

Declarative Access Control for WebDSL

MSc Thesis

Version of April 4, 2008

Danny Groenewegen

Declarative Access Control for WebDSL

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Danny Groenewegen
born in Nootdorp, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Declarative Access Control for WebDSL

Author: Danny Groenewegen
Student id: 1149903
Email: dgroenewegen@gmail.com

Abstract

This thesis describes the extension of WebDSL, a domain-specific language for web application development, with abstractions for declarative access control. The extension supports the definition of a wide range of access control policies concisely and transparently as a separate concern. Access is governed to the various resources in a WebDSL application, most importantly the pages and page actions. Besides direct specification of access control, certain access control checks are inferred, for instance to control the visibility of navigation links within the application. This extension shows an approach for designing a domain-specific language to support separation of concerns while preserving static validation. This approach is realized by means of a transformational semantics that weaves separately defined aspects into an integrated implementation.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. K. Langendoen, Faculty EEMCS, TU Delft

Preface

I would like to thank my supervisor, Eelco Visser, for setting up a great project in which I could implement my ideas for an access control language, and motivating me to work on a paper together to report the findings to the research community. Also I would like to thank the other WebDSL developers Sander van der Burg, Zef Hemel, Jippe Holwerda, Lennart Kats, Wouter Mouw, and Sander Vermolen for their support during the creation of the WebDSL access control extension and creating a fun and productive working environment. Finally, a big thanks to my parents and my girlfriend for their support during my time as a student.

Danny Groenewegen
Delft, the Netherlands
April 4, 2008

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
2 WebDSL	5
2.1 Data Models	6
2.2 Page Definitions	7
2.3 Template Definitions	8
2.4 Actions	9
2.5 Access Control	11
3 An Access Control Sub-Language for WebDSL	13
3.1 Authentication	14
3.2 Access Control Rules	16
3.3 Data Structures for Access Control	20
3.4 Inferring Visibility	21
3.5 Administration	22
3.6 Policy Combination	22
4 WebDSL Access Control Syntax and Semantics	25
4.1 Syntax	25
4.2 Principal Declaration	26
4.3 Rule Weaving	27
4.4 Reuse Desugaring	32
4.5 Inferred Access Control	34
4.6 Policy Combination	35
4.7 Program Validation	36

5	Policy Implementations	39
5.1	Mandatory Access Control	39
5.2	Discretionary Access Control	46
5.3	Role-Based Access Control	52
6	Case Studies	59
6.1	Wiki	59
6.2	Social Networking Service	62
6.3	Conference System	65
7	Related Work	73
7.1	Language Design	73
7.2	Policy Languages	75
7.3	Frameworks	75
8	Discussion	77
8.1	Evaluation	77
8.2	Future Work	78
9	Conclusion	79
	Bibliography	81
A	Glossary	85

List of Figures

2.1	View document page	8
2.2	Menu	10
2.3	Edit document page	10
2.4	Create document page	11
3.1	Login	15
3.2	Logout	16
3.3	Inferring visibility for menu links	21
3.4	Access control administration	22
4.1	Syntax	26
4.2	Access denied	28
4.3	Page rule transformation	29
4.4	Action rule transformation	30
4.5	Template rule transformation	30
4.6	Function rule transformation	31
4.7	Page-action rule transformation	32
4.8	Template-action rule transformation	32
4.9	Pointcut transformation	33
4.10	Predicate transformation	33
4.11	Conditional navigation	34
4.12	Rule normalization	35
4.13	Combining rule sets with OR	35
4.14	Combining rule sets with AND	36
5.1	Dominance	40
5.2	Confidentiality information flow	41
5.3	Integrity information flow	42
5.4	Security lattice	42
5.5	MAC clearance level 1	44
5.6	MAC clearance level 2 with menu	44

5.7	MAC sidebar clearance level 2	45
5.8	Access-Matrix	47
5.9	Access Control List	47
5.10	List of capabilities	47
5.11	Authorization table	48
5.12	Manage view and edit permissions	50
5.13	Manage granting permissions	50
5.14	Owner of object with all rights	51
5.15	Resulting navigation links	51
5.16	Role activation	55
5.17	View role activated	55
5.18	Edit role activated	55
5.19	Admin role	56
5.20	Admin role activated	56
6.1	Edit page	60
6.2	Wiki not logged in	61
6.3	Wiki logged in	61
6.4	All pages	61
6.5	Social network home	64
6.6	Your own page	64
6.7	Friends	64
6.8	Administration	65
6.9	More administration	65
6.10	Conference overview	67
6.11	Chair menu	69
6.12	Declare a conflict among users	69
6.13	Chair view of submitted paper	71
6.14	Reviewer view of submitted paper	71

Chapter 1

Introduction

Access control is essential for the security of web applications. Web applications have especially important security requirements, because of their global accessibility. The requirements of some web applications consist of a simple protected and unprotected area, in this case the web server's security options are sufficient, which could simply allow or deny access based on URL patterns. More often however, web applications require more sophisticated controls to govern access. These requirements cannot be implemented using the web server's security features, and the application programmer will have to embed an access control solution in the application.

The identity of a user is often the basis for access control decisions. The process of identifying a user in the system is called authorization. Applications use authorization information also for normal operation, which makes the access control part of an application closely integrated. Not only user data but also other information available in the application can influence the outcome of access control decisions, for instance, ownership of an object.

Access control policies are provided as natural language descriptions resulting from application design or organizational requirements. These policies have to be implemented in applications, which requires a conversion of the document to the elements of the mechanisms available in the implementation language. If the mechanism is low-level, this conversion is very sensitive to errors. Validation of the policy's implementation becomes hard to accomplish. This indicates that the mechanisms for access control should be controlled by high-level descriptions, in order to be verifiable and straightforward to implement.

Ideally access control should be specified separately, but it also has to integrate with the rest of the application. This situation is not unique for the access control aspect of an application. It is common practice in web application development to have multiple languages with specific purposes working together. For example, HTML and CSS for page presentation, Java for generating the pages, SQL for communicating with the database. These languages look entirely different and mix declarative and imperative programming styles. Besides different language styles, another issue is the lack of integration across different languages. For example, Java is statically checked, but XML configuration files for frameworks are not taken into account in this checking.

In practice, existing solutions for access control for web applications are either separate policy languages, which cannot be seamlessly integrated, or application frameworks, which

do not support high-level definition of policies. Access control policy languages such as Ponder and XACML [5, 13] are often implemented as an autonomic system that can be queried for access control decisions. This approach makes it hard to support flexible access control. All the information needed for a check has to be explicitly transferred to the policy engine. Furthermore, these frameworks do not aid in separating checks from the main application; the actual check invocations are still scattered across the application code. Finally, the complexity of these policy languages (e.g., the RBAC template for XACML [2]) decreases their readability, resulting in unclear policies.

Web application frameworks such as Spring/Acegi [9, 1] and Seam [28] support an aspect-oriented approach to access control, separating access control code from application code. These frameworks supply more or less fixed role-based [19, 18] and discretionary [17, 20] access control configurations. Extending the built-in policy or creating other types of policies requires manually implementing these as an extension of the underlying object-oriented framework. Such framework extensions are hard to test and require knowledge of the framework to be able to understand the policies.

Generic aspect languages such as AspectJ [10] offer separation of concerns while preserving the integration with the language, with syntax and static analysis. These generic aspect solutions do not, however, give benefits related to specific domains. In the case of access control, no specific support for encoding of policies is given in the aspect language.

In previous work WebDSL has been developed, a domain-specific language for development of web applications with a rich data model [26]. The language supports separation of concerns by providing sub-languages catering for the different technical domains of web engineering. Linguistic integration of these sub-languages ensures seamless integration of the aspects comprising the definition of a web application.

In this thesis, the extension of WebDSL with abstractions for declarative definition of access control is presented. The access control policy for an application is defined separately from the data model and user interface using declarative rules. While access control rules are defined as a separate concern, the extension is linguistically integrated. That is, access control rules use the same expression language, which refers to the same data models that are used in the rest of an application. Furthermore, access control checks are integrated into the implementation, which allows rules to access the complete object graph, instead of requiring selected data to be sent to a separate engine. Rather than catering for a fixed policy, the extension provides the basic mechanisms for encoding a wide range of access control policies that can be adapted to the requirements of the application and integrated with its data model. Finally, the declarative and domain-specific nature of rules allows us not only to restrict access to pages and actions, but also to adapt the navigation options presented to users to prevent them from navigating to inaccessible pages.

The main contributions of this thesis are: (1) The general approach of designing linguistically integrated domain-specific languages for different technical domains, realized by means of a transformational semantics that reduces separately defined aspects into an integrated implementation. (2) The design of an access control sub-language for expressing a wide range of access control policies concisely and transparently as a separate concern. (3) The use of access control semantics for reducing development effort. The developer can concentrate on the logical design of navigation, leaving the modality of navigation to the

access control rules.

The structure of this thesis is as follows: Section 2 introduces WebDSL and discusses the implementation of an example application. Section 3 describes the access control sub-language and illustrates this with an extension of the example application. Section 4 describes syntax and semantics of the access control sub-language. Section 5 discusses the main access control policy types and gives implementation examples in the access control sub-language. Section 6 shows three web applications implemented in WebDSL together with access control policies. Section 7 covers related work. Section 8 evaluates the access control sub-language and discusses issues and future work. Section 9 concludes this thesis.

Chapter 2

WebDSL

WebDSL [26] is a domain-specific language for implementing dynamic web applications with a rich data model. WebDSL consists of multiple sub-languages that each define a different aspect of the web application. Sub-languages describe data models, pages, operations on data, and the sub-language discussed in this thesis adds access control. Normally one would also have to use several languages for creating a web applications, e.g., (X)HTML, Javascript, CSS, and a language that implements an engine for generating those languages dynamically, such as Java or PHP. These are all connected at run-time to each other and running the applications is often the only way to find errors. In the case of WebDSL the sub-languages are statically integrated. Compile-time checks are performed that connect the different sub-languages.

WebDSL is implemented by a code generator that transforms a WebDSL application into a Seam framework web application. The Seam [28] framework is mainly based on Java technologies, e.g., Java Server Faces (JSF) for presentation, and Java Persistence API (JPA) for persistence of data in a database. Seam combines the different technologies through scoped contexts in which components can get registered and retrieved by a name. These contexts are directly accessible from the different implementation languages, for example Java and JSF, which constitutes the link between the implementation artifacts. This link is dynamic and only checked during run-time, which makes the development of Seam applications involve a lot of deploy, run, and correct cycles. The good thing about using a framework like Seam is that a lot of knowledge applicable to creating web applications in Java is available in it, which means a smaller conceptual gap between the WebDSL application and the generated code compared to generating plain Java Servlets. Data models are described using entity definitions in WebDSL, these are translated to Java classes configured as entities for JPA. This involves adding the properties, getters, setters (sometimes extra operations are necessary here, e.g., in the case of inverse relations), and object-relational mapping annotations. WebDSL pages are transformed to a JSF XHTML page, a Seam Java bean class for entity retrieval and operations, and an interface for this bean class as required by the Seam framework. Coherent parts of a WebDSL application can be put in different files/modules that can be included, to provide modularization of the application code. The generated Seam applications can be deployed on the JBoss application server. The screenshots in this thesis will show a browser interacting with the Seam application.

The code generator is implemented with the syntax definition formalism SDF [24] and the transformation language Stratego [25]. These will not be discussed in detail, Chapter 4 will describe the syntax and transformation semantics used for the access control sub-language in an implementation-independent way.

In this thesis WebDSL is extended with a sub-language for access control. This chapter will explain the elements in WebDSL that are of interest for access control. First the entities that represent the data model are covered in Section 2.1. Section 2.2 covers the page definitions that represent the interface of the application. Section 2.3 discusses the reuse capabilities in WebDSL and how to set up a consistent application look. Section 2.4 shows how operations are performed on data using actions. Section 2.5 concludes with an example of how access control can be done manually.

2.1 Data Models

The data model used in a WebDSL application is specified using named entity declarations. These entities are the data that is persisted to a database and are transformed to JPA entity classes. The entities are declared with a list of properties that consist of a name and a type. Properties can be value types, which use the built-in types like `String`, `Date`, `Int`, and `Bool`, but also domain-specific types such as `Password` (stored as digest, which can be compared but not read), and `WikiText` (allowing markup language and page links to be used). Value types are indicated by `::` between property name and type in the declaration. Besides value types properties can also be associations to other entities of the data model. The most common type of association is a reference to another entity, which is indicated by a `->`. This association doesn't influence the referred object when it is removed. Composite associations, indicated by `<>`, cause the associated object to be part of the associating entity alone, which allows cascading deletes (when removing the association or the associating entity the associated entity can be deleted). Collection types can be specified using `Set` or `List`, which can refer to either value types or defined entities (creating an association type). The `inverse` annotation can be used to declare a relation between two properties of different entities, which will synchronize them whenever changes are made to either of them. A special type of entity is the session entity, declared by using `session` instead of `entity`. This entity becomes a (not persisted) singleton in the session of the user with the WebDSL application. Entities can be extended with extra properties by using `extend entity`. This performs a simple insertion of the extra properties in the original entity declaration.

In the following WebDSL data model two simple entities are shown:

```
entity Document {
  title :: String
  text  :: String
  author -> User
}
entity User {
  name    :: String
  password :: Password
}
```

The Document entity has three properties, title and text are simple strings, and author is a reference to another entity. The password property of the User has type Password, which indicates that it shouldn't be made visible when viewing and that it is stored as a digest. Some instances of the defined entities can be initialized during startup of the application, which makes it easy to put some test data in:

```
var alice : User := User {
  name := "Alice"
  password := "changeme"
};
var d0 : Document := Document {
  title := "First Document"
  text := "This is the first text."
  author := alice
};
```

2.2 Page Definitions

Page definitions are the main component for the interface of a WebDSL application. These definitions consist of a name for the page, the arguments supplied to it, and the elements that make up the presentation and operations of the page. Arguments can be entity references or simple values. Pages are built with basic markup elements such as section, header, table, and list. Data from arguments or local to the page can be displayed using output. Forms are an important part of page definitions because they connect buttons or links with operations. A form usually holds some input elements that determine the data to be specified by the user in that form. Upon submitting a form, the user's inputs are automatically inserted into the instances specified in the input (it is possible to add validation in the data model that is automatically performed for these insertions). Navigation to other pages can be defined using navigate elements. These require a page name, actual arguments for the page, and a text to represent the link. When iterating over collections for loops can be used in page definitions. For instance, this allows one to easily display the contents of a collection with links to related pages. Also filtering and sorting of the items is supported.

The following page definition defines a view page for a Document entity:

```
define page viewDocument(document:Document) {
  main()
  define body() {
    section {
      header{ output(document.name) }
      table{
        row{ "Title:" output(document.title) }
        row{ "Text:" output(document.text) }
        row{ "Author:"
          navigate(viewUser(document.author)){
            output(document.author.name)
          }
        }
      }
    }
  }
}
```

The page takes one `Document` argument (which corresponds to a request parameter of the resulting page). A `main()` template is called, which constructs the common parts of the application (see Section 2.3 for the definition of this template). The body is then redefined locally to create a view of the `Document` entity. The author's name is made a navigation link to the `viewUser` page of that author. Figure 2.1 shows the page that results from this code.

1

Title:	First Document
Text:	This is the first text.
Author:	Alice

Figure 2.1: View document page

2.3 Template Definitions

To allow reuse of groups of elements in page definitions it is possible to declare and use templates. We have already seen template definitions, because page definitions are a special type of template definitions which cannot be included in other pages. This limitation does not hold for normal template definitions, they can be included in other page or template definitions. Using arguments the templates can be specialized for particular calls, template code is inserted with actual arguments substituted. Templates can be locally redefined at the calling site, which allows flexible extension for specific situations.

For illustrating the templates, an interesting template is the `main` described here, which creates a consistent application look, and allows another definition to hook by (re)defining `top`, `sidebar`, `body`, or `footer`:

```
define main() {
  block("top") { top() }
  block("body") {
    block("left_innerbody") { sidebar() }
    block("main_innerbody") { body() }
  }
  block("footer") { footer() }
}
```

In the current WebDSL implementation positioning blocks has to be done using CSS (better style/layout definitions and abstractions are still in development). The template calls in the `main` template need a sensible default:

```
define footer() {}
define top() {
  block("header") {}
  block("menubar") { menubar{ applicationmenu() }}
```

```

}
define body() {}
define sidebar() {
  list{
    listitem{ navigate(home()){ output("home") }}
  }
}

```

A common way of working with this standard template is to redefine `body` in pages after `main` is called (for instance in the `viewDocument` page code in Section 2.2). It also becomes easy to create custom menus/sidebars for certain parts of the application. The `applicationmenu` template contains all the navigation needed for this simple WebDSL application:

```

define applicationmenu() {
  menu {
    menuheader{ "View Document" }
    for(d:Document) {
      menuitem { navigate(viewDocument(d)){ output(d.title) }}
    }
  }
  menu {
    menuheader { "Edit Document" }
    for(d:Document) {
      menuitem{ navigate(editDocument(d)){ output(d.title) }}
    }
  }
  menu { menuheader{ navigate(createDocument()){ output("New document") }}}
  menu {
    menuheader { "View User" }
    for(u:User) {
      menuitem{ navigate(user(u)){ output(u.name) }}
    }
  }
  menu {
    menuheader {"Edit User"}
    for(u:User) {
      menuitem { navigate(editUser(u)){ output(u.name) }}
    }
  }
  menu { menuheader { navigate(createUser()){ output("New user") }}}
}

```

The resulting menu is shown in Figure 2.2.

2.4 Actions

So far the pages only displayed information, but a WebDSL application can also have operations that modify information. These are defined in actions, which support an imperative language for defining operations. The actions are nested in pages, a form is needed to provide a connection between a button or link to an action. The `input` element can be used to connect data from variables to form elements. These are submitted and synchronized with the data of the page before the action is executed (this automatic data binding is used in JSF as well).

View Document	Edit Document	New document	View User	Edit User	New user
----------------------	---------------	--------------	-----------	-----------	----------

First Document
Second Document
Third Document
Fourth Document

document

Text: This is the first text.

Figure 2.2: Menu

The `EditDocument` page has a simple action that simply saves the changes and returns to the `viewDocument` page of the Document being edited:

```

define page editDocument(document:Document) {
  main()
  define body() {
    section {
      header { "Edit Document " output(document.title) }
      form {
        table {
          row{ "Title:" input(document.title) }
          row{ "Text:" input(document.text) }
          row{ "Author:" input(document.author) }
          allowedUsersRow(document)
        }
        action("Save", save())
      }
    }
    action save() {
      document.save();
      return viewDocument(document);
    }
  }
}

```

The `EditDocument` page is shown in figure 2.3.

Edit Document First Document

Title:

Text:

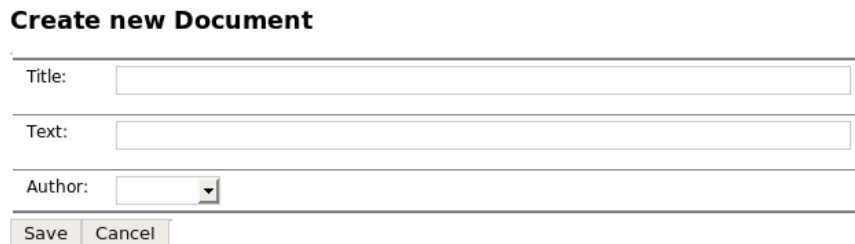
Author:

Figure 2.3: Edit document page

Another page that needs an action for defining an operation on data is the `createDocument` page:

```
define page createDocument() {  
  main()  
  define body() {  
    var document : Document := Document{};  
    section {  
      header { "Create new Document" }  
      form {  
        table {  
          row{ "Title:" input(document.title) }  
          row{ "Text:" input(document.text) }  
          row{ "Author:" input(document.author) }  
        }  
        action("Save", save())  
      }  
      action save() {  
        document.save();  
        return viewDocument(document);  
      }  
    }  
  }  
}
```

The document variable here shows that variables can be locally declared in the page as well which is useful for initializing an entity for creation. The createDocument page is shown in Figure 2.4.



Create new Document

Title:

Text:

Author:

Figure 2.4: Create document page

2.5 Access Control

Access Control in WebDSL can be done programmatically by specifying conditionals in the page and action definitions. It is possible to hide page content by using an if template in a page definition, in this example the navigation link is only shown for the author with name Alice:

```
if(author.name = "Alice") {  
  navigate(viewUser(document.author)){ output(document.author.name) }  
}
```

Actions can be protected using the if statement:

```
action save() {  
  if(document.author.name = "Bob") {  
    document.save();  
    return viewDocument(document);  
  }  
  else {  
    return editDocument(document);  
  }  
}
```

This will prevent the document from being saved unless the author's name is Bob.

Enforcing an access control policy in this manner involves messy and duplicated code. It makes the actual policy being enforced hard to recover because it is scattered all over the application. Testing the application also becomes more complicated when access control has to be taken into account everywhere. However, it is useful to recognize how access control can be done programmatically because our declarative access control involves generating code for the programmatic way of doing access control.

Chapter 3

An Access Control Sub-Language for WebDSL

In web applications access control is needed to shield parts of the application from users that are not allowed there. Some web applications have a light access restriction, wikis often allow anyone to edit pages, but other web applications enforce heavy restriction of access, for example an online banking application. The policies that govern access to the application's resources are encoded in the application. These access control policies are crucial for the correct functioning of the application.

The definitions of policy, model, and mechanism play an important role in any discussion related to access control. A policy is a high-level specification of the way the system should allow access to its users. A model is a formal representation of a policy, which is useful for analyzing policy properties and building a reference for types of policies. A mechanism is a low-level implementation artifact used to enforce an access control policy in a system. WebDSL access control aims to bring mechanisms to higher abstractions, and thus minimize the gap between mechanism and policy. Well-known access control models will be used in Chapter 5 to validate the conciseness and flexibility of WebDSL access control.

Existing access control systems (see related work, Chapter 7) are either separated completely from the application, or provided mechanisms are low-level. Access control systems with policy description languages like Ponder and XACML have high-level mechanisms but lack a strong connection with the application. Any data needed for access control decisions needs to be passed in requests which can become inefficient and is prone to errors. These languages also suffer from a lot of complexity to compensate the lack of constructive features, e.g., role concepts are often built-in and hard to deviate from. Other types of access control systems, for example those supplied by the web application frameworks Spring/Acegi and Seam, have a clear connection with the application, but consist of encoding policies in aspect-oriented programming constructs. The problem here is that the description of these mechanisms is hard to relate to the policy they are meant to implement, which makes reviewing or adjusting the policy a difficult task.

The goal of WebDSL access control is to have a flexible and extensible way of implementing application-specific policies using high-level mechanisms. It should be integrated

with the system to avoid any possible inconsistencies and allow an efficient way of enforcing access control. This integration also allows having an easily configurable access control policy, with administration integrated into the application.

This chapter will give an overview of the WebDSL access control sub-language. The language features and their motivation are covered here together with an example of how to build mechanisms for policy enforcement. The example mechanism will add access control to the WebDSL application from Chapter 2. Section 3.1 introduces the authorization mechanism of WebDSL access control. Section 3.2 covers the access control rules, which mainly define the access control policy. Section 3.3 covers the data used for access control. Section 3.5 shows how an access control policy can be administrated within WebDSL. Section 3.6 discusses the possibility of combining policies.

3.1 Authentication

Authentication is the process of identifying a user in the system, which plays an essential part in access control policies. If authentication is not secure, an access control system can't protect the resources because any information related to users and subjects might be wrong. There are multiple ways to authenticate a user, proving his or her identity:

- Ask something only the user should know: this is the easiest authentication, usually consisting of asking the user a password.
- Check whether the user possesses a certain object: for web applications this could be certain information existing in a cookie. Normally some security card is often used for this type of authentication.
- Check aspects of the user: this is also called biometrics and involves comparison of finger prints, iris scans, or other of the users' features.

Web applications mostly use password authentication (based on user's knowledge). When authentication is completed, the result needs to be stored somewhere. In WebDSL access control the result is stored in the `principal` property of the `securityContext` session entity. Also a predicate `loggedIn()` is available which simply checks whether `principal != null`.

When a WebDSL application is compiled, the access control system needs to know which entity corresponds to a user. The following line specifies this:

```
principal is User with credentials name, password
```

The principal is the entity that can log in and is available via the `securityContext.principal` property. The credentials in the principal declaration are used to generate an authenticate function automatically. The login template and the generated authenticate function are shown here:

```
function authenticate(name : String, pw : Secret) : Bool {
  var users : List<User>
  := select u from User as u where (u.username = ~name);
```

```

    if ( users.length = 1 && users[0].password.check(pw) )
      { securityContext.principal := users[0]; return true; }
    else { return false }
  }
  define login() {
    var usr : User := User{};
    form {
      table {
        row{ "Name: " input(usr.name) }
        row{ "Password: " input(usr.password) }
        row{ action("Log In", login()) " " }
      }
      action login() {
        if (authenticate(usr.name,usr.password)) {
          return viewUser(securityContext.principal);
        }
        return accessDenied();
      }
    }
  }
}

```

This template consists of a page variable of type `User` (which is also the type of the principal), a form that fills the `User` page variable, and an action verifying the credentials and registering the `User` in the `securityContext` if authentication is successful. The user is either returned to his or her own view page when login is successful or to the home page if authentication failed (which contains the login template). The login template as represented in the browser is shown in Figure 3.1.

Figure 3.1: Login

The logout template takes care of invalidating the principal in the `securityContext` and thus revoking the authentication result:

```

define logout() {
  "Logged in as " output(securityContext.principal)
  form {
    actionLink("Log Out", logout())
    action logout() {
      securityContext.principal := null;
      return home();
    }
  }
}

```

The logout template also consists of a form with an associated action. The resulting web interface is shown in Figure 3.2.

View Document	Edit Document	New document	View User	Edit User	New user
Welcome to the example application which manages users and documents. The supported operations are creation, viewing, and editing. Logged in as Alice Log Out					

Figure 3.2: Logout

The templates defined here for login and logout and the declaration of the principal provide the access control system with an authentication subsystem. With this subsystem the actual access control policy can be enforced.

3.2 Access Control Rules

WebDSL access control is driven by rule declarations. These declare the conditions for allowing access to resources. Before exploring the rules, the environment the rules work in will be explained.

Although WebDSL access control can define any policy (policy neutral), it uses a closed policy for the overall mechanism. A closed policy is one where access is denied by default and (positive) authorizations specify what is allowed. An open policy on the other hand is one where access is allowed by default and can be denied by (negative) authorizations. Open policies can be simulated easily by having a rule that matches all resources and allows access, the other authorization rules can limit the accesses further. The choice for a closed policy by default is based on the security principle of least privilege. In the access control context this entails allowing no more than necessary, which corresponds naturally with a closed policy.

3.2.1 Rule Structure

The rule definitions are the main component of WebDSL access control. They specify where to enforce policy and what the condition for allowing access to the protected resource is. As mentioned earlier, the default is to deny access. For WebDSL the resources are the pages, actions, templates, and functions. All instances of these resources are disabled by default by the access control system. An extra advantage is that automatically generated pages are disabled by default, so you don't have to worry security issues from pages you didn't define directly.

The decision making in rules is specified by expressions. These expressions reuse the expressions from the action sub-language of WebDSL. This offers a couple of important advantages:

- A standard way of creating expressions in the entire WebDSL application.
- Access control grows with WebDSL when the expression language is enriched with new elements/syntactic sugar.

- Ease of implementation, WebDSL access control can reuse several important language implementation parts like the type checker and code generator.

A simple rule protecting the `editUser` page to be only accessible to the user being edited, can be created as follows:

```
access control rules
  rule page editUser(u:User) { u = principal }
```

An analysis of this rule:

- `access control rules`: Indicates the start of a (named) set of access control rules, more on this in Section 3.6. This declaration will be omitted in further example rules if it is not relevant to the example.
- `rule`: A keyword for access control rules.
- `page`: The type of resource being protected here, the resource types allowed in WebDSL access control are: `page`, `action`, `template`, and `function`.
- `editUser`: The name of the resource the rule will apply to.
- `(u:User)`: The arguments (if any) of the resource, the types of the arguments are used when matching a rule with a resource. This also specifies what variables can be used in the checks. When a wildcard (`""*`) is used here you match any number of arguments but lose the possibility of using arguments in the check.
- `u = principal`: The check that determines whether access to this resource is allowed, this check must be a correct WebDSL boolean expression or an error is produced when compiling the WebDSL application. Note that the members of the `securityContext` entity are directly visible in the context of access control rules. The use of a member of `securityContext` implies that the `securityContext` is not null and when using references to entities such as `principal` in checks, these are also verified to be not null (these null-checks are generated automatically).

Matching can be done a bit more freely using a trailing `""*` as wildcard character, both in resource name and arguments. The following example simply allows unconditional access to all pages with name `viewUser`, any argument configuration matches:

```
rule page viewUser(*) { true }
```

The next example opens up every page that starts with `view` and ignores the arguments. Note that the actual arguments to the pages, for example the `User` argument of `viewUser`, cannot be used anymore in the check this way:

```
rule page view*(*) { true }
```

When more fine-grained control is needed for rules, it is possible to specify nested rules. This implies that the nested rule is only valid for usage of that resource inside the parent resource. In WebDSL access control rules the allowed combinations for nesting are:

- page - action: Only the actions in the pages matched by the page rule are affected by this action rule.
- template - action: Only the actions in the templates matched by the template rule are affected by this action rule.
- page - template: Only the templates in the pages matched by the page rule are affected by this template rule.
- template - template: Only the templates in the templates matched by the parent template rule are affected by this child template rule.

The next example shows a nested rule for an action in a page:

```
rule page editDocument(d:Document) {
  d.author = principal
  rule action save() {
    d.author = principal
  }
}
```

This rule will only allow the author of the document access to the page and access to the save action on that page. The nested rules make it possible to have a more strict condition on certain actions on a page. This flexibility is often not necessary, and it is also inconvenient having to explicitly allow all the actions on the page, for these reasons some extra desugaring rules were added to the rule language. When specifying a check on a page or template without nested checks, an action `*(*)` rules block with the same check is added to it by default. For example:

```
rule page editDocument(d:Document) {
  d.author = principal
}
```

becomes

```
rule page editDocument(d:Document) {
  d.author = principal
  rules action *(*) {
    d.author = principal
  }
}
```

When multiple rules in an access control rules set apply to a certain resource, a conjunction of their checks becomes the enforced check. This way you can be sure that any condition you specify has to be fulfilled. Other combination strategies are possible, but to keep the language clear and easy to comprehend, this single strategy (all checks must permit access) has been chosen.

3.2.2 Rule Reuse

The rules representing the access control policy as discussed so far do not offer any reuse of checks or expressions, resulting in possibly a lot of replication in code. Two extra elements are added to support some reuse, predicates and pointcuts.

Predicates are functions consisting of one boolean expression, which allows reusing complicated expressions, or simply giving better structure to the policy implementation. An example of a predicate:

```
access control rules
  predicate mayViewDocument (u:User, d:Document){
    d.author = u
    || u in d.allowedUsers
  }
  rule page viewDocument(d:Document){
    mayViewDocument(principal,d)
  }
  rule page showDocument(d:Document){
    mayViewDocument(principal,d)
  }
```

The `mayViewDocument` predicate holds the expression used to check whether viewing access to the document is allowed by a certain user. This predicate can be used in multiple rules, for example a rule for the `viewDocument` page and one for the `showDocument` page. Because the predicate is contained in a `access control rules` block it also has direct access to the `securityContext`'s members.

Pointcuts are groups of resources to which conditions can be specified at once. Especially the open parts of the web application are easy to handle with pointcuts, for example:

```
pointcut openSections(){
  page home(),
  page createDocument(),
  page createUser(),
  page viewUser(*)
}
rule pointcut openSections() {
  true
}
```

The pointcut `openSections` lists the pages that should always be allowed access to. The check that opens all these pages is only specified once in the rule that works on the pointcut. This feature adds the pointcut as a possible resource to the rules syntax. It is also possible to use arguments in pointcuts, which is shown in the next example:

```
pointcut ownUserSections(u:User){
  page showUser(u),
  page viewUser(u),
  page someUserTask(u,*)
}
rule pointcut ownUserSections(u:User){
  u = principal
}
```

The arguments of the separate pointcut elements can be constructed here using the argument name of the actual pointcut, but all the arguments specified in the pointcut need to be linked with an argument of the pointcut element. If this is not the case, the check enforced might not be possible at all. Take for instance the following pointcut element:

```
page allUsers()
```

Adding this to the above pointcut would conflict with the check that needs a user argument from the page.

3.3 Data Structures for Access Control

An important part of access control models lies in the data needed to represent a model. While an access control system that is entirely separated from the application must deal with generic data structures and references to data in the application, WebDSL access control can directly use the application data structures. The entity sub-language of WebDSL can be used to interface with existing data, add new types or extend certain types. This creates a strong basis for supporting varying types of models and their policies in a convenient way.

3.3.1 Using Existing or New Entities

The entities of the application that is to be protected are available to use in the checks. When a rule matches on resources with arguments, those arguments can be used in the checks. An example of this is the following:

```
rule page editUser(u:User) {
  u = principal
}
```

This access control check knows the `User` entity from its specification in the application:

```
entity User {
  name :: String
  password :: Secret
}
```

This allows typechecking of the access control check during compilation. In this case, the `u = principal` check, the type of the `principal` needs to match with the type `User`.

Another type of entity available in WebDSL is the session entity. These entities are available in the entire interaction of one user and the application. This can be used to hold additional information for making access control decisions. Here is an example of a session entity that is generated from the declaration of the principal:

```
principal is User with credentials name, password
```

results in the following session entity being created:

```
session securityContext {
  principal -> User
}
```


3.3.2 Extending Entities

When adding new entities to the application it can be useful to link these with existing entities. You could add a reference to an existing entity type in the new entity. However, this will incur extra retrieval work when checking the attached properties and the separation between normal properties and attached properties needs to be explicitly handled in the application code. In order to make this easier it is possible to extend an entity with extra properties. This extension becomes a real part of the extended entity. This example shows the addition of a `Set<User>` property to the `Document` entity representing the users allowed access to the `Document`:

```
extend entity Document {
  allowedUsers -> Set<User>
}
```

By specifying the extra information as extensions, the usage of these properties is the same as the normal properties. This results in easier to understand access control rules.

3.4 Inferring Visibility

A disabled page or action redirects to a very simple page stating that access is denied. Since this is not a very user-friendly solution, it would be better if the link to the denied page or the button submitting the denied action was not available at all. It is possible to do this in WebDSL by putting a condition on template elements in a page definition:

```
if(mayViewDocument(securityContext.principal,d)){
  navigate(viewDocument(d)){ "view " output(d.title) }
}
```

The part inside the `if`-block will only be visible if the `mayViewDocument` call returns true. This solves the problem of the unwanted link, but it couples access control code with the page definition, something we try to avoid. In general it can be assumed that a page which is not accessible should not be available for navigation. Taking this into account WebDSL access control will automatically add these conditions to navigate links, and action calls (or actionlinks). Figure 3.3 shows how a menu is influenced by inferred visibility on navigate links, only the allowed pages (in this case the document authored by the logged in user) are visible in the menu.

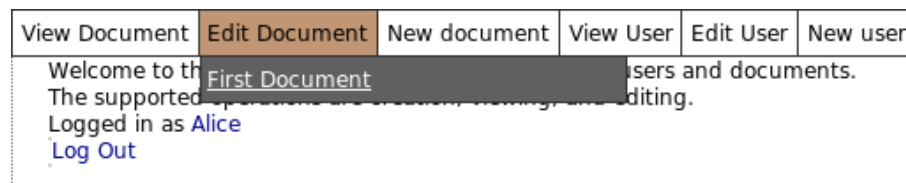


Figure 3.3: Inferring visibility for menu links

3.5 Administration

Administration of access control is often not well supported by existing solutions and involves a lot of manual encoding. In WebDSL access control all the data of the policy is integrated into the WebDSL application. This setup makes it easy to construct administration pages using WebDSL itself. These pages can of course be protected using the very same WebDSL access control definition under administration. Another option is to incorporate the administration into an existing page with a template. An example that illustrates this template solution is the administration of the `allowedUsers` set of a `Document` entity:

```
define allowedUsersRow(document:Document){
  row{ "Allowed Users:" input(document.allowedUsers) }
}
```

The template call for this template is added to the `editDocument` page:

```
table{
  row{ "Title:" input(document.title) }
  row{ "Text:" input(document.text) }
  row{ "Author:" input(document.author) }
  allowedUsersRow(document)
}
```

This template solution allows for the access control administration code to be placed in a separate place together with the rest of access control related code. When this part is (temporarily) removed the form will still be generated but without the access control input (note that undefined template definitions only give a warning in WebDSL). The result of this administration code can be seen in Figure 3.4.

Edit Document First Document

Title:	<input type="text" value="First Document"/>
Text:	<input type="text" value="This is the first text."/>
Author:	<input type="text" value="Alice"/>
Allowed Users:	<div> <input type="text" value="Alice"/> <input type="text" value="Bob"/> <input type="text" value="Charlie"/> <input type="text" value="Dave"/> </div>

Figure 3.4: Access control administration

3.6 Policy Combination

Rules are combined by conjunction in rule sets, this fixed combination strategy results in clear rule definitions that are easy to understand. However, in certain situations it is more

convenient to specify a disjunction of the rules, e.g., the situation where an administrator user has access to all pages. Specifying this in a rule set would involve adapting all the rules to allow access if the user is an administrator. To solve this problem, rule sets can be combined in a boolean expression. Names can be given to rule sets by specifying the name after access control rules. This gives the sets a handle to use in the boolean set combination expression. Unnamed rules (which we have seen so far) are added under the name anonymous. The following example illustrates the administrator user situation:

```
access control policy anonymous OR admin
access control rules admin
  rule page *(*){
    isSuper(principal)
  }
access control rules
  rule page viewDocument(d:Document) {
    d.author = principal
  }
```

Two rule sets are specified: one named admin, and the other is unnamed or anonymous. The access control policy anonymous OR admin specifies the combination expression for rules in the sets of admin and anonymous.

Chapter 4

WebDSL Access Control Syntax and Semantics

In the previous chapters we have seen that access control is defined separately from the rest of a WebDSL application. In order to enforce the rules specified in a policy, local checks are introduced into pages, templates, and actions. An implementation-independent description of the weaving transformations realizing this will be given in this chapter.

The resources of a WebDSL application that can be protected are pages, templates, and actions. Furthermore, it is possible to restrict access more specifically to resources relative to other resources, e.g., actions within pages and actions within templates, or to resources that use another resource such as actions that use functions. The semantics of protecting a resource differs for these resources and the combinations, which indicates the need for the semantics description given in this section.

This chapter describes the syntax and semantics of the WebDSL access control sub-language. The descriptions shown here will not be in the languages used in the actual implementation (which are SDF [23] for syntax and Stratego [25] for transformation), instead the mechanism will be presented in a more general form.

Section 4.1 describes the syntax of WebDSL access control. Section 4.2 describes the transformations used for the principal declaration. Section 4.3 describes the semantics of the language constructs with an example transformations followed by a generic description of the transformation. Section 4.4 describes the transformations used for predicates and pointcuts. Section 4.5 describes the visibility inference on certain page elements. Section 4.6 describes the transformations on rule sets used in policy combination expressions. Section 4.7 describes various validations performed on the access control rules by the WebDSL compiler.

4.1 Syntax

This section discusses the syntax for WebDSL access control. The WebDSL access control syntax is shown in Figure 4.1. Parts of the WebDSL syntax are reused, indicated by

a preceding `w-`, to show where access control is inserted in normal WebDSL application syntax.

The syntax basically consists of four different declarations which are the principal, rule, predicate, and pointcut declarations. Together these can form a WebDSL definition, this is where WebDSL access control is integrated into WebDSL, at the same level of sections. Resources are described using generic ids, which are the ids from the WebDSL syntax. Verifying that the id is a valid one will be done by the transformation engine and is described in Section 4.7. Correct possibilities for nested resource checks are also verified during transformation.

```
w-definition -> "access control rules" w-id? ac-definition*

ac-definition -> ac-principal | ac-rule | predicate | ac-pointcut

ac-principal -> "principal is" w-id "with credentials" {w-id ","}*

match-args -> w-farg | "*" | {w-farg ","}+ ","*?

ac-rule -> "rule" w-id w-id "***? "(" match-args ")" "{" w-expr ac-rule* "}"

predicate -> "predicate" w-id "(" {w-farg ","}* ")" "{" w-expr "}"

ac-pointcut -> "pointcut" w-id "(" {w-farg ","}* ")" "{" {pointcut-elem ","}+ "}"

pointcut-elem -> w-id w-id "***? "(" {w-id | w-id | {w-id ","}+ ("," "****)? ")"
```

Figure 4.1: Syntax

4.2 Principal Declaration

The declaration of the principal is required for WebDSL access control. It is assumed a user is always represented by an entity in the system. This entity becomes the type referenced by the `principal` property in the `securityContext`. The transformation applied is the following:

```
principal is User with credentials name, password

⇓

session securityContext {
    principal -> User
}

access control rules
    predicate loggedIn() { principal != null }
function authenticate(name : String, pw : Secret) : Bool {
    var users : List<User> := select u from User as u where (u.username = ~name);
    if ( users.length = 1 && users[0].password.check(pw) )
    { securityContext.principal := users[0]; return true; }
    else { return false }
}
```

From the principal declaration the entity type is first collected, which is then used to construct the `securityContext`. A `loggedIn` predicate is added for a more readable login check. The `authenticate` function is generated from the entity name and the specified credentials.

4.3 Rule Weaving

WebDSL access control is aspect-oriented in nature. The implementation is done by weaving checks into an existing WebDSL application, implemented in the transformation language Stratego. The transformations will be discussed using typical examples together with a description of the intended effect and possible variations. This way of describing semantics is intuitive and abstracts from implementation artifacts. Also no knowledge of Stratego is needed to understand what the implementation is actually doing.

4.3.1 Page Weaving

The primary resource being protected in WebDSL is the page. Upon denying access to a page, the user is redirected to a simple page stating that access is denied. This redirection will be avoided most of the time by disabling links to denied pages (see the visibility inference implementation in Section 4.5), however someone might still try to access the page directly by typing it in the address bar of the browser. The transformation that enforces the rule protecting a page, is that of adding an initialize action whichs redirects if the condition is not satisfied. The basic `viewDocument` page is taken as an example here; the `init` block added will make sure the page will not be shown when access is not allowed:

<pre> define page viewDocument(document:Document){ main() define body() { ...rest of viewDocument page... } } access control rules rule page viewDocument(doc:Document) { doc.author = principal } </pre>	\Rightarrow	<pre> define page viewDocument(document:Document) { init { if(!(document.author = securityContext.principal)){ goto(accessDenied()); } } main() define body() { ...rest of viewDocument page... } } </pre>
---	---------------	--

A page can have initialization actions which should also be restricted if access is denied. These `init` actions are taken into account and made conditional. An example presented here is a page that starts by emptying some properties as an initialization action. The other `init` blocks are gathered and the statements are made conditional using the access control check:

The order of evaluation of the `init` blocks is not specified in WebDSL, so they can be safely added together inside the `if` block if more are present. The check from the rule is substituted using the argument identifiers of the page. The `goto` statement creates a redirection in WebDSL, preventing access to the current page.

```

define page overwriteDocument(d:Document) {
  init {
    d.title := null;
    d.text := null;
  }
  ...rest of overwriteDocument page...
}
access control rules
rules page overwriteDocument(doc:Document) {
  doc.author = principal
}

```

⇒

```

define page overwriteDocument(d:Document) {
  init {
    if(d.author = securityContext.principal){
      d.title := null;
      d.text := null;
    }
    else {
      goto(accessDenied());
    } } }

```

As discussed in the previous chapter, WebDSL access control will deny access to pages which are not protected by any rules. If the rule in the above example was not used the resulting WebDSL page would be:

```

define page clearDocument(d:Document) {
  init {
    if(false) {
      d.title := null;
      d.text := null;
    }
    else {
      goto(accessDenied());
    } } }

```

The resulting page is shown in Figure 4.2. As an optimization the denied page could be removed entirely from the generated application.

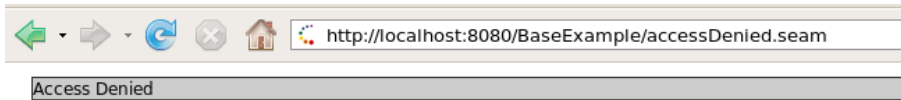


Figure 4.2: Access denied

The general form of the page transformation is shown in Figure 4.3. In these generic transformation descriptions \vec{x} is used to denote a list formal parameters $x1 : S1, \dots, xn : Sn$. The rule states that an access control rule with signature page $p(\vec{x})$ inserts an `init` block in the page definition with the signature page $p(\vec{x})$ containing a redirect to the `accessDenied` page in case the condition of the rule evaluates to false, while accounting for existing `init` blocks as well.

4.3.2 Action Weaving

The second resource that needs to be protected in WebDSL is the action. It must be ensured that any action is directly protected, because even if a form is not visible for a user, that user could forge a request in the browser invoking the form and the corresponding action. This direct protection is instantiated by a transformation of the action code. The following example shows how a rule protecting an action is enforced by the transformation:

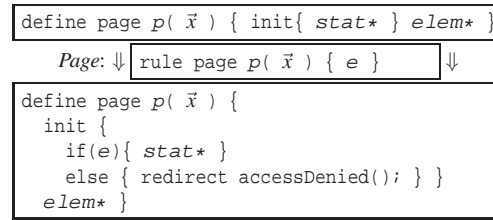


Figure 4.3: Page rule transformation

```

define page editDocument(document:Document){
  ...form calling save action...
  action save(){
    document.save();
    return viewDocument(document);
  }
}

access control rules
rule page editDocument(doc:Document) {
  doc.author = principal
  rule action save() {
    doc.author = principal
  }
}

```

⇒

```

define page editDocument(document:Document){
  ...form calling save action...
  action save(){
    if(document.author =
      securityContext.principal){
      document.save();
      return viewDocument(document);
    }
    else{
      return accessDenied();
    } }
}

```

The rule for the action is the same as for the page (so it could be left out as well for the same result, action rules with identical checks are added automatically in page rules), the doc identifier is known in the context of the nested rules, which makes it possible for the `save()` check to not have any arguments but still use `doc`.

The example provides a minimal setup to show the weaving for actions. There are two types of actions in WebDSL, actions that return a page reference (redirecting upon completion) and actions that stay on the same page after executing. The first type will be transformed to a check that returns the `accessDenied` page, the second type will redirect to the `accessDenied` page using a `goto` statement. The previous example would change to the following if no return was specified in that action:

```

define page editDocument(document:Document){
  action save(){
    if(document.author = securityContext.principal){
      document.save();
    }
    else{
      goto(accessDenied());
    } }
}

```

In Figure 4.4 the generic transformation is shown for the actions that do not return to another page (thus showing a redirect and not a return in the else-clause).

4.3.3 Template Weaving

A WebDSL program offers generic reuse of page components with templates. The template definitions can consist of arbitrary page components, and can have arguments supplied that

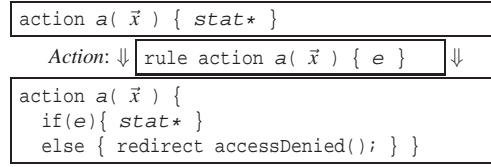


Figure 4.4: Action rule transformation

are to be substituted in the template. A call to a defined template takes into account its environment which allows overriding of template parts at the calling site. This mechanism of the environment affecting reusable parts is called dynamic scope. The T_EX typesetting system served as an example for the WebDSL templates.

Templates can be used to group related aspects of a page. A good example of this are the login and logout templates described in Chapter 3. The visibility of these templates and the access to their actions can be controlled through WebDSL access control by specifying rules on templates. A simple example that hides the login template when a user is already logged in and hides the logout template when a user is not logged in is shown here:

```

rule template login(){ !loggedIn() }
rule template logout(){ loggedIn() }
  
```

The rules control the visibility of all the elements in these templates:

```

define page home() {
  login()
  logout()
}
⇒
define page home(){
  if(!securityContext.loggedIn){
    login()
  }
  if(securityContext.loggedIn){
    logout()
  }
}
  
```

Since rules for nested actions are also automatically added to templates, the actual transformation involves applying the checks to the action inside the template definitions as well. This transformation will be illustrated in Section 4.3.5.

Figure 4.5 shows the generic transformation being applied for template rules.

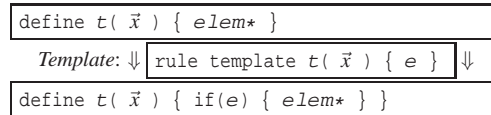


Figure 4.5: Template rule transformation

4.3.4 Function Weaving

Functions are available in WebDSL and allow reuse of action code. Since the access control rules are protecting certain operations inside actions, it makes sense to protect functions as

well. Putting checks in functions directly produces problems with partly executed actions and illegal return values. However, it doesn't make sense allowing an action that uses a function which it has no access to, also the access control checks might invoke some of the rules themselves (which is already the case for predicates); therefore checks for functions are lifted to the action calling the function, to be able to handle a denial of access cleanly. The example shown here displays where the function check is placed in the case of a call in an action:

```
rules function storeChanges(document:Document) {
  document.author = principal
}
```

The `storeChanges` function is protected, only the author is allowed to execute that function on a particular document. The transformation applied:

```
action saveDoc(d:Document){
  someUnprotectedOperation(d);
  storeChanges(d);
}

⇒

action saveDoc(d:Document){
  if(d.author = securityContext.principal){
    someUnprotectedOperation(d);
    storeChanges(d);
  }
  else{
    goto(accessDenied());
  }
}
```

As you can see here, the transformation applied is similar to that of a normal action rule. The possibility of specifying this rule on the function allows for similar reuse in the WebDSL access control declarations. It is important to note here that this particular transformation is not valid in all cases since the arguments to the function might not be the same or not even exist at the beginning of the action. However, this naive way of lifting the check works in most of the practical cases, therefore it is allowed in the access control sub-language. Some checks are added to verify that the arguments to the function are untouched between the lifted check and the function invocation, and give an error if it is not the case.

Figure 4.6 shows the generic transformation being applied for function rules. In this rule the actual arguments \vec{z} match the signature of the formal arguments \vec{y} . e' is the expression e with the formal arguments \vec{y} substituted by the actual arguments to the function, \vec{z} .

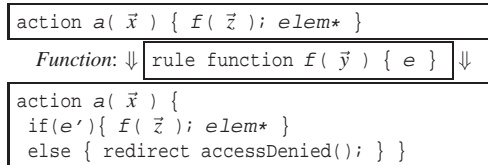


Figure 4.6: Function rule transformation

4.3.5 Nested rules

To support more fine-grained access control it is possible to specify nested rules. These nested checks inherit the checks from the parent rule, and they can access the arguments

of the parent rule. It becomes possible to have a page form with multiple actions that are each protected by different rules. The buttons or links for these actions will be hidden and disabled as specified by the rules (more on visibility inference in Section 4.5).

Typically nested rules are created to protect actions in certain pages or templates. In previous examples nested rules for actions were already shown, here the generic transformation will be described for these typical cases. Figure 4.7 shows the generic transformation for an action rule nested in a page rule. The action in that page is protected by a conjunction of the page rule and the specific action rule.

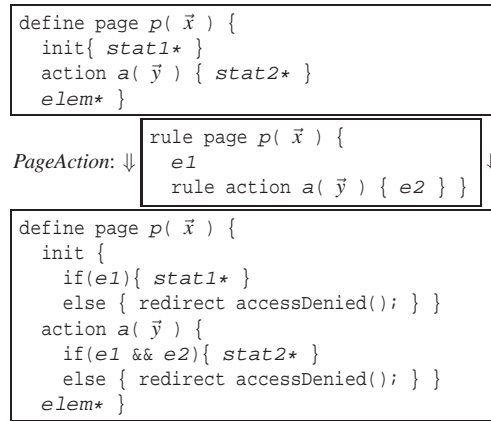


Figure 4.7: Page-action rule transformation

Figure 4.8 shows the generic transformation for an action rule nested in a template rule. The action in that template is protected by a conjunction of the template rule and the specific action rule. This transformation is specified in the defined template before it is expanded, which means the access control checks are placed in all the instantiations of the template.

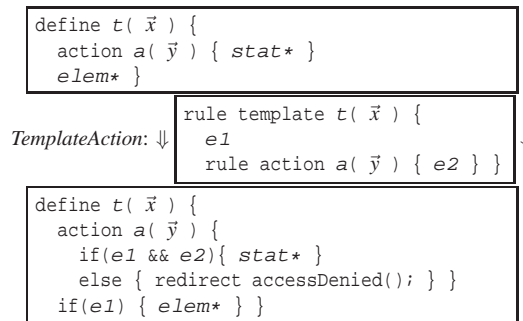


Figure 4.8: Template-action rule transformation

4.4 Reuse Desugaring

The pointcuts and predicates allow reuse of access control definitions. These elements are transformed into other WebDSL access control definitions and do not transform the

WebDSL application directly. The pointcut produces a single rule for each element in the pointcut:

<pre>pointcut userSection(usr:User){ page viewUser(usr), page editUser(usr) } rule pointcut userSection(usr:User){ usr = principal }</pre>	\Rightarrow	<pre>rule page viewUser(usr:User){ usr = principal } rule page editUser(usr:User){ usr = principal }</pre>
--	---------------	--

Figure 4.9 shows the general transformation being applied for pointcuts. In the pointcut transformation the \vec{y} are a form of actual arguments using the identifiers in \vec{x} , which allows different ordering and possible wildcards.

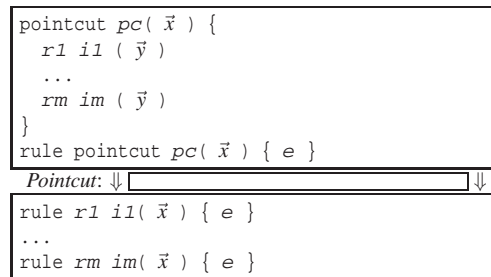


Figure 4.9: Pointcut transformation

Predicates are transformed to functions with a `Bool` return value. The expression determining the value of the predicate is simply returned in that function:

<pre>predicate mayViewDocument (u:User, d:Document){ d.author = u u in d.allowedUsers }</pre>	\Rightarrow	<pre>function mayViewDocument (u:User, d:Document):Bool{ return d.author = u u in d.allowedUsers; }</pre>
--	---------------	--

The generic transformation for predicates is shown in Figure 4.10.

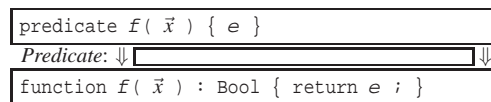


Figure 4.10: Predicate transformation

By reusing the existing language features these additions can be implemented with great ease and inherit all the supporting code. This supporting code primarily involves the validation of the declared elements. The predicate is validated before transforming it into a function, in order to get good error reporting (error is linked to the declared predicate and not to the generated function).

4.5 Inferred Access Control

The `accessDenied` page is useful to clearly state the access denied status of a request, but the navigation to denied pages and the use of denied actions should be avoided as much as possible for a user-friendly experience when using a WebDSL application. Disabling elements manually in the views corresponding to the access control policy is time consuming and error prone. Templates offer more configurable views but using them for all visibility variations becomes messy. For these reasons WebDSL access control will hide elements automatically if they don't make sense under the given policy. This consists of hiding navigate links and buttons or links to denied actions. An example of hiding navigate links is presented first. The rule used in the example prevents any user other than the owner to access the `editDocument` page. The transformation applied to the navigation link is the following:

<pre>define page editlink(doc:Document){ navigate(editDocument(doc)) { "edit" output(doc.title) } } access control rules rule page editDocument(d:Document){ d.author = principal }</pre>	\Rightarrow	<pre>define page editlink(doc:Document){ if(doc.author = securityContext.principal){ navigate(editDocument(doc)) { "edit" output(doc.title) } } }</pre>
---	---------------	---

The `if`-block will prevent the navigate link from showing up when the condition is not satisfied and thus save unnecessary redirections to the `accessDenied` page. The generic transformation is shown in Figure 4.11, the e' means e with formal arguments \vec{x} substituted by actual arguments \vec{y} .

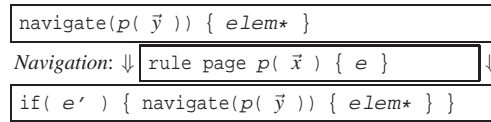


Figure 4.11: Conditional navigation

The next example shows what happens to an action button or action link when the action being called is not allowed to be executed. Given the similarity with page navigation restriction, only an example is shown:

<pre>define page editDocument(document:Document){ form { table { row{ "Title:" input(document.title) } row{ "Text:" input(document.text) } row{ "Author:" input(document.author) }} action("Save", save()) ...declaration of the save action... }} access control rules rule page editDocument(d:Document) { true rule action save(){d.author = principal}}</pre>	\Rightarrow	<pre>define page editDocument(document:Document){ form{ table{ row{ "Title:" input(document.title) } row{ "Text:" input(document.text) } row{ "Author:" input(document.author) }} if(document.author = securityContext.principal){ action("Save", save()) } ...declaration of the save action...}}</pre>
---	---------------	--

The visibility inference performed here could be expanded to handle other elements as well. These type of transformations provide a fast way to create a consistent application; they're made possible by the explicit notion of access control in a WebDSL application definition.

4.6 Policy Combination

The transformations performed to convert rule sets and the overall policy definition to a single set of non-overlapping rules are shown in this section. Two rules are said to *match* if they have the same signature, i.e., resource type, name, and parameters are the same. If a rule set contains two matching rules, they can be merged into a single rule with the conjunctions of the two expressions as expression, see Figure 4.12.

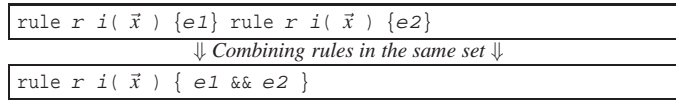


Figure 4.12: Rule normalization

The OR operator applied to a pair of matching rules turns into a single rule with the disjunction of the expressions. The OR operator applied to two sets of rules produces the pairwise disjunction of matching rules. That is, assuming that the argument sets are normalized and thus contain for each resource at least one rule, if a matching rule exists in each set they are combined with the OR operator. Rules for which no matching counterpart exists are taken as is. The OR operations are shown in Figure 4.13.

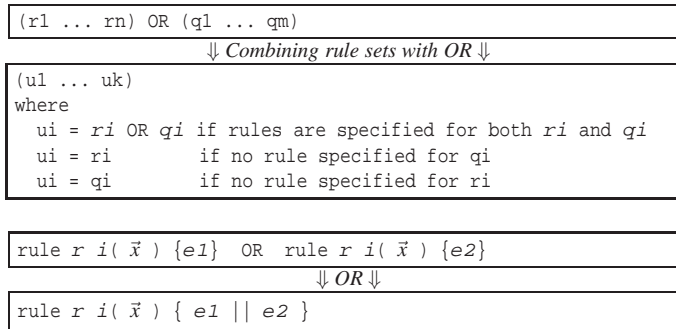


Figure 4.13: Combining rule sets with OR

The AND operator applied to a pair of matching rules turns into a single rule with the conjunction of the expressions. The AND operator applied to two sets of rules produces the pairwise conjunction of matching rules, similar to the OR operator. The AND operations are shown in Figure 4.14.

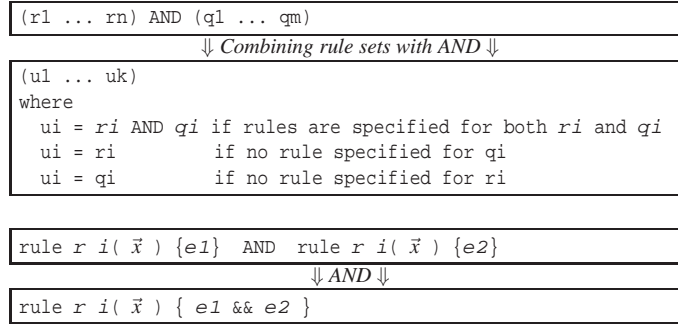


Figure 4.14: Combining rule sets with AND

4.7 Program Validation

Besides errors related to incorrect syntax (produced by the generated parser) WebDSL access control gives errors when incorrect semantic constructions are declared. The type of feedback given by the WebDSL compiler is shown in this section.

Pointcuts are transformed in separate rules for each pointcut element. Valid pointcuts have their arguments correctly represented in each element. If an argument is missing in an element, it will result in an error:

<pre> pointcut userSection(usr:User){ page viewUser(), page editUser(usr) } </pre>	\Rightarrow	<pre> pointcut element page viewUser() does not have a usr argument specified </pre>
--	---------------	--

Rules matching on undefined pointcuts are also not allowed:

<pre> pointcut userSection(usr:User){ page viewUser(usr), page editUser(usr) } rule pointcut userSectio(usr:User){ usr = principal } </pre>	\Rightarrow	<pre> pointcut userSectio does not exist </pre>
---	---------------	---

This example shows the result of incorrect argument matching in a pointcut rule:

<pre> pointcut userSection(usr:User){ page viewUser(usr), page editUser(usr) } rule pointcut userSection(usr:User){ usr = principal } </pre>	\Rightarrow	<pre> argument usr:Usr not in pointcut userSection </pre>
--	---------------	---

After pointcuts are desugared to normal rules the principal declaration is checked. For this declaration a valid entity needs to be referenced:

<pre> principal is Usr with credentials name, password </pre>	\Rightarrow	<pre> Usr is not a declared type </pre>
---	---------------	---

Also the properties used for login template generation need to be valid:


```
principal is User                                ⇒    nam is not a property of User
  with credentials nam, password
```

Predicates are transformed into functions, which is a pretty direct transformation. If an error is made in the check it is important that this is linked with the predicate:

```
predicate mayViewDocument(                      ⇒    d.author = principl check in predicate
  u:User, d:Document){                          mayViewDocument(u:User, d:Document)
  d.author = principl                            failed typechecking
}
```

Because predicates are transformed into functions with a Bool return type, the expression can be verified to have the Bool type:

```
predicate mayViewDocument(                      ⇒    principal check in predicate
  u:User, d:Document){                          mayViewDocument(u:User, d:Document)
  principal                                      is not a boolean expression
}
```

The actual access control rules are checked next. The type of resource needs to be verified:

```
rule pae editDocument(d:Document){              ⇒    pae is not a valid security check type
  d.author = principal
}
```

The expression is checked to be a correct WebDSL expression:

```
rule page editDocument(d:Document){             ⇒    d.aotor = securityContext.principal
  d.author = principal                           failed typechecking
}
```

Also the resulting Bool type of the expression is verified:

```
rule page editDocument(d:Document){             ⇒    d.author is not a boolean expression
  d.author
}
```

Because unused rules are likely to be an error in the declaration of the access control rules, this situation is caught and reported:

```
rule page editDcument(d:Document){             ⇒    warning: in access control rules: rule not used:
  d.author = principal                           rule page editDcument(d:Document){
}                                                  d.author = principal
}                                                  }
```


Chapter 5

Policy Implementations

The access control sub-language of WebDSL provides high-level, policy neutral mechanisms for defining access control, that is, without making assumptions about the type of policy to be enforced. The flexibility of the mechanisms allows the adaptations of standard policies typically needed in practical settings, and enables the combination of elements from different policy models. In this section, the three major access control paradigms and their encoding in WebDSL access control are discussed.

The example used in chapters 2 and 3 will be the basis for the policy examples here. It will be extended to show the relevant properties of the policy types. The discussion of the examples in this chapter is structured as follows:

- Information on the type of policy.
- Data structures and extensions of the example.
- Predicates used for checking authorization.
- Rules for the example's resources.
- Administration facilities.

Section 5.1 discusses Mandatory Access Control (MAC) policies and their implementation in WebDSL, Section 5.2 discusses Discretionary Access Control (DAC), and Section 5.3 discusses Role-Based Access Control (RBAC).

5.1 Mandatory Access Control

Mandatory Access Control (MAC) [16, 17, 3, 21] models are based on assigning labels (e.g. TopSecret, Secret, Confidential, Unclassified) to subjects and objects for determining access permissions. Subjects have a clearance label that indicates what type of resources the subject can access. Objects have a classification label which represents their level of protection. The relative importance of labels is determined by a partial order on labels. In MAC policies the distinction between the user (human interacting with the system) and the subject (process working on behalf of the user) is important, because a user

can create a subject at any clearance label dominated by theirs. Domination of a clearance label means the label itself and all below it in the hierarchy. For instance, in Figure 5.1, a user with clearance level (CL) 2 would be able to access the system at both CL2 and CL1.



Figure 5.1: Dominance

MAC policies are mainly aimed at preserving confidentiality of objects (Section 5.1.1), and to a lesser extent preserving the integrity (Section 5.1.2). These properties deal with information flow, by preventing unsafe transfer of (the contents of) objects to other security levels in the system. The classifications in MAC models and the flow of information can be described using lattices (Section 5.1.3).

5.1.1 Confidentiality

The main property of MAC models is that of preserving confidentiality or secrecy of objects. To protect confidentiality two properties must hold:

- simple security rule: also known as the read-down property, which states that a subject needs to have a security clearance higher than or equal to the security classification of an object to be able to read it.
- *-property: also known as the write-up property, which states that a subject needs to have a security clearance lower than or equal to the security classification of an object to be able to write it. This is needed to prevent leaking confidential information to lower clearances. A stricter form can be used to prevent low subjects to overwrite high data (which is possible with the liberal *-property). This form only allows writing where the clearance matches the classification, and is known as the strict *-property.

A read in this model is a view operation, a write in this model is a write once or create operation, without the possibility of editing this data afterwards. The MAC example in this chapter will use create and view as actions to be protected. In Figure 5.2 the permissions for viewing and creating of data for 3 levels of classification is shown. The arrows show the read-down and write-up properties. A subject with security clearance level 2 can create

documents of level 2 or 3 classification, and can view level 1 or 2 classified documents. The information flows upwards in this example.

Subjects									Objects
clearance	create	view	clearance	create	view	clearance	create	view	
CL: 3	↑	↓	CL: 2	↑	↓	CL: 1	↑	↓	Documents with classification: 3 Documents with classification: 2 Documents with classification: 1

Figure 5.2: Confidentiality information flow

The confidentiality of the documents is protected from leaking to lower levels in the system, because a subject cannot create a document at a lower level than the objects being read. This doesn't mean a user cannot give information to other users with a lower security clearance, but it forces them to access the system at a lower clearance level and thus limiting several risks. One of these risks is malicious code that is executed during the user's session without the user having knowledge of it (better known as a Trojan Horse). This code won't be able to cross the boundaries of the user's subject and will be prevented from creating documents at lower classifications to steal protected information. In the domain of web applications a Trojan Horse could be a cross-site request forgery (CSRF). CSRF attacks consist of a page request being made without the user having knowledge of that request. Since most browsers will send cookie data (often containing authentication or session information) along automatically, a request can use the permissions of the user. If a MAC policy for confidentiality is used, the secret request is limited to creating something in the same or higher classifications only. The creator of this secret request doesn't have access to these higher levels, and thus can't steal information this way.

5.1.2 Integrity

Besides the confidentiality protection, there is also a MAC model where the intent is to protect integrity. Integrity is a broad property, the MAC model only deals with integrity with respect to information flow. The properties for enforcing integrity are:

- simple-integrity property: also known as the read-up property, which states that a subject needs to have a security clearance lower than or equal to the security classification of an object to be able to read it.
- Integrity *-property: also known as the write-down property, which states that a subject needs to have a security clearance greater than or equal to the security classification of an object to be able to write it. This property makes sure the integrity of the higher classified objects cannot be damaged by lower clearance subjects.

The direction of read and write are swapped compared to the confidentiality protection. In order to protect both confidentiality and integrity on the same objects levels are required to

be equal for both read and write. In Figure 5.3 the permissions are shown for the situation of 3 security levels, missions as objects, create as write action, and view as read action. A subject with clearance level 1 can create missions of level 1 classification, and can view missions of classification level 1, 2, or 3.

Subjects									Objects		
clearance	create	view	clearance	create	view	clearance	create	view			
CL: 3	↓	↑	CL: 2	↓	↑	CL: 1	↓	↑	Missions with classification: 3		
	↓			↓					Missions with classification: 2		
	↓								Missions with classification: 1		

Figure 5.3: Integrity information flow

5.1.3 Lattice-based models

MAC models enforce one-directional information flow in a lattice of security labels. For this reason they are also known as lattice-based access control models. Global security levels are not always suited to describe the information flows of an application. To get a more flexible model, categories are introduced which define the area of competence of both users and objects. An example lattice for 2 levels of security and 2 categories is shown in Figure 5.4. A user holding clearance level 3 for category cat1 can activate subjects for CL3 [cat1], CL2 [cat1], CL3 [], CL2 [] (following the lattice down the hierarchy).

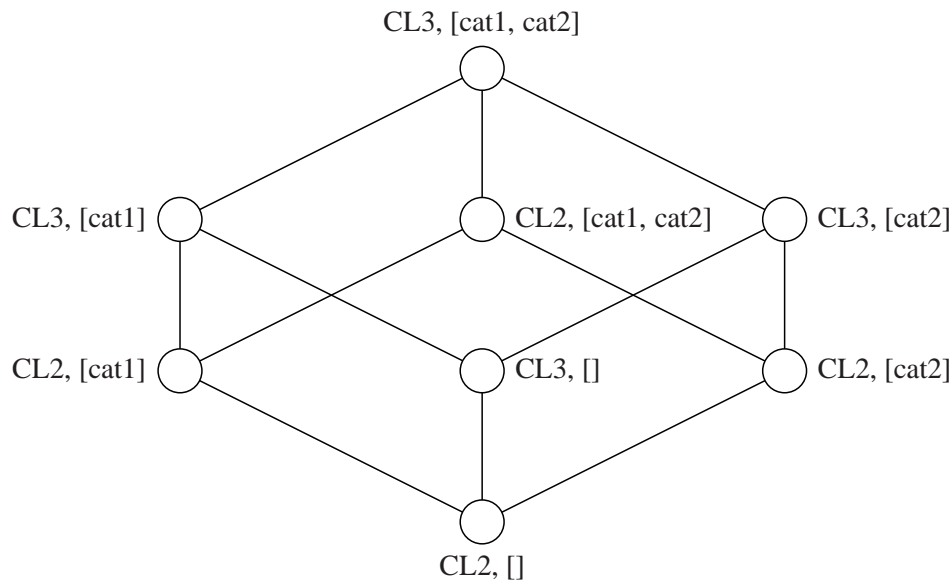


Figure 5.4: Security lattice

5.1.4 Limitations

Mandatory Access Control models only cover viewing (reading) and creation (writing) of data, which make them not a very complete solution for application policies. Nevertheless for parts of an application that rely heavily on confidentiality a MAC policy can be used to protect the information flow.

Adhering to the write-up policy for confidentiality has some problems, for instance when creating data at a higher level without being able to view the data at that level, which might result in a collision of similar identified documents. This is a difficult problem to handle, one option would be to destroy the existing data. That would mean the information in the system is corrupted. Another option would be to disallow this operation, but this would give information of the existence of a certain document at a higher security level (also known as a covert channel of information). Any information of a higher classification document should not be available to preserve the confidentiality. A solution that would work is changing write-up to write only at equal level (the strict *-property), but this would also make the system more rigid (users are forced to enable the clearance level exactly matching the object they write).

5.1.5 MAC Example

Example policy The Document entity will be used to demonstrate confidentiality protection. It will only allow viewing of Document entities when the security clearance of the User is higher or equal than the object classification of the Document (read down), and allow creating only when the clearance is lower or equal than the classification (write up).

Policy data The first thing needed for implementing the MAC policy is a place to store classifications and clearances. Since these properties belong to the Document and User entities, these entities are extended with them:

```
extend entity Document {
  classification :: Int
}
extend entity User {
  clearance :: Int
}
```

Because a user must be able to access the system at its own clearance level or a lower level, the current level activated is stored with the securityContext:

```
extend session securityContext {
  clearance :: Int
}
```

Predicates The expressions that check whether MAC is enforced have been put into predicates for a clean implementation without replication:

```
access control rules
predicate mayViewDocument(d:Document) {
  clearance >= d.classification
}
```

```

predicate mayCreateDocument(d:Document) {
    clearance <= d.classification
}

```

The use of `securityContext` properties requires the user to be logged in. A `loggedIn()` check is automatically added by WebDSL access control.

Rules With the predicates declared, we can now protect the pages, in this case the `viewDocument` and `createDocument` pages:

```

access control rules
rule page viewDocument(d:Document) {
    mayViewDocument(d)
}
rule page createDocument(d:Document) {
    mayCreateDocument(d)
}

```

This creates the basic MAC policy implementation, the result is shown for clearance level 1 in Figure 5.5 and for clearance level 2 in Figure 5.6.

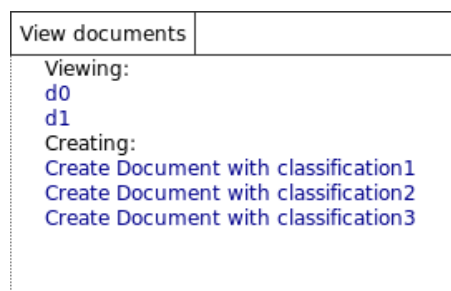


Figure 5.5: MAC clearance level 1

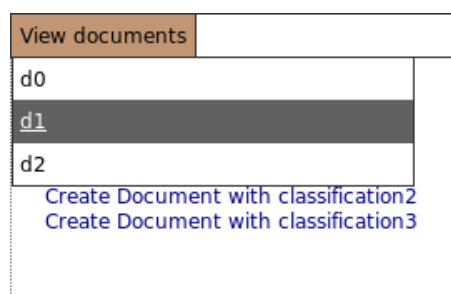


Figure 5.6: MAC clearance level 2 with menu

Administration Administration is not included explicitly in the MAC model, the model assumes a single administrator sets the fixed clearance level associations beforehand. In the example shown here, the clearances are also fixed, for example:


```
var charlie : User := User {  
  name := "Charlie"  
  clearance := 2  
};
```

A bit more interesting aspect of this policy's administration is the enabling of clearance levels that are below the user's clearance. Users need this to be able to write to lower levels. The action to set the current clearance is controlled from the sidebar:

```
for(cl:Int in [0,1,2,3]){  
  listitem{ actionLink("Activate CL: " + output(cl), activateCL(cl)) }  
}
```

The action being called sets the current clearance:

```
action activateCL(i:Int){  
  securityContext.clearance := i;  
  return home();  
}
```

Finally the activation of the clearance is controlled by an access control rule:

```
access control rules  
rule template sidebar(){  
  true  
  rule action activateCL(clear:Int){  
    principal.clearance >= clear  
  }  
}
```

This resulting sidebar elements are shown in Figure 5.7.



Figure 5.7: MAC sidebar clearance level 2

5.2 Discretionary Access Control

Discretionary Access Control (DAC) models are based on listing permissions for users and objects [16, 17, 20]. The user's identity and authorizations determine the permissions granted for each object. DAC policies are usually closed policies, only specifying the granting authorizations and denying by default. DAC policies often use the concept of ownership to determine permissions, the user that creates an object becomes the owner and has all the permissions for it. The owner can allow other users access to its owned objects (this decision is at the owner's discretion). This also puts the administration tasks in the hands of the owner. These tasks include granting other users access to the object, revoking that access, allowing others to help with administration (delegation), or simply deleting the object. Policies vary greatly in administrative capabilities available, some of the variation possibilities are: allowing ownership transfer; the granting of administration tasks to others can be limited (for example, only one person can get these permissions besides the owner); the revocation of the permissions granted can be linked to the user that specified the permissions; when a user loses administrative permissions, all the permissions specified by that user could be revoked as well (cascade).

Section 5.2.1 discusses the Access-Matrix Model and its relation with DAC policies. Section 5.2.2 discusses delegation of rights in DAC. Section 5.2.3 discusses some of the limitations of DAC. Section 5.2.4 discusses how to encode DAC in a WebDSL access control definition.

5.2.1 Access-Matrix Model

DAC policies are often described using the Access-Matrix Model [16, 17, 3, 20], a generic model for describing access control policies. It is based on the idea that all resources controlled by a computer system can be represented as objects. By listing all the permissions for these objects, the entire access control system can be described. To use the access matrix model, three things need to be identified:

- Objects: the resources that need to be protected.
- Subjects: the users or user's processes that want to access the objects.
- Permissions: the operations that are possible for an object, and which need to be protected in the system.

These concepts are used to describe a policy matrix, with subjects as indices for rows and objects as indices for columns. The set of permissions, i.e. the operations a subject s may apply to object o is listed in the matrix at $[s,o]$. A subject can also be an object if there are permissions that apply to them, in that case the subject will be represented in both a row and a column. Figure 5.8 shows an example of an Access-Matrix. In this matrix you see two types of objects, with different actions belonging to them. Ownership can be represented with the permission *Own*, but this could also be specified by listing all the actions an owner can do on its property.

	Document 1	Document 2	Document 3	Document 4	Mission 1
Alice	Own View Edit		Own View Edit	View Edit	Read
Bob	View	View	View	Own View Edit	Read Start
Charlie		Own View Edit		View Edit	Read

Figure 5.8: Access-Matrix

Since the Access-Matrix is usually sparse, it is rarely stored as an actual matrix in a system. Instead it is represented as:

- Access Control Lists (ACL): each object holds a list of the subjects and their permissions for the object. In the Access-Matrix a column represents the Access Control List of an object. In Figure 5.9 the ACL for Document 1 of the Access-Matrix in 5.8 is shown. It only needs two elements because one user has no permissions for Document 1. It is efficient to list all permissions for an object in an ACL, but expensive to list all the permissions a subject has, because all the ACLs need to be traversed to determine this. A well-known example of ACL for Access Control are the file permissions in a Unix file system.

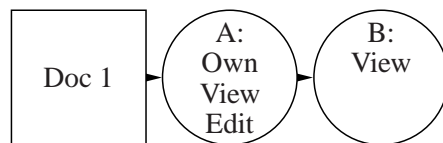


Figure 5.9: Access Control List

- Capabilities: each subject holds a list of the objects and the permissions for those objects. In the Access-Matrix a row represents the capabilities of a subject. In Figure 5.10 the Capabilities for subject Alice of the Access-Matrix in 5.8 is shown. There are no permissions for Document 2, so it is not stored. Here the list of all subject's permissions is efficient to retrieve, but retrieving the permissions per object is expensive.

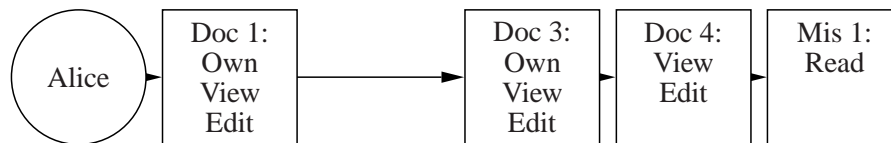


Figure 5.10: List of capabilities

- Triples of subject, object, and permissions: this occurs when a relational database is used to store the matrix in an authorization table. These tables do not have a preference to a certain type of review (showing lists of all permissions per subject or per object). In Figure 5.11 the authorization table entries for subject Alice of the Access-Matrix in 5.8 is shown.

Subject	Permission	Object
Alice	Own	Document 1
Alice	View	Document 1
Alice	Edit	Document 1
Alice	Own	Document 3
Alice	View	Document 3
Alice	Edit	Document 3
Alice	View	Document 4
Alice	Edit	Document 4
Alice	Read	Mission 1

Figure 5.11: Authorization table

5.2.2 DAC delegation

Most of the DAC policies use the concept of ownership to model the permission sets. The creator of an object usually becomes the owner. This also puts the administration tasks in the hands of the owner. These tasks include granting other users access to the object, revoking that access, allowing others to help with administration (delegation), or simply deleting the object. Policies vary greatly in how much administration is available or needs to be done by the owner, some of the variation possibilities are:

- Transfer of ownership is sometimes possible.
- The granting of administration tasks to others can be limited (for example, only one person can get these permissions besides the owner).
- The revocation of the permissions granted can be linked to the user that specified the permissions.
- When a user loses administration permissions, all the permissions specified by that user could be revoked as well (cascade).

5.2.3 DAC limitations

DAC policies can be very fine-grained, they can simply list a permission set for each user and each object combination. This approach also has drawbacks, first of all the administration of the access control system is complex. As seen in Section 5.2.2, users need to be able to perform a lot of administration tasks. These tasks vary according to the policy, but they will need to be offered to the user which adds work to the application.

Another drawback which relates to MAC is that users and subjects are not distinguished by DAC policies. This makes controlling flow of information impossible. When a user is logged in, all the permissions associated with that user are available. Code that gets executed without the user's approval gets all the permissions.

5.2.4 DAC Example

Example policy The example implemented here does not allow change of ownership but does allow one level of grant authority delegation. Grant revocation is independent of the granting operation. The first thing needed to implement this DAC policy is to add properties to the entities that hold the relevant information (an access control list for each object is chosen). The data model will be the Document and User entities from the example introduced in Chapter 2. Ownership is represented as an inverse one-to-many relation between User and Document.

Policy data viewAccess is a set of users allowed to view the Document, editAccess is for editing, and grantingRights state which users may change the viewAccess and editAccess properties for that Document:

```
extend entity Document {
  owner -> User
  viewAccess -> Set<User>
  editAccess -> Set<User>
}
extend entity User {
  ownedDocuments -> Set<Document> (inverse=Document.owner)
}
```

Predicates With the access control lists in place, we can specify the predicates to be used in this DAC policy implementation:

```
access control rules
  predicate mayViewDocument (u:User, d:Document) {
    d.owner = u || u in d.viewAccess
  }
  predicate mayEditDocument (u:User, d:Document) {
    d.owner = u || u in d.editAccess
  }
```

Rules The predicates are used in the relevant page rules that enforce the policy in the application:

```
access control rules
  rule page viewDocument(d:Document) {
    mayViewDocument(principal,d)
  }
  rule page createDocument() {
    loggedIn()
  }
  rule page editDocument(d:Document) {
    mayEditDocument(principal,d)
  }
```

In these cases, where no action rules are explicit, the rules on pages are desugared to the same rules with a nested action check added, matching any action and having the same condition as the page check.

Administration For supporting a one-level grant delegation we add a `grantingRights` set to the `Document` entity. This holds the set of users that are allowed to change view and edit access:

```
extend entity Document {
  grantingRights -> Set<User>
}
```

The `editViewEditGrants` page allows editing the view and edit access for documents. `editGrantingRights` is created to allow an owner to edit the `grantingRights` of a `Document`:

```
define page editViewEditGrants(d:Document) {
  main()
  define body() {
    form {
      "ViewAccess: " input(d.viewAccess)
      "EditAccess: " input(d.editAccess)
      action("save", saveGrants(d))
      action saveGrants(d:Document) {
        d.save();
        return viewDocument(d);
      } } }
  define page editGrantingRights(d:Document) {
    main()
    define body() {
      form {
        "GrantingRights: " input(d.grantingRights)
        action("save", saveGranting(d))
        action saveGranting(d:Document) {
          d.save();
          return viewDocument(d);
        } } }
  }
```

Figure 5.12 shows the `editViewEditGrants` page and Figure 5.13 shows the `editGrantingRights` page.

Figure 5.12: Manage view and edit permissions

Figure 5.13: Manage granting permissions

The first page is accessible to the owner and the users in the `grantingRights` property of `Document`. It allows editing of the `viewAccess` and `editAccess` properties of a

Document. The second page is accessible only to the owner and can edit the grantingRights property of Document.

```
access control rules
  predicate mayChangeViewEditGrants (u:User, d:Document) {
    d.owner = u || u in d.grantingRights
  }
  predicate mayChangeGrantingRights (u:User, d:Document) {
    d.owner = u
  }
  rule page editViewEditGrants(d:Document) {
    mayChangeViewEditGrants(principal,d)
  }
  rule page editGrantingRights(d:Document) {
    mayChangeGrantingRights(principal,d)
  }
```

The resulting visible navigation links are inferred from this access control policy. Figure 5.14 shows the navigation options for the owner of an object. Figure 5.15 shows the navigation options for the owner of Document d2 with view access to d3 and view, edit, and granting access for d1.

d1	create new	
view		
edit		
view/edit access		
granting		

Figure 5.14: Owner of object with all rights

d1	d2	d3	create new
d1: view edit view/edit access			
d2: view edit view/edit access granting			
d3: view			

Figure 5.15: Resulting navigation links

5.3 Role-Based Access Control

Role-Based Access Control (RBAC) [16, 17, 19, 18, 8] models have been the basis for access control research in the last decade and they are also widely applied in application frameworks. The observation leading to RBAC is that individual users are usually not that important in deciding on permissions (besides auditing purposes), rather it is the task they need to perform that determines the necessary permissions. A *role* corresponds to a group of activities needed to perform a job or a task. These activities form the permissions that are linked with the role. When users are assigned to roles, they gain the permissions assigned to those roles. This better reflects organizational structures, which make common operations easy. For instance, a change of function inside an organization only requires a change of role assignment in the access control system. This action would have been a lot more complicated in a DAC policy where the permissions are directly linked to the user.

Roles have a lot in common with user groups, both link sets of users with sets of permissions, but there is a slight difference. The user group is a set of users, the users are always part of that group. The role is a set of privileges, the user can activate and deactivate the role. When roles represent real jobs, the distinction becomes smaller because the role will probably be activated all the time.

The main benefits of RBAC are:

- Access control administration: user-role assignment and role-permission assignment are separated. The administration is mostly concerned with user-role assignment, so the role/permission assignment can be hidden in an application.
- Hierarchical roles: many applications consist of a natural hierarchy of roles, where some roles subsume the permissions of others. An example of hierarchical roles is shown in Section 5.3.3.
- Least privilege: a user can activate the minimal role able to perform a task, this can protect the user from malicious code or inadvertent errors (similar to MAC policies).
- Separation of duties: no user should have enough permissions to abuse the system on their own, this can be enforced by separating the steps in critical actions among roles. For example, a user should not be able to create fake payments and also accept them.
- Constraint enforcement: the roles can be extended with constraints on activation or assignments, this allows more specialized access control policies.

The formalisation of RBAC in [19, 18] proposes a family of 4 models for RBAC. RBAC0 is the basic model, which consists of users, roles, and permissions as entities. User assignment to roles and permission assignment to roles determine the configuration of RBAC0. Besides these assignments there is also the concept of a session, which is an activated subset of the user's roles. The permissions from the roles in the session are the ones that can be used in access control decisions. The concept of user controlled sessions creates a distinction between subject and user similar to MAC policies. RBAC1 introduces role hierarchies to model lines of authority and responsibility. Senior roles inherit the permissions

of junior roles, and junior roles inherit the user assignment of senior roles (other implementations of role hierarchies allow activation of junior roles to support hierarchies). To support some encapsulation of permissions private roles can be constructed, that have only one junior role and holds all the permissions not needed higher up the hierarchy. RBAC2 adds constraints to the RBAC model, these constraints are related to the entities user, role, permission, and session. Examples of constraints are cardinality constraints on user-role assignment, limiting the number of users that may be assigned to a role, and mutually exclusive roles, not allowing a user to be assigned to multiple roles of the mutually exclusive set. Separation of Duties is one of the principle motivations for adding constraints to the RBAC model which explains the type of constraints possible. RBAC3 is the consolidated model of RBAC0, RBAC1, and RBAC2. This allows constraints on the hierarchy to be specified as well, which could be useful for decentralized configuration of the role hierarchy.

5.3.1 RBAC example

Example policy The basic RBAC example in this section has a configurable user-role assignment (configuration is allowed for users having the administrator role), but the permissions have been fixed in the rules. Roles will need to be explicitly activated in this application in order to use the permissions. viewer and editor roles refer to the viewing and editing of Document entities.

Policy data The first part of the implementation is adding a Role entity, and extending the User entity to hold user-role assignments. The currently activated Role is stored in the securityContext:

```
entity Role {
  name :: String
}
extend entity User {
  roles -> Set<Role>
}
extend session securityContext {
  activeRole -> Role
}
```

The roles in this application (administrator, viewer, and editor) are statically defined:

```
var admin : Role := Role {
  name := "Administrator"
};
var editor : Role := Role {
  name := "Editor"
};
var viewer : Role := Role {
  name := "Viewer"
};
```

Predicates With the roles in place, and the user-role assignment available, we can now link the permissions with the roles in predicates:

```
access control rules
  predicate mayViewDocument (r:Role) {
    r = viewer || r = editor
  }
  predicate mayEditDocument (r:Role) {
    r = editor
  }
```

Rules The next task is to actually enforce these checks on the different pages:

```
access control rules
  rules page viewDocument(*) {
    mayViewDocument(activeRole)
  }
  rules page createDocument() {
    mayEditDocument(activeRole)
  }
  rules page editDocument(*) {
    mayEditDocument(activeRole)
  }
  rules page editUserRoles(*) {
    mayEditRoles(activeRole)
  }
```

The explicit activation of roles also needs to be managed, this code has been placed in a function. The actual activation link is put in a template, so it can easily be included in a page or sidebar:

```
function activateRole(r:Role) {
  securityContext.activeRole := r;
}
define roleActivation() {
  form {
    list {
      for(r:Role) {
        listitem { actionLink(r.name,activate(r)) }
      }
    }
    action activate(r:Role) {
      activateRole(r);
    } } }
}
```

This function is protected by a rule that only allows activation if the user is assigned to the role. The template `roleActivation` can be simply made accessible to anyone, the activation is protected and the links will be hidden automatically if necessary:

```
access control rules
  rules function activateRole(r:Role) {
    r in principal.roles
  }
  rules template roleActivation() {
    true
  }
```



Figure 5.16: Role activation

Figure 5.16 shows the template where a user has both the viewer and editor roles. Figure 5.17 shows the options available to the viewer role and the same is shown for the editor role in Figure 5.18.

d0	d1	d2	d3
view d0	view d1	view d2	view d3
			view

Figure 5.17: View role activated

d0	d1	d2	d3	create new document
view d0	edit d0	view d1	edit d1	view d2
				edit d2
				view d3
				edit d3

Figure 5.18: Edit role activated

Administration The `editUserRoles` page needs to be defined to allow an admin to change the roles assigned to a user. This page is only accessible to an active administrator role:

```
define page editUserRoles(u:User) {
  action save(u:User) {
    u.save();
  }
  main()
  define body() {
    form {
      input(u.roles)
      action("save", save(u))
    }
  }
}
access control rules
predicate mayEditRoles(r:Role) {
  r = admin
}
rules page editUserRoles(*) {
  mayEditRoles(securityContext.activeRole)
}
```

Figure 5.19 shows the input form of the `editUserRoles` page, and Figure 5.3.1 shows the home page for a logged in administrator.



Figure 5.19: Admin role

Welcome Alice
 Activate Role:
 Administrator

edit roles of Alice
 edit roles of Bob
 edit roles of Charlie
 edit roles of Dave

Figure 5.20: Admin role activated

5.3.2 Parameterized Roles

Applications sometimes need to cope with multiple organizations or user groups having separate role-based access control requirements. This situation is not easy to deal with in application frameworks where a single role structure is assumed. Roles will need to be parameterized so they are linked with the application section they are valid in.

In this example a Role is linked to a Project. The ProjectRole entity represents the roles that can be assigned and activated:

```
entity Role {
  name :: String
}
entity ProjectRole {
  role -> Role
  project -> Project
  description :: String (name) :=
    project.name + role.name
}

extend entity User {
  roles :: Set<ProjectRole>
}
extend session securityContext {
  activeProjectRole -> ProjectRole
}
```

The same static roles are used as in the previous example. A function takes care of creation and retrieval of ProjectRole entities:

```
function retrieveProjectRole(p:Project,r:Role):ProjectRole {
  var proles : List<ProjectRole> :=
    select pr from ProjectRole as pr
    where (pr._role = ~r) and (pr._project = ~p);
  if(proles.size()>0) {
    return proles.first();
  }
  var newpr : ProjectRole := ProjectRole {
    role := r
    project := p
  };
  newpr.save();
  return newpr;
}
```

The rules that determine the permissions need the relevant project information in the check:

```
predicate mayViewDocument (pr:ProjectRole,d:Document) {
  (viewer = pr.role || editor = pr.role) && d.project = pr.project
}
predicate mayEditDocument (pr:ProjectRole,d:Document) {
  editor = pr.role && d.project = pr.project
}
predicate mayEditRoles(pr:ProjectRole,p:Project) {
  admin = pr.role && p = pr.project
}
```

The next task is applying checks to pages and actions, the `createDocument` page is available when the logged in user has an editor role, but the actual project the role is related to is checked upon saving:

```
rule page viewDocument(d:Document) {
  mayViewDocument(activeProjectRole,d)
}
rule page createDocument(){
  activeProjectRole.role = editor
  rule action save(d:Document) {
    mayEditDocument(activeProjectRole,d)
  }
}
rule page editDocument(d:Document) {
  mayEditDocument(activeProjectRole,d)
}
rule page editUserRoles(u:User) {
  mayEditRoles(activeProjectRole,activeProjectRole.project)
  rule action addRole(u:User,r:Role,p:Project) {
    mayEditRoles(activeProjectRole,p)
    && !(retrieveProjectRole(p,r) in u.roles)
  }
  rule action removeRole(u:User,r:Role,p:Project) {
    mayEditRoles(activeProjectRole,p)
    && retrieveProjectRole(p,r) in u.roles
  }
}
rule function activateRole(pr:ProjectRole) {
  pr in principal.roles
}
```

In this example the adding and removing of roles are defined as single operations instead of simply changing the list directly, and again the access control rules take care of hiding the unwanted links:

```
define page editUserRoles(u:User) {
  action addRole(u:User,r:Role,p:Project) {
    u.roles.add(retrieveProjectRole(p,r));
  }
  action removeRole(u:User,r:Role,p:Project) {
    u.roles.remove(retrieveProjectRole(p,r));
  }
}
main()
define body() {
  form {
    par { "User: " output(u) }
    for(project:Project) {
      par {
        "Project: " output(project.name)
        for(role:Role) {
          par{actionLink("disable "+role.name+" role",removeRole(u,role,project))}
          par{actionLink("enable "+role.name+" role",addRole(u,role,project))}
        }
      }
    }
  }
}
```

5.3.3 Hierarchical RBAC

In the RBAC model it is possible to define role hierarchies. This can be used for a more concise and manageable implementation of a policy. The relation between two roles indicated by $r1 \geq r2$ results in the following properties:

- $r1$ is a senior role of $r2$
- $r2$ is a junior role of $r1$
- $r1$ acquires permissions of $r2$
- $r2$ acquires user membership of $r1$

One way of implementing a hierarchy is to add a set of all senior roles and a set of all junior roles to the Role entity:

```
extend entity User {
  roles -> Set<Role>
}
extend session securityContext {
  activeRole -> Role
}
entity Role {
  name :: String
  seniorRoles -> Set<Role>
  juniorRoles -> Set<Role>
  viewEditPermission -> Set<Document>
}
```

Initial role permissions can now take hierarchies into account:

```
var viewer1 : Role := Role {
  name := "Viewer1"
  seniorRoles := {viewer2}
  viewEditPermission := {d1}
};
var viewer2 : Role := Role {
  name := "Viewer2"
  seniorRoles := {viewer3}
  juniorRoles := {viewer1}
  viewEditPermission := {d2}
};
var viewer3 : Role := Role {
  name := "Viewer3"
  juniorRoles := {viewer2}
  viewEditPermission := {d3}
};
```

An example predicate that checks whether viewing a certain document is allowed if a certain role is active can be stated as follows:

```
predicate mayViewDocument (r:Role,d:Document) {
  d in r.viewEditPermission
  || Or[ mayViewDocument(role,d) | role: Role in r.juniorRoles ]
}
```

Chapter 6

Case Studies

This chapter will present some applications constructed with WebDSL and the access control sub-language. The applications are given enough functionality to provide an interesting setting for access control but still be manageable. Both normal application code and access control declarations are covered in the discussions of these applications.

Section 6.1 presents a wiki. Section 6.2 presents a social networking service. Section 6.3 presents a conference system.

6.1 Wiki

The first example case presented in this chapter is a wiki. The basic idea of a wiki is that users can create, edit and link together pages (all specified in a simple markup language). The users can collaboratively create a website that contains a lot of information. The wiki application provides search options for finding information among its pages. Version control is used to comment on changes and possibly do a rollback. Different access control policies are chosen for different types of wikis. Some are very open, allowing any user to edit pages. The argument for this soft security is that it is also easy to correct mistakes or vandalism, any user can act as moderator. These open wikis grow fast, the best example is the online encyclopedia Wikipedia. Other wikis are very closed, only specifically authorized users are allowed to edit pages. This will result in a slow growing wiki with little vandalism. Wikis do not have to be public, they can also be used on an intranet for knowledge sharing in an organization.

The entities for the wiki case are Page to store pages and User to hold user information and allow authentication and access control:

```
entity Page {  
  name :: String (name, id)  
  text :: WikiText  
}  
entity User {  
  name :: String(name)  
}
```

The view of a page shows the contents of the page and a link to the page edit section. Upon initialization the contents of the requested page are checked and the edit page (Figure 6.1) is presented if the text of the page was empty:

```
define page page(p : Page) {
  init {
    if(p.text = "") {
      goto editPage(p);
    }
  }
  main()
  define body() {
    section {
      header {output(p.name)}
      output(p.text)
      section { navigate(editPage(p)) { "Edit this page" } }
    }
  }
}
```

create new page	
Edit Page firstpage	
Name:	<input type="text" value="firstpage"/>
Text:	<div> This is the first page. Here's a link to [[page(secondpage) the second page]] </div>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

Figure 6.1: Edit page

The rules that govern access for this wiki allow viewing of pages to anyone but require a logged in user for editing:

```
access control rules
  rule page editPage(p:Page) {
    loggedIn()
  }
  rule page page(*) {
    true
  }
```


Figure 6.3 shows the view of a page when logged in, Figure 6.2 shows the view when not logged in, Figure 6.4 shows the listing of all the pages.

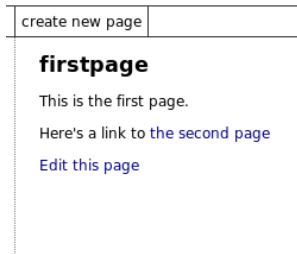
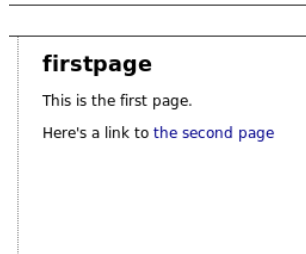


Figure 6.2: Wiki not logged in

Figure 6.3: Wiki logged in

Figure 6.4: All pages

A different policy could be that only a specified list of users is able to edit the pages. The global variable `authorizedUsers` is used to hold this set of users, an administration page is defined to edit it, and the `isAdmin` property is used to determining access to administration:

```
var authorizedUsers : Set<User>;

extend entity User {
  isAdmin :: Bool
}

define page administration() {
  main()
  define body() {
    section {
      header {"Administration"}
      form {
        input(authorizedUsers)
        action("Save", save())
      }
      action save(){
        authorizedUsers.save();
      }
    }
  }
}
```

The rules to protect this variation of the wiki:

```
access control rules
  rule page editPage(p:Page) {
    principal in authorizedUsers
  }
  rule page page(*) {
    true
  }
  rule page administration() {
    principal.isAdmin
  }
```

6.2 Social Networking Service

The second example case is a social networking service, which aims to bring people together. Users can search and group with other users that share interests and activities. The application allows various ways for the users to interact. Users can create profiles with pictures, information, and links to their friends' or groups' pages. Well-known examples of social networking services are Myspace and Facebook. The access control policies for social networking services is often linked with the friend or group structures. Users can specify what part of their personal information is visible to their friends, their groups' members, or any other user.

The data model for the social networking service presented here consists of the `User` entity for representing users, the `UserGroup` entity for user groups, the `UserPage` entity to store a user's personal page and profile, the `GroupPage` entity for information related to the group, and finally the `FriendRequest` and `MembershipRequest` entities which are used to hold requested friendships or group memberships:

```
entity User {
  username :: String(name)
  password :: Secret
  page     -> Page
  friends  -> Set<User>
  groups   -> Set<UserGroup>
}
entity UserGroup {
  name      :: String (name)
  owner     -> User
  moderators -> Set<User>
  members   -> Set<User> (inverse=User.groups)
  page      -> GroupPage
}
entity Page {
  name      :: String (name)
  content   :: Text
}
entity UserPage extends Page {
  owner     -> User (inverse = User.page)
}
entity GroupPage extends Page {
  group     -> UserGroup (inverse = UserGroup.page)
}
```

These entities are presented by straightforward view, edit, and create pages, Figure 6.6 shows the view page of a user. Figure 6.5 shows the home page which simply gives an index of the member pages (in this case all three are accessible).

The rules protecting the pages are supported by a small data structure that holds relevant information about the accesses allowed to owned resources. The possible access options for a member page in this example are public, friends, and private (blocked for anyone except owner). Group pages can be protected with public, members, private (only moderators and owner):

```
entity ViewMode {
  name :: String
}
```

```

var pub : ViewMode := ViewMode{name:="public"};
var fri : ViewMode := ViewMode{name:="friends"};
var priv : ViewMode := ViewMode{name:="private"};
var mem : ViewMode := ViewMode{name:="members"};
extend entity User {
  viewAccess :: ViewMode
}
extend entity UserGroup {
  viewAccess :: ViewMode
}

```

The rules enforcing the protection of the user and group pages are specified as follows:

```

access control rules
  predicate viewAllowed(u:User){
    (u.viewAccess = pub)
    || (u.viewAccess = fri && principal in u.friends)
    || (u.viewAccess = priv && principal = u)
  }
  rule page viewUser(u : User) {
    viewAllowed(u)
  }
  predicate groupViewAllowed(ug : UserGroup) {
    (ug.viewAccess = pub)
    || (ug.viewAccess = mem && principal in ug.members)
    || (ug.viewAccess = priv && (principal = ug.owner || principal in ug.moderators))
  }
  rule page viewUserGroup(ug : UserGroup) {
    groupViewAllowed(ug)
  }

```

The application needs to manage friends and groups, which involves a request for friendship/membership by one user and a confirmation by that friend or the groups' moderators. Since these are similar operations only the friends case is shown here:

```

extend entity User {
  friendRequests -> Set<FriendRequest>
  incomingFriendRequests -> Set<FriendRequest>
  function potentialFriend (f:User): Bool {
    var isrequesting : Bool := f in [fr.requester for (fr : FriendRequest in this.incomingFriendRequests)];
    var isrequested : Bool := f in [fr.requestee for (fr : FriendRequest in this.friendRequests)];
    var isfriend : Bool := f in this.friends;
    if (this != f && !isrequesting && !isrequested && !isfriend) {
      return true;
    }
    else {
      return false;
    }
  }
}
entity FriendRequest {
  requester -> User (inverse=User.friendRequests)
  requestee -> User (inverse=User.incomingFriendRequests)
}
define friendNewRequests(u:User) {
  table {
    form {
      for(us : User where u.potentialFriend(us) ) {
        row{output(us.username) action("request friendship",reqFriend(us)) }
      }
    }
    action reqFriend(us: User) {

```

```

    var freq : FriendRequest := FriendRequest {
      requester := u
      requestee := us
    };
    freq.save();
    u.friendRequests.add(freq);
  } } } }
define friendIncomingRequests(u:User) {
  table {
    form {
      for(incfreqs : FriendRequest in u.incomingFriendRequestsList) {
        row {
          output(incfreqs.requester.username)
          action("accept", acceptFriendRequest(incfreqs))
        }
      }
    }
    action acceptFriendRequest(f:FriendRequest) {
      f.requester.friends.add(f.requestee);
      f.requestee.friends.add(f.requester);

      f.requester.friendRequests.remove(f);
      f.delete();
    } } } }

```

Figure 6.7 shows the friendNewRequests template below the list of friends.

All members

[Alice](#)
[Bob](#)
[Charlie](#)

All groups

Member page: Alice

Member: Alice
 My Friends:

• [Bob](#)

My Groups:

This is Alice's page.

[Edit my page](#)

My Friends

[Bob](#)

New Friendship requests

Charlie	request friendship
Dave	request friendship

Figure 6.5: Social network home

Figure 6.6: Your own page

Figure 6.7: Friends

Editing of user information and page is only allowed to that user, group information can only be edited by the owner or the moderators:

```

access control rules
pointcut groupediting(ug : UserGroup) {
  template group*(ug),
  page editUserGroup(ug)
}
rule pointcut groupediting(ug : UserGroup) {
  principal = ug.owner || principal in ug.moderators
}
pointcut userediting(u:User) {
  template userEdit(u),
  page editUser(u),
  template friend*(u),
  template group*(u),
  page groups(u),
  page friends(u)
}

```

```
rule pointcut userediting(u:User) {
    u = principal
}
```

Figure 6.8 and Figure 6.9 show the edit user configurations that were active in the previous three figures. The logged in user there was Alice; she has access to three member pages: her own, Bob's page (she is a friend and Bob allows friends to see his member page), and Charlie's page (who allows anyone to view his page).

Figure 6.8: Administration

Figure 6.9: More administration

6.3 Conference System

The third example case is a conference system that manages paper submissions, reviewing, and conference acceptance. The Program Chair of a conference can create and manage a conference in the system. Some of the operations this managing entails are:

- Opening and closing submissions.
- Adding reviewers and program committee members.
- Assigning reviewers and program committee members to papers.
- Accepting or rejecting papers.
- Manage manual conflicts between users.

The last item in the list is directly related to access control administration. Static conflicts could be specified, e.g., stating that people from the same company shouldn't be able to see/review/recommend each others work. But it is also possible for the Program Chair to

specify manual conflicts between users for any other reason which will be shown in this example.

The data model for the conference system is defined as follows:

```
entity Conference {
  name :: String (name)
  topic :: String
  programChair -> User
  programCommittee -> Set<User>
  reviewCommittee -> Set<User>
  submissionsOpen -> Bool
  submittedPapers -> Set<Paper>
  acceptedPapers -> Set<Paper>
  rejectedPapers -> Set<Paper>
}
entity Paper {
  title :: String (name)
  authors -> Set<User>
  conference -> Conference (inverse=Conference.submittedPapers)
  assignedReviewers -> Set<User>
  reviews -> Set<Review>
  assignedPC -> Set<User>
  recommendations -> Set<Recommendation>
}
entity Review {
  comments :: String
  author -> User
  paper -> Paper
}
entity Recommendation {
  comments :: String
  author -> User
  paper -> Paper
}
entity User {
  name :: String (name)
  email :: Email
  papers -> Set<Paper> (inverse=Paper.authors)
  organization -> Organization
}
entity Organization {
  name :: String (name)
}
```

The Conference entity represents a conference with the creator of the conference as Program Chair. Figure 6.10 shows the view page for the Conference entity. The Paper entity holds information related to a specific paper. Review and Recommendation can be attached to a paper by the assigned reviewer and assigned program committee member respectively.

In order to get an idea of the interface of the conference system the menu is shown in Figure 6.11. The code for this menu is the following:

```
define themenu() {
  for(c:Conference) {
    menu {
      menuheader{ navigate(conference(c)){output(c.name)} }
      menuitem{ navigate(editConference(c)){"edit"}}
      menuitem{ navigate(managePapers(c)){"manage papers"}}
      menuitem { navigate(addConflict(c)){output("add conflict")} }
    }
  }
}
```

ICWE08

Name:	ICWE08
Topic:	International Conference on Web Engineering
ProgramChair:	Alice
ProgramCommittee:	♦ Bob
ReviewCommittee:	♦ Dave
SubmissionsOpen:	true
SubmittedPapers:	♦ Declarative Access Control for Web Applications
AcceptedPapers:	
RejectedPapers:	
Userconflicts:	

Figure 6.10: Conference overview

```

    menuitem { navigate(manageConflicts(c)){output("manage conflicts"}} }
  }
}
menu { menuheader { navigate(createConference()){output("new conference"}} } }
menu { menuheader { navigate(createUser()){output("new user"}} } }
menu { menuheader { navigate(submitPaper()){output("submit paper"}} } }
}

```

Most of these navigation options are directing the user to pages with simple input forms.

`managePapers(c)` shows a list of all papers in conference `c` with links to the `managePaper(p:Paper)` page:

```

define page managePaper(p:Paper) {
  main()
  define body() {
    table {
      row{output(p.name)}
      viewRowsPaper(p)
      row{navigate(assignReview(p)) { "assign reviewers/pc" }}
      row{navigate(paperReview(p)) { "create review"}}
      row{navigate(paperRecommend(p)) { "create recommendation"}}
      row{navigate(paperAccepting(p)) { "acceptance for conference"}}
    } } }
}

```

The links in the `managePaper` page consist of pages that allows various operations on the paper. The `assignReview` page is shown here as an example:

```

define page assignReview(p:Paper) {
  main()
  define body() {
    table {
      row{output(p.name)}
      form {
        row{"assign reviewers "}
        row{input(p.assignedReviewers)}
      }
    }
  }
}

```

```

        row{"assign program committee "}
        row{input(p.assignedPC)}
        row{action("save",save(p))}
    }
}
action save(p:Paper) {
    return managePaper(p);
} } }

```

Manual conflicts can be specified between users for a certain conference. Only the Program Chair is allowed access to this functionality. The data extensions used to hold the user conflict configurations are:

```

entity ConflictUserSet {
    conflictingusers -> Set<User>
}
extend entity Conference {
    userconflicts -> Set<ConflictUserSet>
}

```

Some pages are created to manage the conflicts data; addConflict allows specifying an additional set of conflicting users, and manageConflicts allows deletion of conflicts. The usersInDeclaredConflict function iterates a set of ConflictUserSet objects to determine whether two users have a conflict declared:

```

define page addConflict(con:Conference) {
    main()
    define body() {
        var cset : ConflictUserSet := ConflictUserSet{};
        form {
            table {
                row{input(cset.conflictingusers)}
                action ("Add conflict",save(cset))
            }
        }
        action save(c:ConflictUserSet) {
            c.save();
            con.userconflicts.add(c);
            return manageConflicts(con);
        } } }
define page manageConflicts(con:Conference) {
    main()
    define body() {
        form {
            for(c:ConflictUserSet in con.userconflictsList) {
                table {
                    row{output(c.conflictingusers) action ("Remove conflict",remove(c))}
                }
            }
        }
        action remove(c:ConflictUserSet) {
            con.userconflicts.remove(c);
            return home();
        } } }
function usersInDeclaredConflict(u1:User, u2:User, conflictsets:Set<ConflictUserSet>):Bool {
    for(conflictset : ConflictUserSet in conflictsets) {
        if(u1 != u2 && u1 in conflictset.conflictingusers && u2 in conflictset.conflictingusers) {
            return true;
        }
    }
}

```



```

    }
  }
  return false;
}

```

Figure 6.12 shows an input form for manual conflicts.



Figure 6.11: Chair menu



Figure 6.12: Declare a conflict among users

The following predicates use data of user conflicts to determine access:

```

access control rules
  predicate isValidRevPCAssignment(p:Paper) {
    !(Or [authorInConflict(a,p) | a:User in p.authors])
  }
  predicate authorInConflict(u:User,p:Paper) {
    conflictingUsers(u,p.assignedReviewers,p.conference)
    || conflictingUsers(u,p.assignedPC,p.conference)
  }
  predicate conflictingUsers(u1:User,uset:Set<User>,c:Conference) {
    Or [u1.organization = u2.organization
        || usersInDeclaredConflict(u1,u2,c.userconflicts) | u2:User in uset]
  }

```

The conflicts and normal access control checks are connected to the pages with the following declarations. First the principal entity is declared:

```

access control rules
  principal is User with credentials name

```

Open sections of the application are specified in a pointcut:

```

pointcut openSections() {
  page home(),
  template sidebar(),
  page user(*),
  page createUser(*),
  page conference(*),
  page createConference(*),
  page paper(*),
  page submitPaper(*)
}
rule pointcut openSections() {

```

```

    true
}

```

Access to the conference paper management pages is allowed to the Program Chair, the program committee, and the reviewers:

```

predicate mayManageConferencePapers(u:User,c:Conference) {
    u = c.programChair
    || u in c.reviewCommittee
    || u in c.programCommittee
}
rule page managePapers(c:Conference) {
    mayManageConferencePapers(principal,c)
}
rule page managePaper(p:Paper) {
    mayManageConferencePapers(principal,p.conference)
}

```

The sections that are only accessible to the Program Chair are shown next:

```

pointcut chairSections(c:Conference) {
    page editConference(c),
    page addConflict(c),
    page manageConflicts(c)
}
rule pointcut chairSections(c:Conference) {
    principal = c.programChair
}
pointcut chairSections(p:Paper) {
    page assignReview(p),
    page paperAccepting(p)
}
rule pointcut chairSections(p:Paper) {
    principal = p.conference.programChair
}
rule page assignReview(p:Paper) {
    true
    rule action save(p:Paper) {
        isValidRevPCAssignment(p)
    }
}

```

Reviewers and program committee members assigned to papers are allowed to review and recommend respectively:

```

rule page review(r:Review) {
    principal = r.author
    || principal = r.paper.conference.programChair
}
rule page recommendation(r:Recommendation) {
    principal = r.author
    || principal = r.paper.conference.programChair
}
rule page paperRecommend(p:Paper) {
    principal in p.assignedPC
}
rule page paperReview(p:Paper) {
    principal in p.assignedReviewers
}

```

Figure 6.13 shows the manage view of a paper when the Program Chair is logged in. Figure 6.14 shows the manage view of a paper when a reviewer of that paper is logged in.

ICWE08 ▾	new conference	new user	submit paper		ICWE08 ▾	new conference	new user	submit paper	
Declarative Access Control for Web Applications					Declarative Access Control for Web Applications				
Title: Declarative Access Control for Web Applications					Title: Declarative Access Control for Web Applications				
Authors: <ul style="list-style-type: none">• Danny• Eelco					Authors: <ul style="list-style-type: none">• Danny• Eelco				
Conference: ICWE08					Conference: ICWE08				
AssignedReviewers: <ul style="list-style-type: none">• Dave					AssignedReviewers: <ul style="list-style-type: none">• Dave				
Reviews:					Reviews:				
AssignedPC: <ul style="list-style-type: none">• Bob					AssignedPC: <ul style="list-style-type: none">• Bob				
Recommendations:					Recommendations:				
assign reviewers/pc					create review				
acceptance for conference									

Figure 6.13: Chair view of submitted paper

Figure 6.14: Reviewer view of submitted paper

Chapter 7

Related Work

7.1 Language Design

Mikkonen and Taivalsaari [12] argue that software engineering principles have degraded with the recent paradigm shift to web applications. The incremental growth from static HTML pages to desktop application replacements has left a trail of languages which require tool support to cope with. Developers need to learn these technologies which prevents them from focussing on learning actual web application design principles, such as access control. We review the software engineering principles of [12] for WebDSL access control: *Separation of concerns*: One of the main goals of WebDSL access control is achieving separation of concerns while still having an integrated language with static verification. WebDSL applications can be easily adapted to support a different access control policy. *Information hiding*: The details of how and where access control is applied is hidden in the semantics of access control rules. *Consistency*: Access control checks are specified in the same expression language that is used in other parts of WebDSL applications. *Simplicity*: one mechanism can be used to define a wide range of policies. *Reusability*: Policy specifications can be reused for other pages and in other WebDSL applications. *Portability*: WebDSL access control is translated to normal WebDSL which is a model that abstracts from platform specific details.

Another benchmark is the requirements formulated by Evered and Bögeholz [7] for an ideal access control mechanism based on a case study of a health information system. *Concise*: Access control checks in rules simply become the boolean WebDSL expression that is executed for determining access. Reuse through predicates limits the amount of mechanical repetition. *Clear*: Matching of access control rules with resources is based on clear semantics and can easily be verified to be correct. *Aspect-oriented*: Access control can be specified separately from the rest of a WebDSL application, weaving takes care of integration. *Fundamental*: All resources that can be protected by WebDSL access control are denied access by default, this forces the application developer to explicitly specify conditions for access. *Positive*: The access control rules determine conditions for allowing access, they are positive authorizations. *Need-to-know*: Access control can be used to hide information on pages, more specifically template contents and navigation links. *Efficient*:

Because access control is integrated with the rest of the WebDSL application it can use the same database interface and caching mechanisms.

The paper also includes ideas on at what level access control should be applied in an ideal component based system. Data should be represented by abstract objects which correspond to objects in the real world together with operation interfaces which have application-level meaning. These operations are the ones that should be protected by access control. While working with WebDSL access control we also observed that there should be a more abstract, application-level representation of data instead of the simple entities. This would allow WebDSL access control to infer page and action checks now specified manually.

Tschantz and Krishnamurti [22] present a set of properties for examining the reasonability of access control policies under enlarged requests, policy growth, and policy decomposition. We discuss their properties for WebDSL access control. *Deterministic*: If the application's data is considered part of the policy, then identical requests always result in the same access control decision. *Totality*: A decision to allow or deny is always made. The default to deny can be caused by a condition evaluating to false, no rule matching the resource, or an error occurring during checking. *Safe*: Since WebDSL access control is integrated with the application it is not possible to make incomplete access control requests. *Independent composition*: It is possible to reason about rules in isolation, combining them will not change the result of an *individual* rule. *Not monotonic*: Decisions can change from granting to non-granting by adding another rule, caused by the single rule combination strategy of taking the conjunction of matching rules. We believe a standard way of combining rules by conjunction is easier to comprehend than having multiple rule combination strategies, and it results in a deny overrides strategy that we consider a safe default combination of rules.

Their motivating example is the following list of policy rules:

1. faculty members can assign grades
2. students cannot assign grades
3. non-faculty members can enroll in courses.

These rules can be specified in WebDSL access control, when you link the actions to a page where that action is possible:

```
rule page assignGrades(u:User) {
  principal.isFaculty
}
rule page assignGrades(u:User) {
  !principal.isStudent
}
rule page enroll(c:Course) {
  !principal.isFaculty
}
```

The resulting access decisions are:

- faculty member, not student: can only assign grades.

- student, not faculty member: can only enroll in courses.
- both faculty member and student: denied access.
- not faculty member and not student: can only enroll in courses.

To get a more sensible result an extra rule is needed that only allows enrolling to students. It is easy to determine these results from the stated rules, the only knowledge of the language necessary is that rules on the same resource are combined by a conjunction and that no applicable rule for a request results in a deny.

7.2 Policy Languages

The Ponder policy specification language [5] is a policy language aimed at specifying access control for firewalls, operating systems, databases, and Java programs. The language has many features such as built-in notions of groups and roles, delegation, obligations (which specify what a user must do), meta-policies for defining constraints on the policies themselves (for example SoD). The paper presents examples of policies to cover the different elements available in the language. However, there is no information on how the policies are enforced in different contexts. This makes it hard to compare the results of using Ponder to specify policies and our own approach. The discussion of the runtime model for Ponder [6] suggests the usage of generic access-controllers that are close to the target, for example the method dispatch mechanism in the case of a programming language, similar to the framework approaches we discuss in this section. Several of the elements available in the Ponder language, such as delegation, obligations, and policy reuse, are still open for exploration in WebDSL and provide options for future work.

XACML (eXtensible Access Control Markup Language) [13] is a standard that describes both a policy language and an access control decision request/response language (both written in XML). A policy is described using rules that specify conditions for being applicable to a request. The requests in the request/response language are access control queries and the responses can be permit, deny, indeterminate (an error occurred) or not applicable (cannot answer this request). In WebDSL we opted for 'allow' and 'deny' as only results of rules. Rules are combined in policy sets, and policy sets in a policy, both of these operations are controlled by selecting a combination algorithm. In WebDSL access control rules are combined by using the conjunction of the expressions, rule sets can be combined in a policy with a boolean expression over the sets. For finer grained authorization in XACML attributes are used, these are characteristics of the subject, resource, action, or environment in which the access request is made. Most of these attributes have to be specified in the XML request message when checks are needed. WebDSL access control is integrated with the WebDSL application, there is no need for creating requests and explicitly transmitting all the required data, this avoids any inconsistencies.

7.3 Frameworks

Acegi [1] is the security component of the Spring Java web application framework. In keeping with the Spring approach, Acegi is based on XML configuration of the framework components. Enforcing access control is done by an aspect mechanism, these aspects are the components that need to be configured to protect either URLs or specific methods. The weaving occurs when the protected resource is invoked for the first time. In WebDSL access control weaving is done statically which allows better error reporting and compile-time guarantees. In Acegi a security context is available where the authenticated user can be stored together with a collection of granted authorization objects, for instance roles. This can be used to specify RBAC checks in the XML configuration. These checks are application wide and cannot be further customized with conditions. An active roles collection as used in the RBAC example in Section 5 for enforcing the concept of least privilege has to be managed in the application to work in this framework. When fine-grained access control is needed Acegi offers a DAC policy implementation which can protect generic objects by listing the permissions available for each user. This is stored in different database tables than the actual application. How to combine the supported RBAC and DAC has to be specified in the XML configuration. Fine-grained access control must be encoded in the generic DAC policy, which might involve duplicating information available in the application such as ownership of an object. WebDSL provides an integrated way to specify access control, where policies are combined in the rules, data is integrated with the application, and accessible for administration.

The Seam Java web application framework [28] offers a security API for access control. The basic mode is controlled by including restrict annotations in the application code which verify whether a principal has a certain role when accessing the annotated Java method or page URL. There is also an interface where you can provide a DAC policy by implementing a has permission function which takes the type of object, the action, and a reference to the object in particular. This is similar to the options Acegi has to offer and suffers the same drawbacks. The advanced mode consists of using JBoss Rules [28] for determining permissions. This is a logic language with an inference engine that can deduce permissions from facts available in the engine. These facts specify the access control policy. Although JBoss Rules allows separation of concerns for access control, it is a separate engine which must be invoked correctly. Any check based on information related to data in the application needs to have this data supplied in the requests. WebDSL access control is integrated with the application and does not have the risk of inconsistencies between known data during access control checks and actual data of the application. Besides the integration issue, the semantics of the rules is not specific to access control but simply generic JBoss Rules, which prevents any assumptions to be made about the access control policy (such as the visibility inference in WebDSL).

Chapter 8

Discussion

8.1 Evaluation

While WebDSL access control elements are specified in separate language constructs, the effect requires integrating with the WebDSL application. The first type of integration is the use of WebDSL expressions for specifying access control conditions. Besides the expressions, the data model is also completely accessible for use in checks. Matching of arguments allows access to all the data relevant for the type of resource that a rule protects. This data model integration works the other way around as well, the data from WebDSL access control is available in the WebDSL application which provides a simple way to produce administration support. The semantics of the access control rules illustrate that it is a pure WebDSL to WebDSL transformation, which lets WebDSL access control reuse all the code generation. We observed that it is tempting to add constraints in the rules that are not related to access control, but for instance to data validation, which indicates that a better mechanism for data validation is needed in WebDSL in addition to access control.

WebDSL access control encodes policies and can be seen as a high-level mechanism for access control. The default access control decision of denying access provides a safe default and allows an incremental approach when specifying the policy. Section 5 shows that WebDSL access control is transparent and concise in expressing Mandatory, Discretionary, and Role-Based Access Control and provides good support for the management of such policies. Besides these paradigms there is a strong connection with the application allowing application-specific customizations to be expressed easily.

By viewing access control as a proper language element, it becomes possible to infer related elements from the policy specification. Assumptions can be made about navigation to pages and visibility of page elements. This greatly increases the productivity of application developers and also helps to make sure the applications produced are consistent. This is a distinction from using generic aspect mechanisms for access control, because there the semantic value of access control rules is lost.

8.2 Future Work

Although we have created a flexible language for access control, abstractions for common policies are still possible. Mainly for RBAC, built-in abstractions can provide a more efficient means to support a role-based policy. Separation of duty checks as presented in the RBAC example in Section 5 could be expressed declaratively instead of explicitly specifying where the check should be enforced.

The access control rules specified in WebDSL are coupled with the presentation of the application, this allows inconsistencies where similar pages have different rules. Better abstraction and encapsulation of entities and their operations is needed in WebDSL which will allow access control to protect entities and their specified interface directly and infer the current type of checks. This would require tracing the use of these interfaces in pages, but will provide better consistency verification of the access control policy.

Logging of access control requests/decisions (not just writing to a text file but integrated logging, persisted like other application data) is required for doing access control audits and intruder detection. Adding this to WebDSL would be a major improvement. An important point here is that the amount of logging data can easily become unwieldy; a specification is needed that determines what information is stored, how detailed the entries are, and when they may be deleted or archived.

The DAC policy example presented in Section 5 included some facilities for delegation of access control, this could be specified in a more general way. Several models have been proposed for delegation [27, 14, 29, 30], which also shows there is a need for abstracting delegation details from access control policies. Supporting high level definitions of access control delegation would be an interesting addition to WebDSL.

WebDSL does not have support for scheduled operations yet, which is why we have chosen to ignore time constraints for now. Temporal policies have been modeled in the literature [11] and the application of such models would provide an interesting case for WebDSL access control. Workflow abstractions for WebDSL are being researched, which also provides options for exploring access control policies in that area. This can be compared to other studies regarding workflow and access control [4].

Digital rights management has been connected to access control in [15], where a conceptual model for usage control is presented. Adding this functionality to WebDSL would provide useful insights in the applicability of such an approach.

Chapter 9

Conclusion

The extension of WebDSL with a declarative access control language provides us with insight in how enforcing access control could be done better in general web applications. Firstly, the use of integrated, access control specific, aspect-oriented language elements result in a clear extension of the base language. Secondly, WebDSL access control shows that various policies can be expressed with simple constraints, allowing concise and transparent mechanisms to be constructed. Finally, the advantage of having a language element for access control, allowing assumptions to be made about the related parts in the application. Practical solutions for access control often consist of libraries or generic aspect-oriented implementations of fixed policies. These rarely have clear interfacing capabilities and require manual extension and integration with the application code. The extensions and integrations provide room for errors that possibly invalidate the whole access control policy. The realization that access control as a language element is necessary will provide the means to defeat the errors caused by encoding policies in application code. Integration of the language means that language additions influence the semantics of access control elements. New basic elements and new abstractions require new rule types. Using transformational semantics access control rules can be defined clear and concise.

Bibliography

- [1] B. Alex. Acegi Security, Reference Documentation 1.0.5.
- [2] A. Anderson. XACML Profile for Role Based Access Control (RBAC). *OASIS Access Control TC Committee Draft*, 1:13, 2004.
- [3] Messaoud Benantar, editor. *Access Control Systems*. Springer, 2006.
- [4] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, 1999.
- [5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. *Policies for Distributed Systems and Networks: Int. Workshop, Policy 2001, Bristol, Uk, January 29-31, 2001: Proceedings*, 2001.
- [6] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. Towards a Runtime Object Model for the Ponder Policy Language. *in DSOM 2000 conference*, 2000.
- [7] Mark Evered and Serge Bögeholz. A case study in access control requirements for a health information system. In *Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation (ACSW Frontiers '04)*, pages 53–61, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.
- [8] D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli. *Role-based Access Control*. Artech House, 2003.
- [9] R. Johnson et al. *Professional Java Development with the Spring Framework*. Wrox Press Birmingham, UK, 2005.
- [10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. *Ecoop 2001-Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001: Proceedings*, 2001.
- [11] U. Latif. A Generalized Temporal Role-Based Access Control Model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, 2005.

- [12] T. Mikkonen and A. Taivalsaari. Web Applications-Spaghetti Code for the 21st Century. Technical Report TR-2007-166, Sun Microsystems, June 2007.
- [13] T. Moses et al. eXtensible Access Control Markup Language (XACML) Version 2.0. *OASIS Standard*, 200502, 2005.
- [14] S.Y. Na and S.H. Cheon. Role delegation in role-based access control. *Proceedings of the fifth ACM workshop on Role-based access control*, pages 39–44, 2000.
- [15] J. Park and R. Sandhu. The UCON ABC Usage Control Model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.
- [16] R.S. and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, Sep 1994.
- [17] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196, London, UK, 2001. Springer-Verlag.
- [18] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: towards a unified standard. *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.
- [19] R. S. Sandhu. Role-based access control. In M. Zerkowitz, editor, *Advances in Computers*, volume 48. Academic Press, 1998.
- [20] Ravi S. Sandhu. The typed access matrix model. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 122, Washington, DC, USA, 1992. IEEE Computer Society.
- [21] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.
- [22] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM.
- [23] Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997.
- [24] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
- [25] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.

BIBLIOGRAPHY

- [26] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. To appear.
- [27] H. Wang and S.L. Osborn. Delegation in the role graph model. *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 91–100, 2006.
- [28] M.J. Yuan and T. Heute. *JBoss Seam: Simplicity and Power Beyond Java EE*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2007.
- [29] L. Zhang, G. J. Ahn, and B. T. Chu. A rule-based framework for role-based delegation and revocation. *ACM Trans. on Inf. and System Security (TISSEC)*, 6(3):404–441, 2003.
- [30] X. Zhang, S. Oh, and R. Sandhu. PBDM: a flexible delegation model in RBAC. *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 149–157, 2003.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ACL Access Control List

DAC Discretionary Access Control

MAC Mandatory Access Control

mechanism This refers to the actual implementation of an access control policy.

model A tool used for describing policies. It allows people to reason about the behaviour of the policies being described and has systematic features.

object A passive entity or active device on which access control has to be applied.

policy A policy defines what rules the computing system should use for making access control decisions.

principal An authenticated user's internal representation in the system. A user can have multiple principals, but a principal is associated with only one user.

RBAC Role-Based Access Control

resource A part of the system protected by an access control policy.

security context Authentication information and activated principals get stored in this part of an access control system.

session The current status of a user interacting with an application, for example, used to hold activated roles.

SoD Separation of Duty

subject This identifies a running process or program. Assumes identity of a principal.

user Information representing an entity external to the system, usually a human but also software agents. User information is generally encapsulated in an account or profile.