Fine-Grained Role- and Attribute-Based Access Control for Web Applications

Seyed Hossein Ghotbi and Bernd Fischer

University of Southampton
{shg08r,b.fischer}@ecs.soton.ac.uk
http://www.soton.ac.uk

Abstract. Web applications require an access control mechanism such as role-based access control to enforce a set of policies over their shared data. An access control model that is based on the desired security properties is thus a core security aspect, and the development of such models and their mechanisms are a main concern for secure systems development. Fine-grained access control models provide more customization possibilities and administrative power to the developers; however, in Web applications the corresponding policies are typically hand-coded without taking advantage of the data model, object types, or contextual information. This paper presents and evaluates *Φ*RBAC, a declarative, fine-grained role- and attribute-based access control model which is implemented by code generation. The generator uses a translation into logical satisfiability problems to check the *Φ*RBAC model for correctness and completeness, and against independently defined coverage criteria. If the model passes these tests, the generator then compiles it down to the existing tiers of WebDSL, a domain-specific Web programming language. We describe the test and code generation phases, and show the application of *Φ*RBAC to the development of a departmental Web site.

Key words: Fine-grained access control; RBAC; attribute constraints; access control testing; Web application security; language design

1 Introduction

Web applications, such as Facebook, are deployed on a set of servers and are easily accessible via any Web browser through an Internet connection. As the number of users of a Web application grows, its security and the privacy of the users' data become major concerns [5], and access to shared data needs to be controlled based on a set of specific policies via one of the many types of access control such as *discretionary*, *mandatory*, or *role-based access control* [24].

For reasons such as maintainability and cost effectiveness [4], role-based access control (RBAC) is the most widely used [14] access control mechanism. RBAC [8] uses the notion of *role* as the central authorization element; other components of the system such as *subjects* and *permissions* (describing the allowed operations on *objects*) are assigned to one or more roles. Over the last two decades RBAC has been extended by different types of attributes, such as temporal [2] or content-based [10] attributes, which allow more secure and flexible policies [19]. For example, with time constraints such as in TRBAC [2] the developer can limit access by the users of the system to a certain period of time such as office hours. This lowers the risk of system abuse outside office hours.

Web applications typically consist of elements that have different granularity levels and are scattered throughout the application code, which makes access control more difficult. For example, a web page can contain smaller elements such as sections or even individually controlled cells of a table. Currently, developers need to hand-code the access control elements around the objects that require access control, using languages such as Scala [17], XACML [1], or Ponder [6]. However, such approaches have three main drawbacks that complicate development of fine-grained access control and can easily lead to security holes. First, they lack the right *abstraction level* to define flexible access control models that allow the specification of different policies for different individual objects. Second, they lack a *separation of concerns* [32] between access control and application [3]. If we can develop the access control components separately, we can check for potential vulnerabilities separately and mechanically, instead of manually analyzing the access control checks in the application code, which is an error-prone and time consuming process. Third, they lack a *code generation* mechanism that can automatically translate the specified abstract access control model into corresponding access control checks and weave these into the application to enforce the model without introducing coding errors.

Although testing the access control model is essential [23, 20, 21], testing the model on its own is not enough. Any access control is defined to cover a set of objects in an application. Therefore, the testing mechanism should also take the target application, with the woven-in access control checks, into consideration. The testing phase should thus first mechanically verify the correctness and completeness of access control model itself and then validate the application code based on a set of test cases. These test cases should cover a set of objects and policy scenarios for which a correct outcome gives the developer sufficient confidence that the deployed access control model is appropriate for the given application. For example, the system should produce more restrictive test cases for a medical application than for an internet forum.

This paper describes ΦRBAC, a fine-grained RBAC with additional attribute-based constraints for the domain of Web applications. It provides a novel mechanism for declaratively defining RBAC policies and test objectives over a range of objects with different granularity levels within a single model. This model can be formally analyzed and verified. ΦRBAC is implemented on top of WebDSL [30], a domain-specific language for Web application development. ΦRBAC uses code generation techniques weave access control checks around the objects within the application code written in WebDSL. We describe the test and code generation phases, and show the application of ΦRBAC to the development of a departmental Web site.

2 Background and Related Work

2.1 Role-based Access Control with Additional Attributes

RBAC belongs to the *grouping privileges* class of access control models [24]. In this class privileges are collected based on common aspects, and then authorizations are assigned to these collections. The fundamental advantage of using a grouping privileges model is that it factors out similarities, and so handles changes better, which leads to an easier authorization management [11]. RBAC is used in many domains and there are number of languages that support RBAC [14, 1, 6]. RBAC uses the notion of *role* as the central authorization mechanism [8]. Intuitively, a role is an abstract representation of a group of subjects that are allowed to perform the same operations, on behalf of users, on the same objects. For example, in an RBAC model we can define a role `supervisor` and state that *any* user with this role can edit marks, while users with the role `student` can only read them. The other main elements of RBAC are subjects, objects, and permissions. A *subject* is the representation of an authorized user, an *object* is any accessible shared data and a *permission* refers to the set of allowed operations on objects. In the example above, the subject could actually be a session that belongs to the user after authentication and the permissions describe the allowed operations on the marks objects. Adding attributes (such as time, date, or location) to RBAC is beneficial and well studied [10, 19, 2], because it leads to more flexibility in the model and solves a set of the core RBAC shortcomings. Core RBAC has been standardized by the National Institute of Standards and Technology (NIST) [25], but despite a recent study trying to unify attribute-based access control models [19] there is no common ground yet.

Fine-Grained Access Control In the existing literature, the notion of fine-grained access control refers only to models that can control access to fine-granular objects but where the policies themselves remain coarse-grained, and thus lack flexibility. For example, a number of studies [27, 34] discuss fine-granular access control in the context of databases in terms of the table structure (i.e., columns, rows, etc.), while others [18, 26] discuss it in the context of XML and the hierarchical structure of XML documents. Our notion is related to both objects and access control, so that the access control model itself becomes more flexible and can provide a more efficient development environment.

Typically, objects are scattered throughout the application code; therefore, if the programmer writes the access control component by hand or uses access control approaches such as XACML, the access control checks will be scattered throughout the application code as well [1]. This is not suitable from many points of views. From a design point of view, it is hard to track the access rights for each object wherever it occurs within the application, and to reason conclusively about its access control checks. From an implementation point of view, coding and maintenance of the code will be time-consuming and error-prone [33]. Finally, from a testing point of view it is hard for the tester to figure out the usage coverage of hard-coded access control checks.

Testing Access Control Models Access control as a software component needs to be tested [28]. Testing needs to consider three aspects of an access control model, correctness, completeness and sufficiency. First, an access control model needs to be correct so that we can derive the required access control predicates for controlled objects. Since RBAC has a standard and therefore its semantics is well defined and understood, the *correctness* of any defined model should be checked based on the standard. Second, the *completeness* check of an access control model is essential. A complete access control model covers all possible outcomes of its defined policies with respect to the RBAC structure. For example, if we have `student` and `teacher` as two roles in an access control model and there is a static separation of duty between them, then the model only needs to cover three cases to be complete: first, `teacher` is active but not `student`, second, `teacher` is not active but `student` is and third, neither of them are active. The sufficiency of an access control depends on its target application and should therefore be defined by the developer. For example, a developer might define an access control model in a way that gives too much power to a user (the so-called the super-user problem [9]). In this case, the developer might check the application sufficiency against a set of objectives and discover this issue before application deployment. We will discuss the correctness, completeness and sufficiency checks of ΦRBAC in Section 3.2.

2.2 WebDSL

WebDSL [30] is a high-level domain-specific language (DSL) for creating dynamic Web applications [13]. It provides developers with the notion of entities for defining a data-model and enforcing data validation on those entities [12]. Listing 1.1 shows an example. The properties of an object are specified by their name and their type. Types can describe values, sets, and composite associations; in particular, the type of a property can be another entity. For example, in the data model shown in Listing 1.1 the `tutor` property is of type `Teacher`, and `marks` is a property that holds a set of `Mark` entities.

Listing 1.1. Student Entity in WebDSL

```
entity Student{
  studentID  ::  String        (name)
  courses    ->  Set<Course>
  tutor      ->  Teacher
  marks      <>  Set<Mark>     (inverse = Mark.students)
}
```

The WebDSL compiler is implemented using SDF [15] for its syntax definition and Stratego/XT [29] for its transformation rules. It consists of a number of smaller DSLs (e.g., user interface, access control) that are structured around a core layer. It uses code transformation techniques [16] to transform the WebDSL code to mainstream Web application files (HTML, JavaScript, etc.). It uses these together with other provided layout resources (image, CSS, etc.) to compile and package to a server-deployable WAR file.

Even tough WebDSL increases the level of abstraction during development, it has two main shortcomings that our work here addresses. First, it does not support fine-grained attribute-based RBAC. WebDSL has a powerful data model, and even supports the validation of input data, but its RBAC model is oriented towards the presentational elements (i.e., pages and templates), rather than the data model, and remains coarse-grained. Moreover, if the developer wants to add attributes to the RBAC model, she needs to hardcode the policy within the application code. Second, it does not check the correctness of the defined RBAC model elements nor their implementation within the application code.

3 *Φ*RBAC

*Φ*RBAC is an approach for declaratively defining and implementing a flexible, expressive, and high-level RBAC mechanism on top of WebDSL. It generates access control elements and then weaves them into the application code in order to enforce the access control on fine-grained elements of the data model, instances of the data, and template and page elements. Moreover, it provides a testing mechanism to check the correctness of the model itself and with regard to its application. In this section we introduce the language by means of an example.

3.1 Access Control Model

As Listing 1.2 shows, a *Φ*RBAC model consists of three main sections: basic *RBAC* elements (lines 2-5), *policy cases* (lines 7-11) and *coverage* (lines 14-18).

Listing 1.2. *Φ*RBAC Example

```
1  PhiRBAC{
2   roles{teacher(10),admin(1),manager(1),advisor(10),student(*)}
3   hierarchy{advisor -> teacher}
4   ssod{(teacher,admin,advisor,manager) <-> student}
5   dsod{(and(advisor,teacher),admin)     <-> manager}
6
7   objects{G(roleAssignment),XML(address),Person.password,P(marks)}
8   policies{teacher,student,admin,advisor,Self.username == "faz",
9           This.User.location != "New York",Sys.Time(>=9:00,<=17:00)}
10  cases{ (+,-,-,?,+,?,-) -> ([r,u],[r],[s],[r,u]),
11         (-,-,+,?,-,+,+) -> ([r,u],[r],[s],[i]) }
12
13  coverage {
14   objects{P(root),student.marks}
15   policies{admin,teacher}
16   cases { (+,?) -> ([r,100],[i]), (-,+) -> ([i],[u,(>80,<=100)])}
17  }
18}
```

Basic RBAC Elements At the core of an *Φ*RBAC model are the basic RBAC elements. The developer first defines roles and their cardinalities (cf. line 2), which specify the maximum number of subjects that may acquire the respective roles at any given time. The developer can also define an optional role hierarchy. In the example, the `advisor` role is defined as a specialization of `teacher` (cf. line

3). In addition, the developer can define optional *separation of duty* (SOD) constraints [25] (cf. lines 4 and 5). Static SOD (SSOD) constraints affect the role *assignment* (e.g., the roles `admin` and `student` can never be assigned to the same subject) while dynamic SOD (DSOD) constraints affect the role *activation*. For example, the DSOD constraint in Listing 1.2 states that a subject cannot activate the `manager` role together with either `admin` or both `advisor` and `teacher` roles. We then use a matrix-structure to specify the actual access control policy as well as the test case coverage. The matrix' rows and columns labels are given as the set of controlled objects and the different policy terms, while the entries of the matrix are given line-by-line as policy cases. These show the relation between the policy combinations and allowed operations on the respective objects.

Controlled Objects *Φ*RBAC supports access control of objects with different types and granularity levels. We can divide them into *data model*, *page*, and *template* elements. The application's data structure is defined in a data model, which is then translated into a database type, such as tables in a relational database. The main benefit of using the data model as a part of controlled objects is that we can define access control on the data model elements *without* considering where or by whom they are used within the application code. *Φ*RBAC thus allow the data model elements as part of its controlled objects. It supports both coarse-grained elements such as the `student` entity shown in Listing 1.2 or more fine-grained components such as `speaker` property of an entity `Seminar`. It is important to note that *Φ*RBAC consequently supports relations, such as inheritance, between data-model components as well. For example, the type of the `speaker` property can be the `Person` entity. If this entity is access-controlled, then *Φ*RBAC automatically adds all the access control predicates from the `Person` entity to `speaker`'s predicate. However, the different properties and entities to be joined may have conflicting access control predicates, which can make a controlled object inaccessible. Such conflicts are checked during *Φ*RBAC's testing phase.

Since we do not want to force the developer to use *Φ*RBAC and the existing WebDSL access control to declare two different types of access control model, the coarse-grained components (i.e., pages and templates) are also supported within our model as controlled objects. Moreover, we support more fine-grained components of pages and templates. For page elements the developer can use *G(GroupNames)* to define a set of group names and *B(BlockNames)* to define the block names which are used by an external CSS style. In WebDSL we can use XML hierarchies within the template code, and the developer can use *XML(NodeNames)* to declare a set of XML node names as controlled objects.

Policy Terms A policy term is an atomic access control check which can be used as building block in more complex policies. In *Φ*RBAC there are two types of policy terms. First, the developer can use a subset of the roles that were defined. Second, *user*, *data*, or *system* attributes can also be used as policy terms. In our example, the policy term `Self.username == "faz"` defines an access control check based on the current user (which is represented in *Φ*RBAC as `Self`) with the `username` property *faz*. The policy term `This.User.location != "New York"` defines an access control check based on a data attribute: for every object (represented by `This`) of the given type `User` that is used in the application, the property `location` is checked against the given value (i.e., *New York*). *Φ*RBAC allows the usual comparison operators, including a range restriction. For example in 1.2 the last policy (cf. line 9), creates an access control check for the system's office hours. The actual policy is then defined case-by-case, dependent on the logical status of the policy terms. The logical status is either *activated* or *true* (represented as `+` in *Φ*RBAC), *not activated* or *false* (`-`) or *don't care* (`?`). Note that *don't care* is not required but simplifies the specification of complex policies.

Policy Cases As shown in Listing 1.2 (see lines 10 and 11), the developer can specify an arbitrary number of cases. Each case defines a logical combination of policy terms for creating an access control predicate. There are five different operations on the controlled objects. The developer can use `c` to denote that users with the appropriate roles (or satisfied attribute checks) are allowed to create an instance of the controlled objects or a set of objects that are embedded within the controlled objects. For example, if the controlled object is an entity, this case controls the create operations of this entity throughout the application; if the controlled object is a page, we look at the embedded objects within the page, and see if there is any create operation related to them. The developer can use `r` to allow read operations on the controlled object itself or its embedded objects (i.e., properties as sub-elements). Similarly, to allow update or delete operations on the controlled object itself or its embedded objects, `u` and `d` are used. `s` ("secret") can be used to hide the content of the object itself or its embedded objects. For example, if the controlled object is `User.username` then its instance will be hidden to the user, regardless of any other specified operations (i.e., create, read, update, delete). Finally, `i` ("ignore") can be used for defining a policy that does not effect the predicates on the controlled objects.

Note that all of these operations are guarded by the defined access control predicates. For example, Listing 1.2 line 10 shows that outside the office hours, when the authorized user with the username `faz` has the role `teacher` but not `student` or `admin`, then she cannot see any `Person`'s `password`.

Coverage Cases This part of the model (see lines 13-17 in Listing 1.2) helps the developer to define a set of independent cross checks on the *Φ*RBAC model and thus get assurance about the functional coverage of access control predicates over the controlled objects. In particular, we allow the developer to specify for each combination of policy terms to which extend the occurrences of an object within the target application are controlled. This can be seen as a summary that is independent of the actual access control mechanism. We allow the developer to define the coverage cases by hand because *only* the developer knows about the context of the target application, its security goals, and in what granularity level both defined *Φ*RBAC and the target application need to be checked.

The developer defines a number of cases, which each check the relative coverage of a set of controlled objects and their related operations for a combination of logical states (similar to Section 3.1). For example, in Listing 1.2 the first case in line 16 states a user with the activated role `admin` must have *read* access to all the controlled objects defined in the `root` page. In other words, all predicates that are derived from the policy cases (see line 11 in Listing 1.2) and will be woven around the objects within the `root` page, must be true for a user with the role `admin` activated. If we for example assume that the controlled object `user.password` is defined in the `root` page; then our first coverage case fails: based on the second defined policy case, a user with the activated role `admin` cannot see the instances of `user.password`. The coverage cases help the developer to check the defined *Φ*RBAC model, based on a different view, with respect to the target application. For example, in Listing 1.2, the policy cases do not directly cover the controlled object `student.marks`. However, in the second coverage case, we check its coverage range based on a case where the user has an activated role `teacher`.

3.2 Code Generation from *Φ*RBAC Models

The *Φ*RBAC code generation process is divided into a *testing* and a *transformation* phase (see Figure 1). The aim of the testing phase is to validate and verify the access control model itself and its integration into the target application. As the *Φ*RBAC model is defined separately from the application code, the aim of the transformation phase is to first generate the access control elements (e.g., data model, access control predicates, etc.) and then to weave them into the target application code.

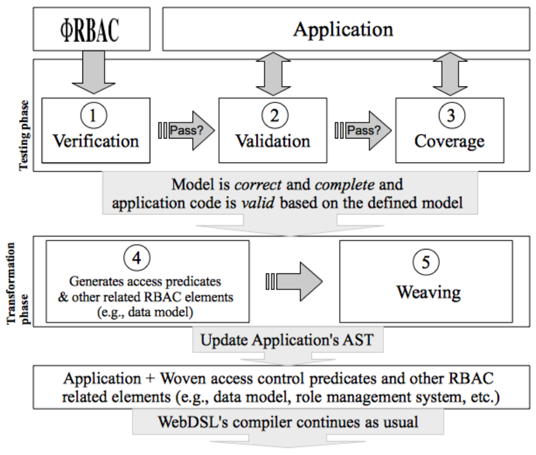
ΦRBAC
Application
Testing phase
1 Verification
Pass?
2 Validation
Pass?
3 Coverage
Model is *correct* and *complete* and application code is *valid* based on the defined model
Transformation phase
4 Generates access predicates & other related RBAC elements (e.g., data model)
5 Weaving
Update Application's AST
Application + Woven access control predicates and other RBAC related elements (e.g., data model, role management system, etc.)
WebDSL's compiler continues as usual

Fig. 1. An overview of the *Φ*RBAC generation pipeline

Testing Phase A number of studies [23, 20, 21] highlight the fact that developing an access control mechanism is error-prone and the result therefore needs to be tested. Unlike the prior approaches, we emphasize the fact that correctness and completeness of the access control model on its own is *not* enough and the target application must be considered as well, based on the defined access control model. The access control predicates are derived from the access control model and need to be implemented (in our case generated) and inserted around the desired objects. Even a partial failure of doing so will result in application code that is compilable but has a number of security holes that need to be handled *after* the application deployment phase. This leads to high testing and maintenance costs after the deployment of the application. It is ideal to give a full guarantee to the developer for the defined access control model and its target Web application before deployment phase. The testing phase consists of three consecutive white-box testing steps (see Figure 1). Failure of each step will terminate the rest of the compilation and its related error messages will be given. In the first step, the defined ΦRBAC model is verified using model checking. Second, the application code is validated with respect to the defined ΦRBAC model. Third, the coverage is checked against the defined objectives.

Model Verification This step mechanically verifies the correctness and completeness of an ΦRBAC model using SMT solver, *Z3* [7]. Here, we first verify the correctness of the defined basic RBAC elements and of each individual case defined in the policy and coverage cases with respect to the defined attributes. Second, we check the completeness of the policy and coverage cases. Since Z3 takes a representation of the model in first-order logic (FOL) and decides its satisfiability, our verification approach is to generate a number of FOL formulas from the given ΦRBAC model, and let Z3 check them individually. We then mechanically analyze Z3's output results to come to a conclusion about the correctness and completeness of the original model.

The basic RBAC elements can already create conflicts in the model. For instance, if a role `supervisor` inherits from a role `teacher` but there is also a (dynamic or static) SOD relation between them, then this specification creates a conflict and consequently an error in the model, because these two roles must be activated (due to the inheritance relation) and deactivated (due to the SOD) at the same time. To check the correctness of the basic RBAC elements, we first mechanically check if there are any undefined roles in inheritence, SSOD, and DSOD relations. Second, we check for possible conflicts between the hierarchy and SSOD (respectively DSOD) constraints by generating two FOL formulas to check with Z3. If the result is unsatisfiable (UNSAT), then there is an error in the defined basic RBAC structure. However if all the results are satisfiable (SAT) then the structure of basic RBAC elements is correct and we consequently go to the next step to check the correctness and completeness of the individually defined policy and coverage cases.

The defined cases can cause three types of errors that need to be checked:

– *Incorrect case*: The policy terms and the policy signs specified in each case can be inconsistent with the other RBAC elements. For example, if there is a SSOD relation between the `teacher` and `student` roles, a case should not define a predicate in which both roles are active (i.e., have a "+" as policy sign), because the case can then never apply. is an error in the model if each has + for their policy sign. Similarly, we have an inconsistency if there is an inheritance relation between a parent and a child role, and a case states the parent role is not active but the child role is active.

– *Overlap*: If two cases overlap, then the respective access control predicatas can be true at the same time; if the specified access rights are then different, the model is inconsistent. An overlap between two cases can be detected by comparing the corresponding pairs of policy signs: if the signs for at least one policy term are complementary (i.e., one is active ("+") while the other is not ("-")), then there can be no overlap.
– *Incompleteness*: If the defined policy or coverage cases do not fully cover all the possible cases, then we have *incompleteness.*

The correctness of each case is checked in two steps. First, we need to check the defined `Time` and `Date` attributes and their intervals (if any). For example, we have an error, if the policy terms `Sys.Time(>=7:00,<=9:00)` and `Sys.Time(>=6:00,<=6:30)` are both activated in a single case, because this policy can never be true (i.e., the time can not be between 6 and 6:30am *and* between 7 to 9 am at the same time). Such error are checked during compile time by a set of custom Stratego strategies. In the second step we check the basic RBAC elements by generating a FOL formula from these elements, using the truth values corresponding to the policy signs for each policy term. Then, Z3 is called to check the satisfiability of the formula; a SAT result means that we have a correct case, and UNSAT means that we have an error. For example, to check the correctness of the RBAC elements defined in Listing 1.2 Line 11, in addition to the defined RBAC structure (Listing 1.2 lines 2-5), we derive a FOL formula that states `admin` must be *true* and `teacher` and `student` must both be *false*. In this case, Z3 gives us a SAT result, which means that there is no conflict in the defined RBAC model.

For overlap checks, we pair any two cases and generate a FOL formula in which there is a conjunction between these two cases and their sub-elements. Then we check each pair for satisfiability; (UN)SAT means that the two cases are (not) overlapping. Then, the overlapping check is repeated until all combinations are covered. For example for a policy set {`teacher`, `student`}, if we have two cases (`+`,`-`) and (`+`,?), then their FOL formula will be `(teacher && not student) && (teacher)`, in which the SMT solver will give a SAT message that results in an error message because these two cases are overlapping. In case of *incompleteness* checks, we disjunctively link the negation of *all* of the cases and conjunction with the policy signs of each negated case. We then call Z3 to check the model for satisfiability. If the result is SAT, then there is a missing case and Z3 gives a counterexample for it. Since this produces the missing cases one-by-one, we need to respectively update and re-check the model, until Z3 finds no more missing cases. All steps mentioned above happen during compile-time. Since the developer does not know about Z3 and its results, we need to interpret these results for the user in terms of the *Φ*RBAC elements. During the last step, we parse the model checking results (UNSAT, SAT) and by retrieving its representation elements in the abstract syntax tree (AST) we give the error during the compilation based on the *Φ*RBAC elements.

Web Application Validation In our approach, the access control predicates are woven into the application code around the controlled objects. The controlled objects and consequently their predicates may be nested within each other and can so create a set of conflicts. For example, in Listing 3, we have two different controlled objects, in which the instances of all users' `username` are embedded within the sub-element of the page `UserList`. We have `P1` that protects `UserList` and `P2` that protects the instances of users' `username`. Moreover, `P1` indirectly protects `P2` as `P2` is nested within `P1`. Let us assume that `P1` and `P2` can conflict, e.g., `P1` is true for users with the role `teacher` activated and `P2` is true for users with the role `admin` activated but in the access control model there is also an `SSOD` relation between role `teacher` and `admin`. It is clear that users with the role `admin` activated can *never* access the instances of users' `username`, even though they have a right to do so. We call these situations *dead authorization code* and the following steps are used for finding such situations.

Listing 1.3. Nested controlled objects and their related predicates may create a set of conflicts.

```
if(P1){ group("UserList") {
  if(P2){ for(u: User){
          output(u.username) //could be unreachable
       }}
}
```

- *Sorting and Pairing*: First, we sort all policy cases based on the controlled objects, their related operations and predicates. Then, for each possible pair of objects, we create a list that is the union of all related predicates for that pair.
- *Potential Conflicts*: We check for conflicts between the predicates of each pair with respect to the defined RBAC structure. For this reason, for each pair, we transform their predicates and the defined RBAC structure into a FOL formula and check its satisfiability by using Z3; in case of *UNSAT*, we have a conflict.
- *Conflict Detection*: Now, we have a list of pairs in which the predicates create a conflict. We finally check whether the paired objects are embedded within each other in the application, and if so, we have found an error.

Coverage The aim of this step is to check the required access control coverage based on the defined policy and coverage cases, and to provide feedback to the developer about the potential shortcomings of the defined *Φ*RBAC model. A coverage percentage shows what percentage of an object's occurrences in the application is protected *directly* or *indirectly* by the derived access control predicates that are defined in the policy cases. For each controlled object used in the coverage cases, the coverage percentage is calculated. The following three steps sketch the calculation of the coverage percentage for each controlled object:

- *Sorting and pairing*: We sort *both* coverage and policy cases into two lists. We then combine each coverage case with all the policy cases.
- *Finding related cases and partial coverage*: We then need to find all the related policy cases for each coverage case based on the access control predicate. For this, we transform each pair of cases into a FOL formula such that the policy case's predicate is used as it is but we transform the negation of the coverage predicates. Then we call Z3 to check the *satisfiability* of the formula. If it is SAT, we omit the paired cases from the coverage computation, as they are not related; however in case of UNSAT, the predicates are related and we use the corresponding object and operation to calculate the coverage of the object based on that particular related predicate.
- *Overall coverage*: We continually repeat the last step to find out all the *direct* and *indirect* coverage of each *(object,operation)* pair based on the defined policy cases. Then, we divide the total value of the computed coverage by the total number of the occurrences for the object throughout the application.

If the computed coverage is outside the specified range we give an error that describes the *Φ*RBAC elements that fail the coverage check, and terminate the compilation. The developer can then fix the coverage errors based on the defined *Φ*RBAC model and/or the target application.

Transformation Phase As Figure 1 shows, the transformation phase is divided into *generating* the required elements and then *weaving* them throughout the Web application code. These elements are related to the RBAC and access control predicates of the system that are defined in the *Φ*RBAC model.

RBAC Generation First, elements generated from the *Φ*RBAC model have to be added to the Web application's data model, in order to provide the data manipulation mechanisms for roles and their associated activities, such as maintaining list of assigned roles for each user. Second, these generated elements have to provide a *role management mechanism* for the authenticated users of the application. This mechanism consists of the *role assignment* and *activation* modules that are based on the overall RBAC structure defined within *Φ*RBAC model.

To extend the Web application's data model, we need to find the entity that represents the *users* of the system. In WebDSL, the developer uses the notion of *principal* to define the users' authentication credentials. For example, in Listing 1.4, the authentication is based on the username and password properties of the `Person` entity which represents the user of the system. We also use this entity type to represent that users have a given role (Listing 1.5, line 4) and extend it to store a set of assigned roles for each user (line 12). In addition, the session element must be extended to hold the activated roles for each user. In this we extend the data model of the application by generating the role entity (lines 2-9), extending the `Person` entity (lines 10-13); and extending the Web application session (lines 15-17).

Listing 1.4. WebDSL authentication credentials

```
principal is Person with credentials username, password
```

Listing 1.5. Generated data model elements

```
//Generated Role entity
entity Role {
  name          :: String (name)
  users         -> Set<Person>
  inheritency   -> Set<Role> (optional)
  ssod          -> Set<Role> (optional)
  dsod          -> Set<Role> (optional)
  cardinality   -> Int
}
//Extending 'Person' entity for role assignment
extend entity Person{
  assignedRoles -> Set<Role> (inverse=Role.users)
}
//Extending session for activated roles
extend session securityContext{
  activatedRoles -> Set<Role>
}
```

We already checked the correctness of the RBAC structure defined within the ΦRBAC model (see 3.2) and as shown in the generated role entity, we store each role's characteristics (e.g., SSOD) for the RBAC management component. The SSOD relations between the roles is used in the *role assignment* component which during the run-time of the system must not allow the administrator to assign conflicted roles to any users of the system. The DSOD relation between roles is used in the *role activation* module of the system, to ensure that two roles in the DSOD relation cannot be activated in any user's session. The *inheritance relation* between roles is used for both role assignment and role activation modules. These relations must be considered for the overall structure of the defined RBAC; in particular, we need to consider more than the directly defined relations for each role. For example, if a role `advisor` inherits from the role `teacher`, and `teacher` has an SSOD relation to `manager`, the roles `advisor` and `manager` can *never* be assigned to one user, even if the defined model did not explicitly state any SSOD relation between `advisor` and `manager`. In order to get all direct and indirect relations of each role, we translate the RBAC structure into a FOL formula, where we give a *true* value to the role whose relations, we want to check and use the SMT solver to get a counterexample in which the related roles are either true (due to inheritance relation) or false (due to SSOD or DSOD).

Predicate Generation At this stage, we know that all specified cases are non-overlapping and correct. Each case then represents a predicate that is used to protect the controlled objects and their related operations. Before starting to generate the access control predicates, we first sort the cases based on controlled objects and operations, by joining their predicates where there is a same operation on the controlled object. For example, in Listing 1.2, for the controlled object `Person.password` both defined cases result in a `secret` operation. Therefore, the access control predicate that protects the instances of `Person.password` is equal to: `(teacher && not student && not admin) || (not teacher && not student && admin)`. These sorted cases will be parsed into an AST which is used by the predicate generator to generate a set of predicates that can be woven around the controlled objects in the Web application's AST.

Weaving stage Weaving is the last step in the *Φ*RBAC transformation phase. In this step, we first get the result of the RBAC and predicate generators. For the RBAC generator, the result is an AST that represents a number of modules that hold the generated data model and RBAC management component of the system. We weave these modules into the Web application's AST and we add a navigator to the authentication code to redirect the user to the role management component after successful authentication. Any user has access to their *role activation* component, however the *role assignment* component is protected, based on the access rights that are defined in *Φ*RBAC model (as shown in Listing 1.2). We then repeatedly traverse the Web application's AST, and iteratively weave in the generated access control predicates around any occurences of the controlled objects. In terms of predicates, the generated AST holds all the predicates sorted based on operations on the controlled objects. We finally pass the updated AST to the next step within the WebDSL compiler.

4 Case Study

The aim of this section is to show the benefits and limitations of the *Φ*RBAC modeling language and its code generation mechanism, based on the evaluation of a case study. The main objective of the evaluation is to check the efficacy of *Φ*RBAC during the *development phase* of a target application with a reasonably large data model, based on a rich set of policies. We chose a departmental Web site as a target application. Moreover, the goal of the evaluation is to derive a set of findings that can be used to improve *any* RBAC-based access control model, including *Φ*RBAC, that is intended to be used in the Web application domain.

We implemented our case study using WebDSL for the Web elements and *Φ*RBAC for the access control elements of the application. This case study is created and deployed for a language research group to cover their internal (e.g., organization of vivas) and external (e.g., publications) needs and to provide a fine-grained access control over the objects.

4.1 Web Application Description

In this case study, the Web application consists of three main elements, data model, pages, and access control. We divided the data model elements into two categories; users and activities. Users' entities belong to different types of users in the system such as academics or visitors. The second set of entities are related to the available activities such as adding an interest. The access control data model is generated at compile time. The size of the data model is quit large. We have nine different entities for the different types of users (e.g., `academic`, `student`) and 13 entities that cover the objects involved in activities (e.g., `publications`). Overall, we have 189 properties that are associated with 22 entities. These are the fine-grained objects that are used throughout the application code for a number of times. We divided the pages based on the different types of users and activities, regardless of the operations used for the system objects. Hence, there is only one page for each type of data and in that page all the available operations exist, but each part of the page will be access-controlled based on the policies defined in the ΦRBAC model. The access control elements for this case study are derived from the needs of the different types of users in a research group. For example, an `academic` can be a `supervisor` of a `PhD student`; however, she cannot be an examiner of a `PhD student` who she is supervising.

4.2 Evaluation and Findings

To evaluate the ΦRBAC model and mechanism, we looked at three aspects: modeling, testing, and transformation phases. The errors in the model were divided into attribute, RBAC, and the application errors, respectively. These errors were discovered before the transformation phase and consequently application deployment. The transformation strategies that were used in testing and transformation phase were tested, in a white-box manner, during their development. The woven AST was inspected manually to make sure all objects were correctly covered by the corresponding access control predicates, and conversely, that no unguarded objects were accidently covered directly or indirectly by any predicates.

The benefits are divided into development efficiency on one side, and correctness and completeness of the model and target application, respectively, before its deployment on the other side. During our case study, the ΦRBAC model was developed separately from the application code. So, in case of errors the developer did not need to search through the access control definitions scattered through the application code. Moreover, the ΦRBAC is developed at the right abstraction level: the developer did not need to use any specific object or agent-oriented terminology to define the access control components and just used the access control concepts such as roles, as they are . ΦRBAC is also a cost effective solution. In our case study the compilation time of our access control model was just 4.5 seconds on a machine with 4GB RAM and 2.3GHz CPU, to cover instances of 189 unique objects throughout the application. The correctness and completeness checks on the ΦRBAC model give an insurance to the developer about the access control of the system, so any security failure of the system during its run-time is not related to its access control element but to the other security elements of the system such as data encryption.

ΦRBAC's weakness originat in the RBAC approach itself; in particular, RBAC does not support an ownership notion. For instance, if in a research group we have a policy that states that the supervisor can edit their students' travel allowance, then any user with the role supervisor can edit the travel allowance of any student in the group regardless of who is the supervisor of those students. In order to overcome this flaw, the developer needs to introduce a number of unnecessary roles such as `supervisorOfStudentA` to enforce the mentioned policy. So ΦRBAC would be more efficient if the developer could use the ownership notion as a policy term.

5 Conclusion and Future Work

This paper introduced *Φ*RBAC, a declarative and fine-grained role- and attribute-based access control model for the Web application domain that enforces separation of concerns between application and access control model at the right abstraction level. *Φ*RBAC is defined and implemented as an extension to a domain-specific language, WebDSL, and its compiler. Its architecture is divided into a testing phase and a subsequent transformation phase. The testing phase uses a fast mechanism to check the correctness and completeness of the model and the application via code transformation and model-checking techniques. We showed how dead authorization code could occur in a fine-grained access control model, and how we check for this before the generation phase. We evaluated the approach and its mechanism based on an application example. The example demonstrated the efficacy and benefits of *Φ*RBAC in terms of defining a fine-grained access control model and checking correctness, completeness and sufficiency. Furthermore, it showed the applicability of *Φ*RBAC model for large data models based on a rich set of policies.

For future work we like to introduce the notion of ownership [22], as a policy term, to improve the *Φ*RBAC model and its mechanism. Also, we plan to integrate other well-known access control models, to achieve access control integration for Web applications that are constructed from mixed sources and require different access control models for different parts of the application. Moreover, in terms of the *Φ*RBAC architecture, we like to explore the possibility of generating our access control predicates on top of the database tier so that the application can retrieve access control settings from the database at run-time and take advantage of the database tier's security options. Furthermore, we will introduce authorization management systems for defined access control models within an application.

References

[1] Abi Haidar, D., Cuppens-Boulahia, N., Cuppens, F., and Debar, H. (2006). An extended RBAC profile of XACML. *SWS'06*, pp. 13–22, ACM.

[2] Bertino, E., Bonatti, P., and Ferrar E. (2001). TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.'01*, pp. 191–233.

[3] Chen, K. and Huang, C.-M. (2005). A practical aspect framework for enforcing fine-grained access control in web applications. *ISPEC'05*, LNCS 3439, pp. 156–167.

[4] Connor, A. and Loomis, R. (2010). Economic analysis of role-based access control. Technical report, National Institute of Standards and Technology.

[5] Dalai, A. K. and Jena, S. K. (2011). Evaluation of web application security risks and secure design patterns. *CCS'11*, pp. 565–568, ACM.

[6] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The Ponder policy specification language. *POLICY'01*, LNCS 1995, pp. 18–38.

[7] de Moura, L. M. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. *TACAS'08*, LNCS 5195, pp. 337–340.

[8] Ferraiolo, D. and Kuhn, R. (1992). Role-Based Access Control. *NIST-NCSC'92*, pp. 554–563.

[9] Ferraiolo, D. F., Barkley, J. F., and Kuhn, D. R. (1999). A role-based access control model and reference implementation within a corporate intranet. *ISS'09*, pp. 34–64, ACM.

[10] Giuri, L., and Iglio, P. (1997). Role templates for content-based access control. *Workshop RBAC'97*, pp. 153-159.

[11] Gorodetski, V. I., Skormin, V. A., and Popyack, L. J., editors (2001). *Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security*, LNCS 2052.

[12] Groenewegen, D. and Visser, E. (2009). Integration of data validation and user interface concerns in a DSL for web applications. *SLE'09*, LNCS 5969, pp. 164-173.

[13] Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. (2008). WebDSL: A domain-specific language for dynamic web applications. *OOPSLA'08*, pp. 779–780. ACM.

[14] Groenewegen, D. M. and Visser, E. (2008). Declarative access control for WebDSL: Combining language integration and separation of concerns. *ICWE'08*, pp. 175–188. IEEE.

[15] Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75.

[16] Hemel, Z., Kats, L. C. L., Groenewegen, D. M., and Visser, E. (2010). Code generation by model transformation: a case study in transformation modularity. *Software and System Modeling*, 9(3):375–402.

[17] Hortsmann, C. (2012). *Scala for the Impatient.* Addison-Wesley Professional.

[18] Hsieh, G., Foster, K., Emamali, G., Patrick, G., and Marvel, L. M. (2009). Using XACML for embedded and fine-grained access control policy. *ARES'09*, pp. 462–468. IEEE.

[19] Jin, X., Krishnan, R., and Sandhu R. (2012). A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. *DBSec'12*, pp. 41-55.

[20] Martin, E., Xie, T., and Yu, T. (2006). Defining and measuring policy coverage in testing access control policies. *ICICS'06*, LNCS 4307, pp. 139–158,

[21] Masood, A., Bhatti, R., Ghafoor, A., and Mathur, A. P. (2009). Scalable and effective test generation for role-based access control systems. *Software Eng.*, 35(5):654–668, IEEE.

[22] McCollum, C., Messing, J., and Notargiacomo, L. (1990). Beyond the Pale of MAC and DAC Defining new forms of access control. *RSP'90*, pp. 190 –200, IEEE.

[23] Montrieux, L., Wermelinger, M., and Yu, Y. (2011). Tool support for UML-based specification and verification of role-based access control properties. *ESEC'11*, pp. 456–459. ACM.

[24] Samarati, P. and di Vimercati, S. D. C. (2000). Access control: Policies, models, and mechanisms. *FSAD'01*, LNCS 2171, pp. 137–196.

[25] Sandhu, R., Ferraiolo, D., and Kuhn, R. (2000). The NIST Model for Role-Based Access Control: Towards a Unified Standard. *Workshop on RBAC'00*, pp. 47–63, ACM.

[26] Steele, R. and Min, K. (2010). Healthpass: Fine-grained access control to portable personal health records. *AINA'10*, pp. 1012–1019, IEEE.

[27] Sujansky, W. V., Faus, S. A., Stone, E., and Brennan, P. F. (2010). A method to implement fine-grained access control for personal health records through

standard relational database queries. *Journal of Biomedical Informatics 5-Supplement-1*, pp. S46–S50.

[28] Tondel, I., Jaatun, M., and Jensen, J. (2008). Learning from software security testing. *ICSTW'08*, pp. 286 –294. IEEE.

[29] Visser, E. (2003). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. *Domain-Specific Program Generation*, LNCS 3016, pp. 216–238.

[30] Visser, E. (2007). WebDSL: A case study in domain-specific language engineering. *GTTSE'07*, LNCS 5235, pp. 291–373.

[31] Win, B. D., Piessens, F., Joosen, W., De, B., Frank, W., Joosen, P. W., and Verhanneman, T. (2002). On the importance of the separation-of-concerns principle in secure software engineering. Workshop AEPSSD'02.

[32] Wurster, G. and Van Oorschot, P. C. (2009). The developer is the enemy. *NSP'08*, pp. 89–97, ACM.

[33] Zhu, H. and Lu, K. (2007). Fine-grained access control for database management systems. *BNCOD'07*, LNCS 4587, pp. 215–223.