A Declarative Fine-grained Role-based Access Control Model and Mechanism for the Web Application Domain

Seyed Hossein Ghotbi and Bernd Fischer
University of Southampton, Southampton, U.K.

Keywords: Fine-grained Role-based Access Control, Testing Access Control Model and Mechanism, Web Applications, Domain-specific Language.

Abstract: Access control policies such as role-based access control (RBAC) enforce desirable security properties, in particular for Web-based applications with many different users. A fine-grained RBAC model gives the developers of such systems more customization and administrative power to control access to fine-granular elements such as individual cells of a table. However, the definition and deployment of such policies is not straightforward, and in many Web applications, they are hand-coded in the database or scattered throughout the application's implementation, without taking advantage of underlying central elements, such as the data model or object types. This paper presents ΦRBAC, a fine-grained RBAC model for the Web application domain. ΦRBAC achieves separation of concerns for enforcing access to a range of objects with mixed-granularity levels. Moreover, it provides a unique testing mechanism that gives a guarantee to the developer about the correctness, completeness, and sufficiency of the defined ΦRBAC model, both internally and in the context of its target application. We use code generation techniques to compile the specification of a ΦRBAC model down to the existing tiers of an existing domain-specific Web programming language, WebDSL. We show the benefits of ΦRBAC on the development of a departmental Web site.

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. The availability of a wide range of server-based deployment mechanisms, such as Google Engine (Sanderson, 2009), make them a popular and suitable development choice for different domains such as e-business or social networking. However, as the number of users of a Web application grows, its security and the privacy of the users' data become major concerns (Dalai and Jena, 2011). Therefore, there is a need for controlling the access to shared data, based on a set of specific policies. One way of doing so is via one of the many types of access control such as *discretionary*, *mandatory*, or *role-based access control* (Samarati and di Vimercati, 2000).

For reasons such as maintainability and cost effectiveness (Connor and Loomis, 2010), role-based access control (RBAC) is the most used (Gofman et al., 2009; Groenewegen and Visser, 2008) access control mechanism. RBAC (Ferraiolo and Kuhn, 1992) uses the notion of *role* as the central authorization element while other components of the system such as *subjects* and *permissions* (describing the allowed operations on *objects*) are assigned to one or more roles. However, Web applications typically consist of elements that have different granularity levels and are scattered throughout the application code which makes the access control more difficult. For example, a page in a Web application can contain smaller elements such as sections or even more fine-grained elements such as a date of a record that is retrieved from the database and displayed in a cell 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 (Hortsmann, 2012), XACML (Lorch et al., 2003; Abi Haidar et al., 2006), or Ponder (Damianou et al., 2001).

However, approaches as those listed above have three main drawbacks that complicate development of fine-grained access control models and can easily introduce 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 the *separation of concerns* (Win et al., 2002) between access

Hossein Ghotbi S. and Fischer B..
A Declarative Fine-grained Role-based Access Control Model and Mechanism for the Web Application Domain.
DOI: 10.5220/0004083400800091
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 80-91
ISBN: 978-989-8565-19-8

control and application (Chen and Huang, 2005). 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 predicates scattered throughout 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.

Recent studies (Wang et al., 2007) have shown that although testing the access control model is essential, 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 predicates, 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 an application in a medical sector than for an online forum accessible over the internet.

In this paper we present ΦRBAC, a fine-grained RBAC version 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 (Visser, 2007), a domain specific language for Web application development. It uses code generation techniques to generate and weave the access control predicates around the objects within the application code written in WebDSL. Here we describe the ΦRBAC language and the architecture of its code generator, and show its application in a case study, a departmental Web site.

2 BACKGROUND AND RELATED WORK

Our fine-grained ΦRBAC model is based on RBAC and its mechanism is implemented as an extension of a domain specific language, WebDSL. We thus give an overview of both approaches.

2.1 Role-based Access Control

RBAC belongs to the *grouping privileges* class of access control models (Samarati and di Vimercati, 2000). 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 the similarities, and so handles changes better, which leads to an easier authorization management (Gorodetski et al., 2001). RBAC is used in many domains and there are number of languages that support RBAC (Groenewegen and Visser, 2008; Abi Haidar et al., 2006; Damianou et al., 2001). RBAC uses the notion of *role* as the central authorization mechanism (Ferraiolo and Kuhn, 1992). 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 three 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. RBAC was standardized by National Institute of Standards and Technology (NIST) (Sandhu et al., 2000).

2.1.1 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 (Sujansky et al., 2010; Zhu and Lu, 2007) discuss fine-granular access control in the context of databases in terms of the table structure (i.e., columns, rows, etc.), while others (Hsieh et al., 2009; Steele and Min, 2010) 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 predicates will be scattered throughout the application code as well (Abi Haidar et al., 2006). 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 predicates. From an implementation point of view, coding and maintenance of the code will be time-consuming and error-prone (Wurster and Van Oorschot, 2009). Finally, from a testing point of view it is hard for the tester to figure out the usage coverage of hard-coded access control predicates.

2.1.2 Testing Access Control Models

Access control as a software component needs to be tested (Tondel et al., 2008). 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 (SSOD) between them, then the model should cover three cases to be complete: first, `teacher` is active and `student` is not active, second, `teacher` is not active but `student` is active and third, neither of them are active. The sufficiency of an access control depends on its target application and therefore it should 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. In this case, to overcome the super-user problem (Ferraiolo et al., 1999), 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 3.2.1.

2.2 WebDSL

WebDSL (Visser, 2007) is a domain-specific language for creating dynamic Web applications (Groenewegen et al., 2008). It provides a development environment with a higher level of abstraction than other Web programming languages (e.g., JavaScript), which enables developers to program abstract components such as pages and other presentational elements (navigations, buttons, etc.) for the presentation tier of Web applications. It also provides developers with the notion of entities for defining a data-model and enforcing data validation on those entities (Groenewegen and Visser, 2009). WebDSL already supports discretionary, mandatory, and role-based access control, but only on a coarse level of granularity such as pages and templates (Groenewegen and Visser, 2008). The WebDSL compiler uses code transformation techniques (Hemel et al., 2010) 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 a WAR file, which is later deployed on a Tomcat server (Brittain and Darwin, 2007) and then accessible (locally or remotely) via a browser.

The WebDSL compiler is implemented using SDF (Heering et al., 1989) for its syntax definition and Stratego/XT (Visser, 2003) for its transformation rules. WebDSL consists of a number of smaller domain-specific languages (e.g., user interface, access control) that are structured around a core layer. The transformation rules transform these layers step-by-step down into the core layer which is finally transformed into the target languages (e.g., JavaScript, XML, etc.).

In WebDSL the data model specifies the application's entities and their properties. Listing 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 the `tutor` property is of type `Teacher`, and `marks` is a property that holds a set of `Mark` entities.

Listing 1: An entity defined in a WebDSL data model.

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

Even though WebDSL increases the level of abstraction in Web application development, it has several shortcomings that our work here addresses, in particular:

- *Data-oriented Fine-grained Access Control:* WebDSL has a powerful data model, and even supports the validation of input data, but its access control model is oriented towards the presentational elements (i.e., pages and templates), rather than the data model, and remains coarse-grained.
- *Access Control Correctness:* Currently, there is no support to check the correctness of the access

control elements or their implementation within the application code.

3 ΦRBAC

ΦRBAC is an approach for declaratively defining and implementing a flexible, expressive, and high level RBAC mechanism. It generates access control elements and then weaves the derived predicates into the application code in order to enforce the access control on fine-grained elements of the data model, instances of the data, 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 2 shows, a ΦRBAC model consists of three main sections: basic *RBAC* elements (lines 3-11), *policy cases* (lines 13-18) and *coverage* (lines 20-26).

Listing 2: ΦRBAC example.

```
1  PhiRBAC{
2
3    roles{  teacher(10),admin(1),manager(1),
4            advisor(10),student(*)}
5    hierarchy{(advisor) -> (teacher)}
6    ssod{
7      (teacher,admin,advisor,manager) <-> (student)
8    }
9    dsod{
10     (and(advisor,teacher),admin)    <-> (manager)
11   }
12
13   objects{G(roleAssignment),XML(address),
14           Person.password,P(marks)}
15   policies{teacher,student,admin,advisor}
16   cases{  (+,-,-,?) -> ([r,u],[r],[s],[r,u]),
17           (-,-,+,?) -> ([r,u],[r],[s],[i])
18         }
19
20   coverage {
21     objects{P(root),student.marks}
22     policies{admin,teacher}
23     cases {  (+,?) -> ([r,100],[i]),
24              (-,+) -> ([i],[u,(>80,<=100)])
25           }
26   }
27 }
```

3.1.1 Basic RBAC Elements

At the core of an ΦRBAC model are the basic RBAC elements. The developer first defines roles and their cardinalities (cf. lines 3 and 4), 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 5). In addition, the developer can define optional *separation of duty* (SOD) constraints (Sandhu et al., 2000) (cf. lines 6-8). Static SOD 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 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 on a line-by-line basis as policy cases. These show the relation between the policy combinations and allowed operations on the respective objects (see Section 3.1.4 for more details).

3.1.2 Controlled Objects

ΦRBAC supports access control of objects with different types and granularity levels. We can divide them into *data model*, *page*, and *template* elements.

Any Web application that is more than just a set of linked static pages needs a supporting data manipulation mechanism, e.g., a relational database. The 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. ΦRBAC 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 2 or more fine-grained components such as `speaker` property of the entity `Seminar`. It is important to note that ΦRBAC 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 ΦRBAC 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 the testing phase (see Section 3.2.1).

Currently, WebDSL supports access control on the pages and templates of a Web application. Since we do not want to force the developer to use ΦRBAC and the existing access control to declare two different types of access control model, these coarse-grained components 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.

3.1.3 Policy Terms

A developer can select an arbitrary number of desired policy terms (i.e., activated roles), following the *policies* keyword (see Listing 2 lines 15 and 22). The actual policy is then defined case-by-case, dependent on the logical status of the policy terms. The logical status is either *activated* (represented as + in ΦRBAC), *not activated* (`-`) or *don't care* (`?`). Note that *don't care* is not required but simplifies the specification of complex policies.

3.1.4 Policy Cases

As shown in Listing 2 (see lines 16-18), the developer can specify an arbitrary number of cases. Each case defines a combination of logical states for creating an access control predicate and the set of allowed operations on the controlled objects. These operations are:

- *Create* (`c`): is used to denote that users with the appropriate roles 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.
- *Read* (`r`): refers to read operations of the controlled object itself or its embedded objects (i.e., properties as sub-elements).
- *Update* (`u`): refers to update operations of the controlled object itself or its embedded objects.
- *Delete* (`d`): refers to the delete operations related to the controlled object itself or its embedded objects.
- *Secret* (`s`): is used for hiding 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 the specified operations (i.e., create, read, update, delete).
- *Ignore* (`i`): states that the defined policy states of this case do not effect the predicates of the controlled object.

Note that all of these operations are used in relation to the defined predicates. For example, Listing 2 line 16 shows that when the user has the role `teacher` but not `student` or `admin`, he/she cannot see the users' `password`.

3.1.5 Coverage Cases

This part of the model (see lines 20-26 in Listing 2) helps the developer to define a set of independent cross checks on the ΦRBAC 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 ΦRBAC 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 3.1.3). For example, line 23 in Listing 2 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 (cf. line 17 in Listing 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 ΦRBAC model, based on a different view, with respect to the target application. For example, in Listing 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` (see the related part in Section 3.2.1 for more details).

3.2 ΦRBAC Architecture

The ΦRBAC architecture (see Figure 1) is divided into a *testing* and a *transformation* phase. The aim of the testing phase is to verify and validate the access control model itself and its integration into the target application. As the ΦRBAC 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.

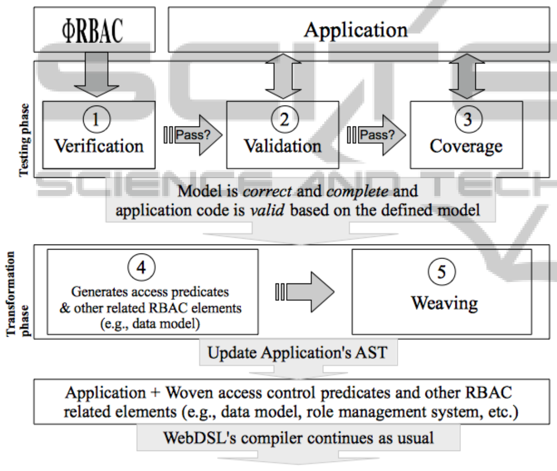
ΦRBAC
Application
Testing phase
1 Verification
Pass?
2 Validation
Pass?
3 Coverage
Model is *correct* and *complete* and application code is *valid* based on the defined model
Transformation phase
4 Generates access predicates & other related RBAC elements (e.g., data model)
5 Weaving
Update Application's 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

Figure 1: An overview of the ΦRBAC generation pipeline.

3.2.1 Testing Phase

A number of studies (Montrieux et al., 2011; Martin et al., 2006; Masood et al., 2009) highlighted 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) around the desired objects. Even partially failure of doing so will result in application code that is compilable but has a number of security holes that need to be closed *after* application deployment. 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 to the developer. In the first step, we verify the defined ΦRBAC model using model checking. Second, we validate the application code with respect to the defined ΦRBAC model. Third, we check the coverage against the defined objectives.

Model Verification. This step mechanically verifies the correctness and completeness of an ΦRBAC model using an SMT solver, *Z3* (de Moura and Bjørner, 2008). Z3 takes a representation of the model in first-order logic (FOL) and decides its satisfiability. Here, we first verify the correctness of the defined basic RBAC elements and of each individual case defined in the policy and coverage cases. Second, we check the completeness of the policy and coverage cases. For these two steps we generate a number of FOL formulas for Z3 to check them individually and then we mechanically analyze Z3's output results to come to a conclusion about the correctness and completeness of the original ΦRBAC model.

The basic RBAC elements can create conflicts in the model. For instance if a role `supervisor` inherits from a role `teacher` but these two have also an DSOD 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 DSOD) 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 models 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 defined cases.

The defined policy and coverage cases can create three types of errors that need to be checked:

- *Incorrect Case:* The policy terms and their signs create an access control predicate for each case. These signs could create an error based on the defined basic RBAC elements. For example, if there is a SSOD relation between `teacher` and `student` roles, a case cannot define a predicate in which both of them are active, so it is an error in the model if each has + for their policy sign.
- *Overlap:* If two cases create the same access con-

trol predicates, then we have an overlap between these cases. Two cases create an overlap conflict, if they are syntactically *equal* or one of them uses the *don't-care state* (?) for a set of policy terms while the other case has *active* (+) or *not-active* (-) state for those policy terms.

- *Incompleteness:* If the defined policy or coverage cases do not fully cover all the possible cases, then we have *incompleteness*.

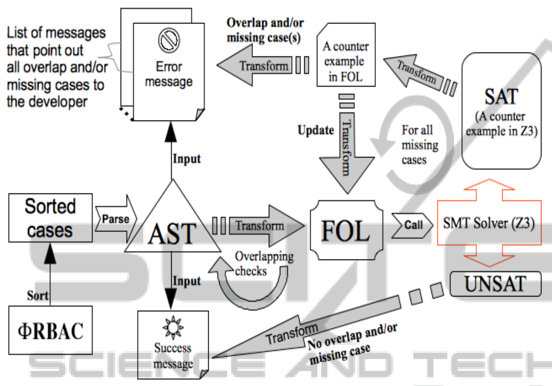

Figure 2: Overlapping and incompleteness verification pipeline.

To check these three cases, we generate three FOL formulas, and individually check their satisfiability using Z3. We then generate the error and success messages in terms of the defined ΦRBAC model instead of the model checking results.

In case of the correctness, we generate a FOL formula, for each case that contains the basic RBAC elements and uses the truth values corresponding to the policy signs for each policy term. Then we call Z3 to check the satisfiability of the formula, where SAT means that we have a correct case, and UNSAT means that we have an error because of an incorrect case. For example, to check the correctness of the case defined in Listing 2 line 17, in addition to the defined RBAC structure (Listing 2 lines 3-11) we transform its definition into a FOL formula that states `admin` is *true*, and `teacher` and `student` both have a *false* value. In this case with respect to the RBAC model, the SMT solver gives us a SAT result, because the formula did not create a conflict based on the defined RBAC model.

In checking the overlap and incompleteness of the policy and coverage cases, as Figure 2 shows, we use a number of sub-steps to check these both issues and then notify the developer about the possible errors. 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. As Figure 2 shows, 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 (see Figure 2)

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 ΦRBAC elements. As Figure 2 shows, during the last step, we parse the model checking results (UNSAT, SAT) and by retrieving its representation elements in AST we give the error during the compilation based on the ΦRBAC elements.

Listing 3: Nested controlled objects and their related predicates may create a set of conflicts.

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

Web Application Validation. In our automation mechanism, the access control predicates are woven into the application code around the controlled objects. These controlled objects and consequently their predicates may be nested within each other and 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 `groupOne`. We have `P1` that protects `groupOne` 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. For instance, `P1` is true for the users with the activated role `teacher` and `P2` is true for the users with the activated role `admin` but in the access control model there is also an `SSOD` relation between role `teacher` and `admin`. It is clear that users with the activated role `admin` can *never* access the instances of users' `username`, even though they have a right to do so. We called these unreachable areas *dead authorization code* and the following steps are used for finding such areas.

- *Sorting and Pairing:* First, we sort all policy cases based on the controlled objects, their related operations and predicates. Then, for each possible pairs 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 the application's AST such that if the paired objects are embedded within each other, we create an error with respect to the ΦRBAC model and the location of objects in the target application.

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 ΦRBAC model. A coverage percentage shows what percentage of an object 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 show in details how we calculate the coverage percentage for each controlled object:

- *Sorting and Pairing:* We sort *both* coverage and policy cases into two lists. We then pair each coverage case with all the policy cases.
- *Finding Related Cases and Partial Coverage:* Then, we need to find all the related policy cases for each coverage case based on the access control predicate. For this, we transform each pairs 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,oper)* 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 specific range we give an error in terms of ΦRBAC elements and terminate the compilation. Therefore, the developer can fix the coverage errors based on the defined ΦRBAC model and/or the target application.

3.2.2 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 ΦRBAC model.

RBAC Generation. RBAC generated elements first have to be a part of the Web application's data model for providing the data manipulation mechanisms for roles and their required 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 defined RBAC structure in ΦRBAC 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 (see Listing 5). The entity that is used for users is used as a type to represent the users that have the role (Listing 4 line 4) and also it is extended to store a set of assigned roles for each user (Listing 4 line 12). Moreover, the session element must be extended to hold the activated roles for each user. For example in Listing 5, the authentication is based on the username and password properties of the `Person` entity. In this case the entity person, represents the user of the system, and we extend the data model of the application by generating the role entity (Listing 4 lines 2-9), extending the `Person` entity (Listing 4 lines 10-13); and extending the Web application session (Listing 4 lines 15-17).

We already checked the correctness of the RBAC structure defined within the ΦRBAC model (see

3.2.1) 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 of each role to the other roles is used in the *role assignment* component which during the run-time of the system must not allow the admin 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, because two roles with DSOD relation between them 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 based on the overall structure of the defined RBAC, as we need to consider more than the direct impact of 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 covered the relation between `advisor` and `manager`. To get all direct and indirect relations of each role, as Figure 3 (second step) shows, we translate the RBAC structure into a FOL logic, and at each cycle 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).

As Figure 3 (cf. third step) shows, we first give the two steps to the *RBAC generator* to generate all the above mentioned elements.

Listing 4: 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>
}
```

Listing 5: Defined authentication credentials.

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

Predicate Generation. We already tested the policy cases (see 3.2.1), so at this stage, all the cases are unique and correct. As mentioned, each case represents a predicate that should protect the controlled objects and their related operations. As Figure 3 shows, 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 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 (see fourth step in Figure 3).

Weaving Stage. Weaving is the last step in the ΦRBAC transformation phase. In this step, we first get the result of the RBAC and predicate generators (see Figure 3). 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 ΦRBAC model (as shown in Listing 2). In terms of predicates, the generated AST holds all the predicates sorted based on operations on the controlled objects. Therefore, we need to weave these predicates *repeatedly*, because at each cycle (see step 5.1 in Figure 3) we are passing a set of predicates for a specific object and its related operation to be recursively woven by the ΦRBAC weaver to prevent AST duplications.

As Figure 3 shows, we mechanically pass the updated Web application's AST to the next step within the WebDSL compiler that is originally a part of the WebDSL compiler.

4 CASE STUDY

The aim of this section is to show the benefits and limitations of the ΦRBAC modeling language and its code generation mechanism, based on the evaluation

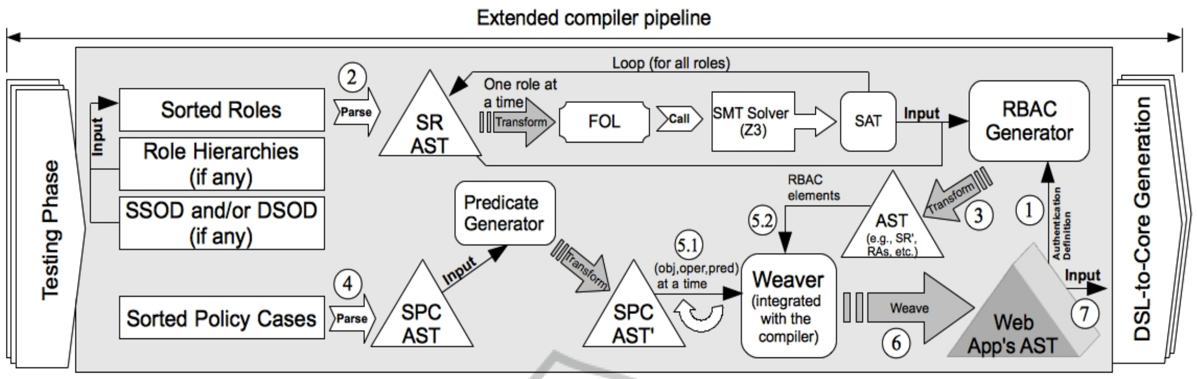

Figure 3: ΦRBAC transformation phase within the extended compiler pipeline.

of a case study. The main objective of the evaluation is to check the efficacy of ΦRBAC 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 ΦRBAC, that is intended to be used in the Web application domain.

We implemented our case study using WebDSL for the Web elements and ΦRBAC 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 viva) 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, 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 cover the set of 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 nine different types of users (e.g., `academic`, `student`, etc.) and 13 entities that cover the objects involved in activities (e.g., `publications`, etc.). Overall, we have 93 properties that are related to the 22 entities. These are the unique fine-grained objects that are used throughout the application code for a number of times. We divided the pages based on different types of users and activities, regardless of the used operations for the objects of the system. So, in this case, there is only one page for each type of data and in that page all the available operations exist in which each part of the page will be divided based on the defined policies in ΦRBAC during the transformation phase. The access control elements for this case study are based on the needs of the 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

To evaluate the ΦRBAC model and mechanism, we looked at three aspects: model, testing and transformation phases. The errors in the model were divided into RBAC and the application errors. Both RBAC and application based errors were discovered during the testing phase (see 3.2.1). The transformation strategies that were used in testing and transformation phase were tested, in a white-box manner, during their development. Also, the correction of the woven AST was inspected manually to make sure all the unguarded objects were not covered directly or indirectly within the ΦRBAC access control policies.

4.3 Findings

We organize the findings into benefits and weaknesses of ΦRBAC model and its mechanism.

The benefits are divided into development efficiency and correctness and completeness of the model and target application before its deployment. 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 scattered access control definitions in the application code. Moreover, the ΦRBAC is developed at the right abstraction level. In this case, the developer did not need to use any object or agent oriented terminology to define the access control components and she just uses these components as they are such as roles. ΦRBAC is also a cost effective solution. In our case study the compilation time related to our access control model was just 3 seconds on a machine with 4GB RAM and 2.3GHz CPU, to cover instances of 93 unique objects throughout the application. Correctness and completeness approach in ΦRBAC gives 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 is originated in the RBAC itself. 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 uses the ownership notion as a policy term as well.

5 CONCLUSIONS AND FUTURE WORK

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

For future work we like to introduce the notion of ownership (McCollum et al., 1990), as a policy term, to improve the ΦRBAC model and its mechanism. Also, we plan to integrate the other well-known access control models into our access control model, to achieve access control integration for a domain of 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 ΦRBAC 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 perform more evaluation of ΦRBAC based a broader set of Web applications.

REFERENCES

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

Brittain, J. and Darwin, I. F. (2007). *Tomcat: the definitive guide, 2nd edition.* O'Reilly.

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.

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

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

Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The ponder policy specification language. *POLICY 2001*, LNCS 1995, pp. 18–38. Springer.

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

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

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.

Gofman, M. I., Luo, R., Solomon, A. C., Zhang, Y., Yang, P., and Stoller, S. D. (2009). RBAC-PAT: A policy analysis tool for role based access control. *TACAS '09*, LNCS 5505, pp. 46–49.

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.

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.

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.

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.

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.

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.

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

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.

Lorch, M., Proctor, S., Lepro, R., Kafura, D., and Shah, S. (2003). First experiences using XACML for access control in distributed systems. *XMLSEC '03*, pp. 25–37, ACM.

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, Springer.

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.

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.

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.

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

Sanderson, D. (2009). *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. O'Reilly Media, Inc.

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.

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

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.

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

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.

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

Wang, L., Wong, E., and Xu, D. (2007). A threat model driven approach for security testing. *SESS '07*, pp. 10–17.

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.

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

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