Eighth International Conference on Web Engineering

Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns

Danny Groenewegen
Software Engineering Research Group
Delft University of Technology
The Netherlands
dgroenewegen@gmail.com

Eelco Visser
Software Engineering Research Group
Delft University of Technology
The Netherlands
visser@acm.org

Abstract

In this paper, we present the extension of WebDSL, a domain-specific language for web application development, with abstractions for declarative definition of access control. The extension supports the definition of a wide range of access control policies concisely and transparently as a separate concern. In addition to regulating the access to pages and actions, access control rules are used to infer navigation options not accessible to the current user, preventing the presentation of inaccessible links. The extension is an illustration of a general approach to the design of domain-specific languages for different technical domains to support separation of concerns in application development, while preserving linguistic integration. This approach is realized by means of a transformational semantics that weaves separately defined aspects into an integrated implementation.

1 Introduction

Access control is essential for the security and integrity of interactive web applications. While a simple 'all-or-nothing' access control policy such as provided by Apache's .htaccess file is sufficient for many web sites and applications, other applications such as conference management systems, social network services, and online auctions require more sophisticated policies to regulate the access to sensitive data and operations on those data. A simple access policy can be enforced by means of a generic 'check at the gate', verifying the identity of the user. More advanced policies grant or deny access based on the identity of the user, but also the particular entry point, the subsequent actions to be invoked, *and* the data to be viewed or modified. For example, in a social network service, certain pages may be only accessible to the members of a group, which requires checking the particular combination of page, group, group membership, and user identity. Definition of such policies requires close integration with the web application, tuned to its data model and operations. Furthermore, validation and verification of a policy requires a concise high-level specification, separate from the implementation details of the rest of the application. 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 [6, 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 [24] 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, 16] and discretionary [15, 17] 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.

This situation is not unique for the domain of access control. The implementation of web applications comprises many other technical concerns, including data representation, querying, and modification, user input, data valida-

978-0-7695-3261-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICWE.2008.15

IEEE computer society

tion, user interface design, and navigation. These concerns are often addressed by separate languages. For example, in (one configuration of) the Java web programming platform we find the Java general purpose programming language, the SQL query language (or some dialect such as HQL), the JavaServer Faces (JSF) presentation language with the EL expression language for accessing data, the CSS stylesheet language, and other XML schemas for configuration such as page flow declarations.

While separation of concerns and 'choosing the right language for the job' are conceptually appealing, the amalgam of languages used in a single web application project are typically poorly integrated, with an adverse effect on productivity and software quality. For example, while Java is a statically checked language, the portions of a web application implemented in XML are outside the control of the Java compiler. Thus, the integration of XML data and Java classes is not statically verified, requiring run-time debugging and possibly errors that go undetected into production systems. The encoding of SQL queries as string literals entails that queries are only checked syntactically and semantically at run-time, and introduce the risk of injection attacks. Similarly, the use of *dependency injection* leads to leaner programs, but also shifts linking of program components from compile-time to deployment- or even run-time. Besides the loss of static verification, the languages are often redundant, i.e. need to address overlapping concerns. For example, the language of EL expressions in JSF is a poor subset of Java expressions used for accessing properties and methods of Java objects connected to a JSF page.

Generic aspect languages such as AspectJ [10] achieve separation of concerns while maintaining the benefits of linguistic integration, such as static analysis and reuse of overlapping functionality (e.g. statements and expressions). However, generic aspect languages are agnostic about particular (technical) domains such as access control, and thus require policies to be encoded programmatically, precluding benefits provided by more declarative definitions.

In previous work we have developed WebDSL, a domain-specific language for development of web applications with a rich data model [23]. 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 paper, we present the extension of WebDSL with abstractions for declarative definition of access control. 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 paper 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.

In the next section we give a brief introduction to WebDSL, illustrated with the implementation of a small wiki application. In Section 3 we introduce the access control extension of WebDSL by providing a group-based access control policy for the wiki. In Section 4 we discuss the expressivity of WebDSL access control by encoding standard policies such as mandatory, discretionary, and role-based access control. In Section 5 we present the transformational semantics for weaving access control rules into an application definition. In Section 6 related work is discussed. Section 7 contains an evaluation and possibilities for future work. Section 8 concludes this paper.

2 WebDSL

WebDSL is a domain-specific language for the implementation of dynamic web applications with a rich data model [23]. The language provides sub-languages for the specification of data models, for the definition of custom *pages* for viewing and editing objects in the data model, and as will be discussed in this paper, access control. The language *integrates* support for the various concerns of a web application in a coherent language, while catering for adequate *modularization* of web application aspects. In this section we introduce data models and page definitions using a simple *wiki* as running example (Figure 1 and 2). In the next section we then consider the extension of this example with access control.

Data Models A data model definition introduces named *entity* definitions (e.g., `Topic` and `User` in Figure 1), containing *properties* with a name and a type. Types of properties are either value types (indicated by `::`) or associations to other entities defined in the data model. Value types are basic data types such as `String` and `Date`, but also domain-specific types such as `WikiText` that carry additional functionality. Associations are *composite* (the referer owns the object, indicated by `<>`) or *referential* (the object may be shared, indicated by `->`). Associations can be to *collections* such as `Set` or `List`. The `inverse` annotation on a property declares a relation with automatic synchronization of two properties. For example, if `User` u is an element of t`.authors` of a `Topic` t, then t is an element of u`.topics`.

Page Definitions Page definitions consist of the name of the page, the names and types of the objects passed as parameters, and a presentation of the data contained in the parameter objects. For example, the `topic(t : Topic)` definition [1] creates a page showing the content of `Topic t`. WebDSL provides basic markup operators such as `section`, `header`, and `list` for defining the structure of a page. Data from the object parameters (and the objects they refer to) are injected in the page by data access operations such as `output`. Navigation is realized using the `navigate` element, which takes a page with parameters and a link text as arguments. Collections of data can be presented using the iterator construct `for`, which can filter and sort the elements of a collection.

Page definitions are a special form of *template definition*. User-defined *templates* allow the developer to define reusable chunks of WebDSL code. For example, the `main()` template [4] defines a general set-up for the page that is shared among pages of the application. A template may call other templates, e.g. `main` calls `menubar`, `body`, and `footer`. Such templates may be locally redefined in order to insert different content in a template. For example, `topic` redefines `menubar` and `body`.

In addition to *presenting* data objects, pages can also *modify* objects. For example, the content of a wiki topic can be modified with the `editTopic` page [2]. The `input(t.content)` page element declares an appropriate form input element based on the type of its argument; in this case a textarea. A data modification is finalized by means of an *action*, which can apply further modifications to the objects involved. For example, in the `saveTopic` action the user is added to the set of authors of the topic. (The notion of `securityContext` will be introduced in the next section.) The `return` statement of an action is used to realize *page flow* by specifying the page and its arguments where the browser should be directed after finishing the action.

Other features include a *module* mechanism for dividing an application into coherent, and possibly reusable units;

```
entity Topic {
  name     :: String (name)
  authors  -> Set<User>
              (inverse=User.topics)
  modified :: Date
  content  :: WikiText
}
```

```
entity User {
  username :: String (name)
  email    :: Email
  password :: Secret
  topics   -> Set<Topic>
}
```

Figure 1. Data model for wiki

```
module wiki-page imports wiki-data

define page topic(t : Topic) { [1]
  main()
  title{"Topic: " output(t.name)}
  define menubar() { topicMenu(t) signinMenu() }
  define body() {
    section{
      header { output(t.name) }
      par { output(t.content) }
      par { "Contributions by "
            for(u : User in t.authors order by t.name)
              {output(u) " "} } } } }
define page editTopic(t : Topic) { [2]
  main()
  title{"Edit topic: " output(t.name)}
  define menubar() { topicMenu(t) signinMenu() }
  define body() {
    section {
      header { "Edit Topic: " output(t.name) }
      form {
        par { input(t.name) }
        par { input(t.content) }
        par { action("Save Changes", saveTopic()) } }
      action saveTopic() {
        t.authors.add(securityContext.principal); [3]
        return topic(t); } } } }
define main() { [4]
  block("page"){
    block("menu"){ menubar() }
    block("body"){ body() }
    block("footer") { footer() } } }
define topicMenu(t : Topic) {
  menu{ menuheader{ output(t) }
    menuitem{ navigate(editTopic(t)){"Edit"} } } } [5]
define signinMenu() {
  menu{ menuheader{ navigate(signin()){"Sign in"} } } }
```

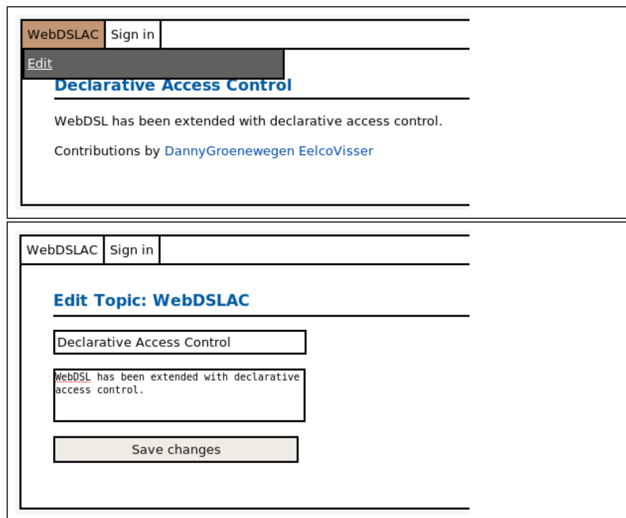

Figure 2. View and edit page for wiki topics with screenshots.

an embedded query language, which ensures that queries are checked syntactically and semantically (as opposed to encoding queries in string literals); and regular expressions for validation of value type properties.

Implementation WebDSL is implemented by a generator that translates WebDSL definitions to applications based on the Java/Seam architecture, consisting of high-level application frameworks, such as the Java Persistence API (JPA), JavaServer Faces (JSF), and the Seam web framework [23]. For each entity definition, a corresponding entity class is generated with fields, getters, and setters for the properties of the entity, annotated for object-relational mapping according to the JPA. For each page definition, a JSF XHTML page, a Seam Java bean class, and an accompanying interface are generated. The implementation of the generator is realized with the syntax definition formalism SDF [21] and the transformation language Stratego [22].

3 Access Control

Web applications that give clients creation or modification access to its data, need to trust those clients to well-behave. Trusting the general public, as is done in sites such as Wikipedia, requires either faith in human nature or a large community of moderators checking modifications. Otherwise a site runs a high risk of corrupted data and/or (scripted) spam attacks. For most applications the only solution to prevent this, is to impose an access control policy allowing only known users to access and modify data. Implementing an access control policy requires checking the permissions of the user to open a page, to perform an action, or even to see part of a page, implying checks scattered across the definition of an application. Such embedded checks make it hard to understand the access control policy they implement. We have designed and implemented an extension of WebDSL to support *separate and declarative specification of access control* for an application. In this section we illustrate the language features for declarative access control by extending the wiki example of the previous section with a variation on discretionary access control. In the next section we give a more systematic account of the expressivity of the language for encoding standard access control policies such as mandatory, discretionary and role-based access control.

Authentication The first step of any access control system is to establish the identity of the user (the *principal*) by means of an authentication procedure typically requiring the user to declare a (public) username and (secret) password. When a username and password combination corresponds to a registered user, that user is logged in, and his identity stored for the duration of the session.

```
principal is User with credentials username, password 6
```

```
session securityContext { principal -> User } 7
function authenticate(name : String, pw : Secret): Bool { 8
  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 page signin() { 9
  main()
  define body() {
    var name : User;
    var pw   : Secret;
    form{ table{
      row{"username: " input(name)}
      row{"password: " input(pw)} }
      action("Sign in", signin()) }
    action signin() {
      if (authenticate(name, pw))
        { return user(securityContext.principal); }
      else { return accessDenied(); } } } }
```

Figure 3. Principal, security context, and authentication.

Thus, the first step in setting up an access control policy is to declare the principal as a combination of an entity type to represent users, and their *credentials*, i.e. the properties of the user entity that play a role in authentication. Figure 3 shows the authentication definitions for the example wiki application. The `User` entity is declared as principal with `username` and `password` as credentials 6. Note that any entity can be used as principal; `User` is not a built-in notion. The credentials should consist of one or more properties with type `Secret` (passwords) and one or more properties that form a primary key to identify the user object.

From this declaration, a *session entity* `securityContext` is derived 7, which holds session data related to access control, in particular a reference to the principal. Session entities are attached to a client session and are accessible from any page definition. In Figure 2 we used the `securityContext` to obtain the principal and record that user as author of the topic in `editTopic` 3. Furthermore, an authentication function is derived 8, which checks the credentials against the database (using an embedded query), and upon success initializes the `securityContext`. The authentication function can be used to implement a dialog such as the `signin()` 9 template to authenticate a user, given his credentials.

Restricting Access Given the ability to authenticate a user against a database of known users, a very first basic access control policy is to distinguish anonymous visitors from authenticated users. For instance, in the wiki we could allow anyone to *view* topic pages, but only allow logged in users to *edit* the content of a topic. This policy is expressed by means of the following *access control rules*:

```
rule page topic(t : Topic) { true }
```

```
rule page editTopic(t : Topic) { loggedIn() }
```

The first rule states that the condition for accessing the `topic` page is `true`, thus access is always granted. The second rule states that the `editTopic` page can only be accessed if `loggedIn()` is `true`; the `loggedIn` *predicate* is generated by default when a principal is declared and consists of a simple `principal != null` check.

In general, an access control rule has the form `rule` r i`(`$\vec{x}$`){`e`}` with r the type of definition or resource (`page`, `template`, `function`, or `action`), i the identifier of the definition, $\vec{x}$ its parameters (a list of identifier and type pairs, just like the parameters of a page definition), and e a Boolean expression over the parameters of the rule and the session entities of the application. When the expression evaluates to `true`, access is granted, when it evaluates to `false`, access is denied. We do not use a logic with more than two values to indicate that a rule is not applicable or that an error occurred in its evaluation. In those cases a rule should just evaluate to `false` and access be denied. When access control is enabled by declaring a principal, and no rule exists for a certain resource, access to that resource is denied by default.

Representing Access Permissions Most applications require a more sophisticated access control policy than just authenticating the user. Rather, some combination of the identity of the user and aspects of the state of the application are involved to make a decision. As an example, we develop a policy in which wiki users have view or edit access to topics based on group membership. Such a policy requires the administration of the access rights of users. Instead of using a specialized, built-in data type for this administration, WebDSL employs the same data models it uses for other data of the application, ensuring a seamless integration of access control with the rest of the application.

To model the group membership policy we introduce a `UserGroup` entity 10 with a `name` and a set of `User`s as `members` (Figure 4). Next we *extend* the declaration of the entity `User` with a property `groups` 11, as the inverse of the `members` property (if u in g`.members` then g in u`.groups`). Entity extension is a modularity feature that allows properties to be declared together with the other definitions for the aspect they pertain to (akin to intertype declarations in AspectJ [10]). With this representation of groups, we can then define the notion of an access control list `ACL` 12 as an entity with a `viewers` and `editors` property, which both refer to a *set* of groups, with the intention that members of these groups have the corresponding permission. Finally, the `Topic` entity is extended 13 with an `acl` property to represent the access permissions for the object.

```
entity UserGroup { 10
  name    :: String (id, name)
  members -> Set<User>
}
extend entity User { 11
  groups -> Set<UserGroup> (inverse=UserGroup.members)
}
entity ACL { 12
  viewers -> Set<UserGroup>
  editors -> Set<UserGroup>
}
extend entity Topic { 13
  acl -> ACL
}
```

Figure 4. Representing permissions.

```
access control rules
  predicate memberOf(gs : Set<UserGroup>) { 14
    Or[g in gs | g : UserGroup in principal.groups]
  }
  rule page topic(topic : Topic) { 15
    (t.acl.viewers.length = 0) || memberOf(topic.acl.viewers)
  }
  rule page editTopic(topic : Topic) { 16
    memberOf(topic.acl.editors)
  }
```

Figure 5. Checking permissions.

Checking Access Permissions Given the encoding of access permissions we can now define the rules that declare the access to specific pages and actions (Figure 5). The `memberOf` predicate 14 tests whether the principal is member of one of a given set of groups `gs`, by checking if one of the groups of the principal occurs in `gs` (which is equivalent, through the inverse relationship between `UserGroup.members` and `User.groups`). Note that inside an access control rules section the members of the `securityContext` are directly accessible, which is why `principal` can be used here instead of `securityContext.principal`. This predicate is then used in the definition of the access control rules. The rule for the `editTopic` page 16 requires that the principal is member of one of the editors groups. The rule for the `topic` page 15 requires that the principal is member of one of the `viewers` groups, however, when no such group is registered, access is open for anyone.

Restricting Navigation The rules above forbid access to an edit *page* if the user is not a member of the editors groups. However, the menu of the view page for a topic contains a link to the edit page for that topic through the `navigate(editTopic(t)){"Edit"}` element 5 in template `topicMenu` in Figure 2. When a user without the proper permission follows this link, he will end up in the `accessDenied()` page. Following these 'dead' links can be prevented by letting the visibility of the navigate link depend on the same check as specified for the target of the navigation. Thus, only relevant navigation options are presented to the user, improving the user experience. This

behavior can be inferred because of the declarative (as opposed to programmatic) formulation of access control rules.

Administration The administration of access rights also requires a user interface. Since WebDSL employs the same data models for access rights as for any other application data, the same user interface modeling can be used to implement a user interface for access administration. For example, an `editUserGroup` page can be defined to edit the collection of members of a group and an `editPermissions` page [18] for editing a `Topic`'s `ACL`. Of course, the access to such administration pages should be controlled as well, lest the access control of regular pages becomes meaningless. If anyone can add members to a group, the rule for edit access of a topic has no value. Thus, to control the access to groups and access control lists themselves, we extend these with a `moderators` property [17] indicating who can modify these objects (Figure 6). Then access control rules can be defined [19] to restrict the access to the pages for editing permissions. A group may only be edited by the users in the `moderators` collection and an `ACL` may only be edited by members of the `acl.moderators` groups.

Policy Combination Often it is useful to equip an administrator with special rights, for instance to access any pages and perform all kinds of edits. With the rules discussed so far, this would require adding to each rule a disjunct that checks whether the principal is an administrator, which would hardwire this policy and pollute many rules. Instead, WebDSL provides the possibility of combining sets of rules into a policy. For this purpose, a set of rules can be given a name, which is used to refer to it. Unnamed rule sets are combined into a single set with the name `anonymous`. Rule sets can be combined using the `OR` and `AND` operators. For example, to allow an administrator to access all pages, a *generic* rule [20] can be used (Figure 7). A generic rule can use a wildcard for the name of the element and/or the parameters of the element. Such a rule applies to elements to which it matches. Thus a rule with signature `page *(*)` applies to all page definitions. Of course, a rule with a wildcard for the parameters cannot involve information from the parameter objects. Here, `adminGroup` [21] is an application-global variable that identifies a particular object of type `UserGroup`. The generic rule is placed inside a rule set named `admin`. The `admin` rule set is then combined with all previously defined (`anonymous`) rules using the `OR` operator [22], which entails one has access to a page when either of the rules matching the same page or action succeed.

Active Group With the `admin` policy above, a user who is administrator can always apply all actions, similar to always being logged in as super user in a Unix system. Thus, an administrator is not protected against mistakes, nor does

```
extend entity UserGroup { [17]
  moderators -> Set<User>
}
extend entity ACL {
  moderators -> Set<UserGroup>
}
define page editPermissions(t : Topic) { [18]
  main()
  define body() {
    section{
      header{"Edit Permissions of " output(t) }
      form{
        table{
          row{"Viewers: " input(t.acl.viewers)}
          row{"Editors: " input(t.acl.editors)} }
        action("Save", savePermissions()) }
      action savePermissions() {
        t.save();
        return topic(t); } } } }
access control rules [19]
  rule page editUserGroup(g : UserGroup) {
    principal in g.moderators
  }
  rule page editPermissions(t : Topic) {
    memberOf(t.acl.moderators)
  }
```

Figure 6. Administration of permissions.

```
access control rules admin
  rule page *(*) { [20]
    isAdministrator()
  }
  predicate isAdministrator() {
    adminGroup in principal.groups [21]
  }
access control policy [22]
  anonymous OR admin
```

Figure 7. Combination of rules.

```
extend entity securityContext { [23]
  activeGroups : Set<UserGroup>
}
access control rules
  predicate memberOf(gs : Set<UserGroup>) { [24]
    Or[g in gs | g : UserGroup in activeGroups] }
access control rules admin
  predicate isAdministrator(){ adminGroup in activeGroups } [25]
```

Figure 8. Active groups.

the administrator ever see the application as normal users see it. In order to enable the privileges of an administrator, or other group membership, only when needed, the policy can be refined by introducing the notion of *active*

groups (Figure 8). The `activeGroups` property that is added to the `securityContext` [23] represents the subset of the groups of the user in which he is active at the moment. By changing the `memberOf` [24] and `isAdministrator` [25] predicates to take into account only `activeGroups` instead of `principal.groups` this policy is implemented.

4 Access Control Policies

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, we discuss the three major access control paradigms and we show how these policies can be elegantly encoded in WebDSL.

Mandatory Access Control Mandatory Access Control (MAC) [18, 15] 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.

MAC policies are mainly aimed at preserving confidentiality of information contained in objects. Protection of confidentiality deals with information flow, by preventing unsafe transfer of (the contents of) objects to other security labels in the system. To protect confidentiality two properties must hold. (1) The *simple security rule*, also known as the *read-down property*, 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. (2) The *liberal *-property*, also known as the *write-up property*, 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 clearance labels. 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.

The policy in Figure 9 is an encoding of the MAC policy in WebDSL. The policy considers reading as accessing the

```
entity Label { 26
  name   :: String
  higher -> Set<Label> (inverse=Label.lower) 27
  lower  -> Set<Label> (inverse=Label.higher) 28
  predicate dominates(l : Label) { 29
    l = this || Or[l2.dominates(l) | l2 : Label in this.lower]
  }
}
extend entity User { 30
  clearance -> Label
}
extend entity Topic {
  classification -> Label
}
extend session securityContext { 31
  activeClearance -> Label
}
access control rules
  rule page topic(t : Topic) { 32
    activeClearance.dominates(t.classification)
  }
  rule page createTopic() { 33
    true
    rule action save(t: Topic) {
      t.classification.dominates(activeClearance)
    }
  }
  rule action activateClearance(c : Label){ 34
    principal.clearance.dominates(c)
  }
```

Figure 9. Mandatory access control.

`topic` view page for a `Topic` object, and writing as saving a `Topic` with the `createTopic` page, which can only be used to create a page; editing existing topics is not possible in this example. The users in the model are simply the `User` entities. To provide a distinction between user and subject, the `securityContext` is extended to represent the current subject with an active clearance label.

Clearance and classification labels are represented by the entity `Label` [26]. The partial order on `Labels` is represented by the `higher` [27] and `lower` [28] properties, which represent the direct parent and direct child `Labels` of a `Label`, respectively. The `dominates` predicate [29] defines the transitive closure of the relation. The `User` and `Topic` entities are extended with a property representing the security clearance and classification, respectively [30]. The `activeClearance` property of the `securityContext` session entity [31] represents the clearance label of the subject.

The simple security rule is now implemented by the rule for `topic` [32], which states that topics may be viewed by subjects with a clearance label dominating the classification of the topic. The liberal *-property is implemented by the rule for the `save` action of the `createTopic` page [33]: a topic can only be created if its classification (as indicated in the form of the `createTopic` page) dominates the subject's clearance label. The activation of the subject's label must also be protected to complete the implementation [34]; only labels dominated by the principal's clearance label can be used as active clearance of the subject.

Note that administration of user-label assignments and editing of labels is not part of the MAC model and we fol-

lowed this model by having predefined user-label assignments and no editing of labels.

Discretionary Access Control Discretionary Access Control (DAC) models are based on listing permissions for users and objects [15, 17]. 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.

DAC policies are often described using the Access-Matrix Model [17], 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, one needs to identify *objects*, the resources that need to be protected, *subjects*, the users or processes created by the user that need to access objects, and *permissions*, the operations that apply to 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 $[s,o]$. Since the Access-Matrix is usually sparse, it is rarely stored as an actual matrix in a system. Instead the matrix is represented by means of an *access control list* (ACL), with each object holding a list of the subjects and their permissions for that object, or as a *capabilities list*, with each subject holding a list of the objects and the permissions the subject has for those objects, or as an *authorization table*, storing permissions as triples of subject, object, and permissions.

The discussion of WebDSL access control in Section 3 already presented elements of a DAC policy. The example in Figure 10 is a variation using ownership of objects. The owners determine the configuration of the ACL for the objects they own, but can also promote other users to be able to configure the ACL, implying a one level granting of administration rights. Ownership is represented by means of an `owner` property in the `Topic` entity [36]. An ACL [35] is used to hold the access rights, this is added to the object [37], in this case the `Topic` entity. The `moderators` set of users consists of the users with rights to change the `viewers` and `editors` sets of the corresponding `Topic`. The view and edit pages are protected by rules [38] that verify that the principal is either the owner, or another user that is specifically allowed a certain type of access. The page for editing the ACL of a topic is accessible only to the owner of the topic or one of the designated moderators. Finally, the set of moderators of a `Topic` can only be changed by the owner [40] (one level granting of administration rights). This implementation is an alternative (more pure) DAC policy, without the RBAC elements that are present in the example from Section 3.

```
entity ACL { 35
  viewers    -> Set<User>
  editors    -> Set<User>
  moderators -> Set<User>
}
extend entity Topic {
  owner -> User 36
  acl   -> ACL 37
}
access control rules
  rule page topic(t : Topic) { 38
    principal = t.owner || principal in t.viewers
  }
  rule page editTopic(t : Topic) {
    principal = t.owner || principal in t.editors
  }
  rule page editTopicACL(t : Topic) { 39
    principal = t.owner || principal in t.moderators
  }
  rule page changeTopicModerators(t : Topic) { 40
    principal = t.owner
  }
```

Figure 10. Discretionary access control.

Role-Based Access Control Role-Based Access Control (RBAC) [17, 15, 19, 16, 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.

The main benefits of RBAC are: *Access control administration*: user/role assignment and role/permission assignment are separated. The administrator 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. *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, 16] proposes a family of models for RBAC. RBAC0 is the basic model, which consists of users, roles, permissions as entities. Role assignment to users and permission assignment to roles determine the configuration of RBAC0, which also provides 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). RBAC2 adds constraints to the RBAC model.

The notion of groups used in the example of Section 3 is similar to roles. They are activated by the user and carry permissions with them. The example presented in Figure 11 gives an RBAC implementation with hierarchical roles — editor as a senior role of viewer — and separation of duty constraints — administrators may not be viewers or editors. The permissions are encoded in the policy, and role/permission assignment is fixed during execution. This is a reasonable simplification that is used in many practical solutions such as Acegi [1] and Seam [24]. The data model extension to implement RBAC consists of a `Role` entity 41, a reference to a set of `Roles` in the `User` entity 44, and a reference to a set of roles in the `securityContext` 45 to represent the activated roles (roles in the session). The role hierarchy is constructed by specifying which roles are the direct junior roles in the `juniors` property 42. The `equalOrSenior` predicate 43 added to the `Role` entity verifies whether a role is equal or senior to another role. In this example application, the roles are defined statically as application variables 46. The access control rules, then, define

```
entity Role { 41
  name    :: String
  juniors -> Set<Role> 42
  predicate equalOrSenior(r : Role) { 43
    r = this
    || Or[r2.equalOrSenior(r) | r2 : Role in this.juniors]
  }
}
extend entity User { roles -> Set<Role> } 44
extend session securityContext { activeRoles -> Set<Role> }45

var admin  : Role := Role { name := "Administrator" }; 46
var editor : Role := Role { name := "Editor"
                            juniors := {viewer} };
var viewer : Role := Role { name := "Viewer" };

access control rules
  predicate isActive(r : Role) {
    Or[r2.equalOrSenior(r) | r2 : Role in activeRoles]
  }
  rule page topic(t : Topic) { isActive(viewer) } 47
  rule page editTopic(t : Topic) { isActive(editor) } 48
  rule page editRoles(u : User) { 49
    isActive(admin)
    rule action save() { 50
      !(administrator in u.roles
        && (viewer in u.roles || editor in u.roles))
    }
  }
```

Figure 11. Role-based access control.

that viewing a `Topic` requires the viewer role to be active or to be a junior of another activated role (editor) 47, and editing a `Topic` requires the editor role to be active 48. For administration purposes the page `editRoles` is created that allows editing the `roles` property of a `User`. This page requires the administrator role to be active 49. Furthermore, implementing separation of duty, the save operation has an additional check to prevent illegal changes to the user/role assignments 50.

5 Transformational Semantics

In the previous sections we have seen that access control is defined separately from the rest of a WebDSL definition. In order to enforce the rules specified in a policy, local checks are introduced into pages, templates, and actions. In this section, we give a high-level description of the weaving transformations realizing this. 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. Figure 12 defines the syntax of WebDSL access control. In this syntax other WebDSL elements start with `w-`, to show where access control is inserted. In the transformations we use $\vec{x}$ to denote a list formal parameters $x1 : S1, ..., xn : Sn$.

```
w-definition
  -> "access control rules" w-id? ac-definition*
ac-definition
  -> ac-principal | ac-rule | predicate
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 "}"
```

Figure 12. Syntax of WebDSL access control

Policy Normalization The first step in the implementation is the normalization of rule sets and the policy definition to a single set of non-overlapping rules, as defined in Figure 13. (Note that only the rules for OR are shown; the rules for AND are dual, with && instead of ||.) We say that two rules *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 (*Combining Rules*). 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.

Rule Weaving Figure 14 defines the instrumentation rules that insert checks into pages and actions. As an example we consider the following simplified page rule transformation:

```
define page p( x⃗ ) { elem* }

Page: ⇓ rule page p( x⃗ ) { e } ⇓

define page p( x⃗ ) {
  init{ if(!e) { redirect accessDenied(); } }
  elem*
}
```

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. The notation is slightly simplified here for clarity, in the actual implementation the $\vec{x}$ also matches with different names for the arguments and the inserted *e* has the correct names substituted accordingly.

The weaving of pages is more interesting if the page has other initialization statements, which is shown in the *Page* transformation in Figure 14. The other transformations shown are the following: *Action* protects the execution

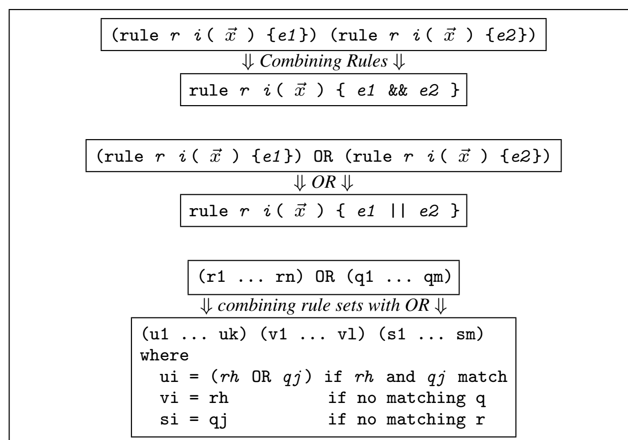

Figure 13. Policy normalization.

of an action, *Template* controls the view of the template's elements, *PageAction* shows what a nested action check in a page means, *TemplateAction* shows a nested action check in a template, *Navigation* is the semantics of inferring navigation link visibility from page rules (*e'* is *e* with formal arguments $\vec{x}$ substituted by actual arguments $\vec{y}$).

6 Related Work

In previous work we have developed MetaBorg [5], an approach for embedding domain-specific languages in general-purpose languages, and StringBorg [4], an approach for syntactically embedding query and shell languages to prevent injection attacks. This work extends the repertoire to the embedding of DSLs in DSLs with a global-to-local weaving transformation as *assimilation*. We have implemented WebDSL, in general, and the weaving transformation, in particular, with the transformation language Stratego [22].

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

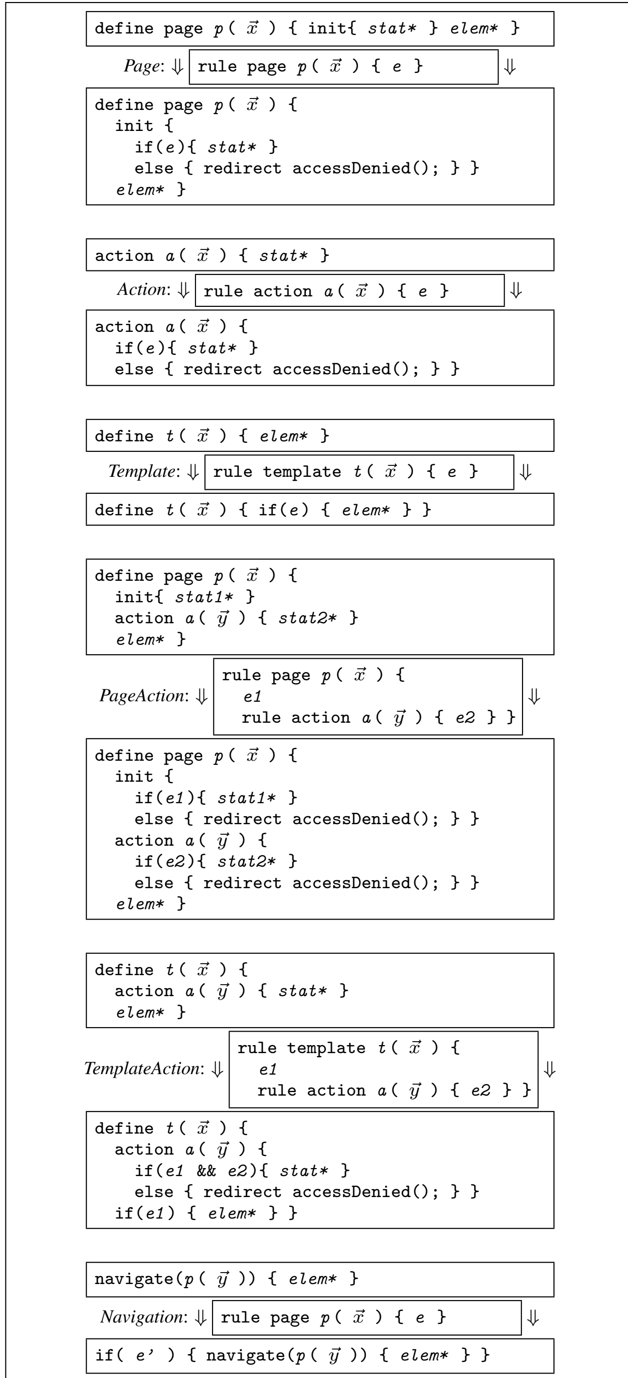

Figure 14. Weaving transformation rules.

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.

Tschantz and Krishnamurti [20] 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 occuring 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 that the 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 combination of rules.

Policy Languages The Ponder policy specification language [6] 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 dif-

ferent 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. 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 the only results for access requests. XACML 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.

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 4 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 protects generic objects by listing the permissions available for each user. This is stored in different database tables than the actual application. The combination of 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 [24] 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. A DAC policy can be implemented in a reserved function that takes the type of the object, the action, and a reference to the specific object of the access request. This is similar to the options Acegi has to offer and suffers the same drawbacks. The advanced mode consists of using JBoss Rules [24] 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).

7 Discussion

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 4 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.

The applications we have build in WebDSL so far are isolated applications. If the need arises to incorporate a WebDSL application into a larger system and access control is going to be handled externally, this would only require changing the checks in the rules to poll the external system instead of performing a local check. The separation of access control rules will help this conversion, and the semantics of access control rules as discussed in this paper are still applicable. A similar argument can be made for authentication, the user representation and authentication function can be customized to use an external authentication system.

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 concise mechanism to support a role based policy. Separation of duty checks as presented in the RBAC example in Section 4 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 4 included some facilities for delegation of access control, this could be specified in a more general way. Several models have been proposed for delegation [25, 26], 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. We are investigating the extension of WebDSL with workflow abstractions, which also provides options for exploring access control policies in that area. This can be compared to other studies regarding workflow and access control [3].

Digital rights management has been connected to access control in [14], 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.

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 extensions 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 clearly and concisely.

Acknowledgments We would like to thank William Cook for the discussions on domain-specific language design for web applications in general and access control in particular. We 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. This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

[1] B. Alex. Acegi Security, Reference Documentation 1.0.7. http://www.acegisecurity.org/guide/springsecurity.pdf, 2008.

[2] A. Anderson. XACML Profile for Role Based Access Control (RBAC). *OASIS Access Control TC Committee Draft*, 1:13, 2004.

[3] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999.

[4] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12, New York, NY, USA, October 2007. ACM.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[6] 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.

[7] M. Evered and S. Bögeholz. A case study in access control requirements for a health information system. In *ACSW Frontiers*, pages 53–61, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[8] D. Ferraiolo, D. 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. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072(327-355):110–121, 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] J. Park and R. Sandhu. The UCON ABC Usage Control Model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[15] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design on Foundations of Security Analysis and Design (FOSAD'00)*, pages 137–196, London, UK, 2001. Springer-Verlag.

[16] 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.

[17] R. Sandhu and P. Samarati. Access control: principle and practice. *Comm. Magazine, IEEE*, 32(9):40–48, 1994.

[18] R. S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[19] R. S. Sandhu. Role-based access control. In M. Zerkowitz, editor, *Advances in Computers*, volume 48. Academic Press, 1998.

[20] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM.

[21] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[22] E. 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. Spinger-Verlag, June 2004.

[23] E. 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.

[24] M. Yuan and T. Heute. *JBoss Seam: Simplicity and Power Beyond Java EE*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2007.

[25] L. Zhang, G. J. Ahn, and B. T. Chu. A rule-based framework for role-based delegation and revocation. *ACM Transactions Information and System Security*, 6(3):404–441, 2003.

[26] 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.