compile time. A formal parameter type {\tt T} might have two
subtypes {\tt T1} and {\tt T2}, one of which implements {\tt
Serializable} while the other does not. The actual argument supplied
to the method invocation might be of any of the types {\tt T}, {\tt
T1} and {\tt T2} as Java allows polymorphism. (This is not a problem
in RMI because RMI, as shown above, uses serialization for remote
calls only. The RMI stub generator complains if a remote interface
extends another, non--remote interface. Thus, it keeps the inheritance
hierarchies for remote and non--remote types separate so that the
above problem caused by polymorphism cannot arise. For a more
general approach, though, this remains a problem.)

Reviewer Id: 8 Paper #: 1

Figure 6.5: Author and Reviewer tools.

6.3 Policy design

The design of the access control policy has two main aspects. The first aspect concerns the initial configuration of access rights and is defined by the roles, role constraints, and the initial views held by roles at the time the system starts operation. The second aspect concerns the dynamic evolution of the system's protection state, which is determined by schemas and the views that are assigned and removed by these schemas.

6.3.1 Roles, Constraints, and Initial Views

The actors identified in the use case analysis are directly mapped to the roles in figure 6.6. The *Author* role is meant to be available to all users while the *Chair* role is restricted to allow no more than one role member. Also, the *Chair* and *Author* roles are mutually exclusive. The *Chair* role is a subrole of *Reviewer*, which models the fact that the program committee chair participates in the reviewing process. The two constraints for the *Chair* role express the requirements that there may only be one chair and that the *Chair* may not act as an *Author*.

```
roles
   Author
      holds SubmissionAccessing on ConferenceManagement
      holds Registering on SubmissionManagement
   Reviewer
      holds Member on SubmissionManagement
      holds SubmissionAccessing on ConferenceManagement
      holds PaperReviewing on Paper
   Chair:  Reviewer
      holds Steering on ConferenceManagement
      holds PaperReading on Paper
      maxcard 1
      excludes Author
```

Figure 6.6: Role declarations, constraints, and initial views.

When the system starts operation, the three roles receive the views specified in their **`holds`** clauses. The *Author* role initially holds a `SubmissionAccessing` view, which is defined in figure 6.7. This view permits operations for reading the call for papers and for navigating to the submission management interface. Additionally, authors hold a `Registering` view that allows registering papers. This conditional view is not considered in access decisions at this time, however, because it requires another view `SubmissionPhase`, which authors do not hold at this time. This required view is defined in the next subsection and will be assigned and removed when use cases are dynamically enabled and disabled.

The `SubmissionAccessing` view is also assigned to the *Reviewer* role, which holds an additional `Member` view. This view permits its holders to list and retrieve papers. Both views described so far are assigned on entire type extensions, but both these extensions are

```
view SubmissionAccessing
   controls ConferenceManagement
{
   allow
      getSubmissionManagement
      callForPapers
}
view Member
   controls SubmissionManagement
{
   allow
      listPapers
      getPaper
}
view Registering
   controls SubmissionManagement
   requires SubmissionPhase
{
   allow
      registerPaper
}
```

Figure 6.7: Views.

assumed to have only a single instance at runtime of the system. Finally, the *Reviewer* role holds a `PaperReviewing` view, which is shown in figure 6.8. Like the `Registering` view held by authors, this view requires another view, so reviewers cannot use the permissions of this view at this stage.

```
view PaperReviewing:  PaperReading
   controls Paper
   restricted_to Reviewer
   requires ReviewingPhase
{
   allow
      listReview
      submitReview
}
```

Figure 6.8: The `PaperReviewing` view.

The *Chair* role holds an initial `Steering` view for the ConferenceManagement object and a `PaperReading` view on all Paper objects. These views are defined in figure 6.9. The `Steering` view contains the rights to issue a call for papers and to switch between processing phases. It is restricted to the *Chair* role to emphasize that these operations are for exclusive use by the chair. The `PaperReading` view, which extends the `Reading` view, is assigned on the extension of Paper and permits the chair to read any paper as soon as it becomes a member of the policy domain.

```
view Steering:  SubmissionAccessing
   controls ConferenceManagement
   restricted_to Chair
{
   allow
      issueCallForPapers
      beginSubmission
      deadlineReached
      makeDecision
}

view Reading                    view PaperReading:  Reading
   controls Document                  controls Paper
{                               {
   allow                           allow
      read                            number
      title                           author
}                                     listReviews
                                }
```

Figure 6.9: Views on documents and papers.

6.3.2 Dynamic aspects

A characteristic feature of this application policy are the regular dynamic changes in the protection state. The first source of changes is the ordering of use cases and the way these use cases are enabled and disabled in the encompassing workflow. The second source of rights changes are requirements from the application policy, viz. that authors lose the right to modify a paper when it is submitted and that reviewers get rights when reviews are submitted: before this point, reviewers may not read other reviews; from then on, they may. They also lose the right to submit a second review for the same paper at this point. Which accesses are permitted thus depends on earlier accesses, similar to the *Chinese Wall* policy [Brewer and Nash, 1989].

When use cases are enabled and disabled, a number of views need to be either assigned or enabled, or removed or disabled, at the same time. This contrasts with more fine–grained changes that affect individual objects and principals when papers or reviews are submitted. Therefore, these different kinds of changes are modeled using different design approaches in VPL. Switching between processing phases and thus between use cases is modeled using the `Steering` schema. Figure 6.10 shows this schema, which describes how the protection state changes in reaction to operations in the ConferenceManagement interface.

If issueCallForPapers() is called, all roles receive a `Reading` view on the call for papers document, which is referenced as `this.callForPapers`, i.e., using its attribute name in

```
schema Steering
   observes ConferenceManagement
{
   issueCallForPapers
      assigns
         Reading on this.callForPapers
            to Author, Chair, Reviewer
   beginSubmission
      assigns
         SubmissionPhase on Object to Author
   deadlineReached
      assigns
         ReviewingPhase on Object to Reviewer
      removes
         SubmissionPhase on Object from Author
   makeDecision
      removes
         ReviewingPhase on Object from Reviewer
}

virtual view SubmissionPhase
virtual view ReviewingPhase
```

Figure 6.10: The `Steering` schema and virtual views.

the ConferenceManagement interface. Note that a reference to the call for papers can be obtained by authors even before issueCallForPapers() is called because the *Author* role holds an initial SubmissionAccessing view which allows this, but the read() operation on this document was not allowed at this time.

To enable the submission use case, the chair calls beginSubmission(), which will assign the virtual `SubmissionPhase` view on all objects to the *Author* role. This view, which is defined in figure 6.10, is removed again when the chair calls deadlineReached(). This operation, in turn, triggers the assignment of the virtual view `ReviewingPhase`, which enables the *Reviewing* use case. This view is removed again when the chair finally calls makeDecision() to disable the *Reviewing* use case. The important point here is that the use of conditional views permits a policy design where only a single view is assigned and another single view removed at each change of processing phases. Since a number of other views depend on these virtual views, a collective change of fine–grained authorizations is achieved without complex schema clauses.

6.3.2.1 The Submission Use Case

After the virtual `SubmissionPhase` view is assigned to the *Author* role, authors can use the permissions in their initial `Registering` view. Thus, they can register papers by calling the registerPaper() operation on the SubmissionManagement object. To model the more fine–grained authorization changes connected with registering and submitting papers, a second schema is defined in figure 6.11 for the submission phase. When the registerPaper() operation is called, the `Submission` schema assigns a `PaperSubmitting` view on the Paper object that is returned as the result of the operation to the caller, which means that authors can now call submit() on the paper. To allow modifications, i.e., to enable writing of the paper, authors also receive a `Modifying` view on the paper.

```
schema Submission
   observes SubmissionManagement
{
   registerPaper
      assigns
         PaperSubmitting, Modifying on result to caller;
}

view PaperSubmitting: PaperReading      view Modifying: Reading
   controls Paper                          controls Document
   restricted_to Author                 {
   requires SubmissionPhase                allow
{                                             update
   allow                                      write
      submit                                  append
}                                       }
```

Figure 6.11: The `Submission` schema and views for authoring.

To control rights changes caused by operations on papers, a further schema `PaperSchema` is defined. Figure 6.12 shows the part that is used in the submission use case. When the submit() operation is called, the schema assigns a `PaperReading` view on the paper to all reviewers. At the same time, the submitting author loses the `Modifying` view that he received upon registering the paper.

```
schema PaperSchema
   observes Paper
{
   submit
      assigns
         PaperReading on this to Reviewer
      removes
         Modifying on this from caller
   // ...
}
```

Figure 6.12: The `Paper` schema (submission part).

The *Submission* use case ends when the deadline is reached and the chair invokes deadlineReached() to disallow further submissions. The `Steering` schema in figure 6.10 removes the `SubmissionPhase` view from the *Author* role at this time, which disables the `Registering` view. However, disabling the `Registering` view is not sufficient to prevent authors from submitting papers that are already registered at this time because submissions are controlled by the `PaperSubmitting` view. Because this view was assigned to individual callers and not to the entire *Author* role, it cannot be removed from all authors at once when the submission phase ends. However, it can be *disabled* just like the `Registering` view because it also requires the `SubmissionPhase` view, so it simply becomes unusable together with the `Registering` view.

6.3.2.2 The Reviewing Use Case

Calling deadlineReached() to end the *Submission* use case also enables the *Reviewing* use case because the `Steering` schema in figure 6.10 assigns the virtual view `ReviewingPhase` on all objects to the *Reviewer* role. Reviewers may now use the permissions in their `PaperReviewing` view, which required the `ReviewingPhase` view. The `PaperReviewing` view permits reviewers to submit their reviews to any paper, but it is assumed that reviews are only submitted according to a predefined distribution of papers to reviewers. The view also allows reviewers to find out about existing reviews for any paper. Because the application policy stated that reviewers lose their right to submit further reviews for the same paper after submitting a review and that they may also read other reviews after handing in their own, corresponding schema clauses for the submitReview() operation on papers need to be defined. These are shown in figure 6.13.

```
schema PaperSchema
   observes Paper
{
   //...
   submitReview
      assigns
         Modifying on result to caller
      assigns
         ReviewReading on result to Reviewer
      assigns
         ReviewAccess on this to caller
      assigns
         NoMorePaperReviewing on this to caller
}
```

Figure 6.13: The `Paper` schema (reviewing part).

The schema defines that a submitting reviewer receives a `Modifying` view on the Review object that is the result of the submitReview() operation. While the caller may now update his own review, the entire *Reviewer* role is assigned a `ReviewReading` view on the new review object, which is shown in figure 6.14.

```
view ReviewAccess
   controls Paper
   restricted_to Reviewer
{
   allow
      getReview
}

view ReviewReading:  Reading
   controls Review
{
   allow
      reviewer // access to an attribute
}
```

Figure 6.14: Views for the reviewing phase.

This view assignment seems to violate the policy that other reviewers may only read a review after submitting their own reviews for the same paper, and this view indeed allows any reviewer who has access to the new review to read it. However, the only reviewers who have access to the object itself are its author and those reviewers who also have the right to call getReview() to retrieve reviews for this paper. Because the getReview() operation is only allowed by the `ReviewAccess` view and this view is only assigned upon calling submitReview(), exactly those reviewers can use the `ReviewReading` view who have already submitted their own reviews.

Finally, reviewers must be prevented from submitting further reviews for the same paper. This cannot be done by simply removing the `PaperReviewing` view because this view was assigned to the *Reviewer* role rather than individually, so removing it would prevent *all* reviewers from submitting further reviews. Moreover, it would also remove the right to call listReview(). The solution here is to individually assign a view that contains a denial and overrides the permission for submitReview(). The view defined for this purpose is the `NoMorePaperReviewing` view in figure 6.15. The denial for submitReview() will take precedence because the two views that contain the permission and the denial, `PaperReviewing` and `NoMorePaperReviewing`, are not related by view extension. The conflict resolution strategy defined in chapter 4 stated that denials take precedence in such a situation. Since `NoMorePaperReviewing` is assigned to the individual caller, the denial is visible to the access decision function regardless of which roles the caller actually activates, i.e., the denial cannot be avoided by selecting a different set of roles.

The *Reviewing* use case ends when the chair finally calls the makeDecision() operation. As defined by the `Steering` schema in figure 6.10, the `ReviewingPhase` view is removed from the *Reviewer* role, which disables the `PaperReviewing` view. The reviewing support of the conference application ends at this stage. Any remaining conflicts between reviews for

```
view NoMorePaperReviewing
   controls Paper
{
   deny
      submitReview
}
```

Figure 6.15: The `NoMorePaperReviewing` view.

the same paper must be resolved by direct communication; the selection of accepted papers itself is also not supported by the application.