Towards Formal Specification and Verification of a Role-Based Authorization Engine using JML

[Position paper]

Tanveer Mustafa
Center for Computing Technologies
Universität Bremen, Germany
tanveer@tzi.de

Michael Drouineaud
Center for Computing Technologies
Universität Bremen, Germany
mdruid@tzi.de

Karsten Sohr
Center for Computing Technologies
Universität Bremen, Germany
sohr@tzi.de

ABSTRACT

Employing flexible access control mechanisms, formally specifying and correctly implementing relevant security properties, and ensuring that the implementation satisfies its formal specification, are some of the important aspects towards achieving higher-level organization-wide access control that maintains the characteristics of software quality. In the access control arena, the role-based access control (RBAC) has emerged as a powerful model for laying out and developing higher-level organizational rules such as separation of duty, and for simplifying the access management process. One of the important aspects of RBAC is authorization constraints that allow one to express such organizational rules. On the other hand, the Java Modeling Language (JML) has evolved as a flexible formal behavioral interface specification language that can be used as a Design by Contract (DBC) approach for developing software written in Java. In this paper, we adopt JML as a DBC approach to implement a prototype of a role-based authorization engine. We specifically focus on how JML can effectively be used in precisely specifying the functional behavior of the authorization engine, including various constraints such as authorization constraints and integrity constraints. We employ few JML tools towards verifying the correctness of the implementation of the authorization engine against its JML specification.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: *Languages*, *Methodologies*; D.4.6 [**Security and Protection**]: *access controls*

General Terms

Design, Languages, Security, Verification

Keywords

Authorization constraints, Design by Contract, Java Modeling Language, Role-based access control, Static code analysis

1. INTRODUCTION

The research in recent years has brought role-based access control (RBAC) combined with authorization constraints [1, 2] as a flexible model for laying out and enforcing higher-level organizational rules. Specifically, such organizational rules are expressed by means of authorization constraints like separation of duty (SOD) [1, 4].

Given the fact that an organization will be running a number of different applications, a centrally administered role-based authorization engine, which implements and helps enforcing the organization-wide authorization constraints, is desirable. In this way, the authorization engine can be developed independent of any specific application, and consequently the access decision requests from various applications across the organization can be delegated to the authorization engine. However, since the authorization engine acts as a core component within an organization, the correctness of the authorization engine itself becomes crucial. The authorization engine should not be designed and implemented on the cost of any assurance compromises; otherwise the organization-wide access control may be on high security risk. It raises a critical question: what approach can we follow to design and implement an authorization engine such that the correctness of the authorization engine can be verified? Hence, in this paper, our main contribution lies in adapting and exercising a methodology that can help in verifying the correct behavior of the authorization engine.

Formal specification and light-weight verification strategies can help in achieving the correct implementation of security-critical applications. In essence, employing formal methods for high-level design verification and formal analysis are considered some of the important steps for the evaluation of high-risk security applications under the Common Criteria [5].

In recent years, the Java Modeling Language (JML) [15, 16] has evolved as a flexible formal behavioral interface specification language, which is based on the Design by Contract (DBC) paradigm [10]. JML has received considerable success because, firstly, JML is specifically tailored to Java, which uses a Java-like syntax. Secondly, apart from other features, JML supports generalized quantifiers, and also provides specification-only concepts, such as model- fields, methods and classes, which can be used to achieve an abstract and compact specification. Thirdly, there exists a range of JML supported tools that can help to conduct a light-weight verification which ensures that the implementation satisfies its formal specification [8].

In this paper, we employ JML as a DBC approach to present a prototype of a role-based authorization engine that implements functions of the RBAC standard [2]. The authorization engine can be used as a core component of an organization's access control system, i.e., it is a real-world case study. Each function is associated with a set of constraints (requirements) such as authorization constraints and integrity constraints. These constraints are formally specified in JML, mostly, in the form of methods' pre- and postconditions, and class invariants. The implementation is supposed to satisfy its JML specification.

In the process of verifying the correctness of the authorization engine we employ JML in two different ways by means of dynamic checking (runtime assertion checking) and static checking.

First, for the runtime assertion checking (which is more like a testing), we have to make the JML specifications executable. For this purpose we use the native JML compiler jmlc and the runtime assertion checker jmlrac to check if the implementation of the authorization engine satisfies its JML specification. This can be seen as a first step towards the verification of the authorization engine. We do this by running the authorization engine as a standalone application which is compiled by jmlc and by testing for violations of JML assertions against user input.

Owing to the fact that it is almost impossible to exercise all execution paths with jmlrac or even by running the test suite generated by jmlunit [8], the correctness of the implementation against its JML specifications still cannot be fully guaranteed. Therefore it becomes crucial to apply light-weight formal verification techniques that can statically check that the implementation satisfies its JML specification. The advantage of statically checking the implementation is that all execution paths (albeit within a defined bound) can be exercised at once. Hence, as a second step, we applied the Extended Static Checker for Java (ESC/Java2) [7, 12] to verify the correctness of the parts of the authorization engine w.r.t its formal specification in JML. ESC/Java2 is an automated static verification tool based on a theorem prover. The tool, however, does not claim to achieve the full verification due to the underlying unsoundness and incompleteness problems [14].

Apart from the use of runtime assertion checker and ESC/Java2, through the course of this paper we also point out their limitations and problems that we came across. We also draw a brief comparison between runtime assertion checking and static checking based upon our overall experience.

The rest of the paper is organized as follows: Section 2 provides a brief overview of related tools and technologies. In Section 3, we provide an overview of the role-based authorization engine and present the formal behavioral specification of important RBAC functions using JML. We show how JML-related verification tools can be applied. In Section 4, we recap a short comparison between runtime assertion checking and static checking with ESC/Java2. We outline our conclusions in Section 5.

2. RELATED TECHNOLOGIES

2.1 RBAC and Authorization Constraints

RBAC [1, 2] has gained much attention as a promising alternative to traditional discretionary and mandatory access control. It is an access control model in which the security administration can be simplified by the use of roles to organize the access privileges and ultimately reduces the complexity and cost of security administration [3]. The basic components of RBAC96, a widely used RBAC model introduced by Sandhu et al. [1], which serves as a basis for the ANSI RBAC standard [2] are listed below:

- the sets U, R, P, S (users, roles, permissions, and sessions, respectively)
- $UA \subseteq U \times R$ (user to role assignment relation)
- $PA \subseteq P \times R$ (permission to role assignment relation)
- $RH \subseteq R \times R$ is a partial order called the role hierarchy relation.

Authorization constraints, such as SOD, cardinality, and prerequisite role constraints [1, 4] are an important aspect of RBAC. Specifically, SOD is a fundamental principle in security systems and is typically considered as a requirement that operations are divided among two or more persons so that no single individual can compromise the security. Typically, SOD constraints are used to enforce conflict of interest policies. In a role-based system, the conflict of interest may arise as a result of a user gaining authorization for permissions associated with conflicting roles. One means of preventing this kind of conflict of interest is through *Static Separation of Duty* (SSD). A simple SSD constraint may restrict a user to be assigned to conflicting roles [2]. For example, it may be required that the same user must not be assigned to the "Cashier" role and the "Cashier Supervisor" role simultaneously. On the other hand, the *Dynamic Separation of Duty* (DSD) constraints limit the permissions that are available to a user by placing constraints on the roles that can be activated within or across a user's sessions [4].

2.2 JML and Related Tools

JML is a formal behavioral interface specification language, specifically designed for specifying the functional behavior of Java programs [15, 16]. Due to the fact that JML specifications are written by the Java programmers themselves at the source code level, JML uses a Java-like syntax and is relatively easy to understand by an average programmer.

JML provides a rich set of language constructs that are necessary to precisely specifying the functional behavior of Java programs, mostly, in the form of class *invariants*, and methods' *pre-* and *postconditions*. Here we give an overview of only few JML constructs that are necessary to understand the JML specification examples provided in this paper. For a detailed description of the language constructs, the interested reader may be referred to the JML reference manual [9].

JML specifications are written in special annotation comments in the form of /*@...@*/ or simply using //@... if a single line specification is intended. The JML tools use these annotations to parse the JML specifications out of the Java programs. JML use **requires** and **ensures** clauses to specify method's pre- and postconditions, respectively. The preconditions enforce the client's obligations, whereas postconditions enforce the implementer's obligations. JML provides a logical variable **\result** that represents the value returned by a method. \result is typically used in JML *ensures* clauses. The **\assignable** clause (also known as modifiable or modifies clause) specifies what class variables can be modified by a method's call.

Given the fact that JML guarantees a side-effect-free specification, one can call (or write) only the side-effect-free methods within JML specifications. Therefore, JML provides a **pure** keyword that can be used to declare a method as a *side-effect-free* method. A pure method does not modify any class variables, hence pure methods can be called within JML specifications.

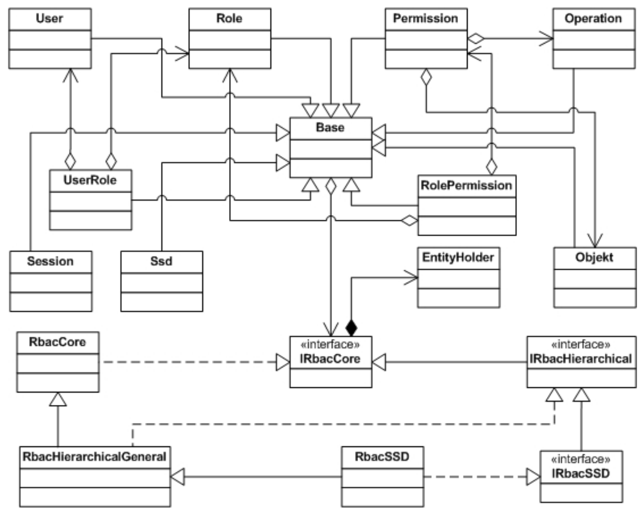


Figure 1. Class diagram of the core authorization engine.

In order to achieve a higher level of specification abstraction, JML provides *model* program concepts, such as model fields, methods and classes, which are only for the specification purposes. Due to the fact that *model* methods are specification-only methods, one can implement model methods even at the interface level within JML annotation comments. The JML release also includes a model class library providing abstract data types such as bags, sets, and sequences, which can be used to write more abstract and compact specifications. In addition, JML also provides quantifiers such as **\forall** and **\exist**.

Besides the expressiveness of JML, there exists a variety of JML tools [8][1], that can be applied to check the correctness of JML specifications, and, in particular, that the implementation satisfies its JML specifications. However, in this paper we only make use of JML compiler (*jmlc*), JML runtime assertion checker (*jmlrac*), and ESC/Java2. *jmlc* and *jmlrac* are part of the JML release [11].

The ESC/Java2 [7, 12] statically checks the Java programs that are formally annotated in JML, and it does not require much user interaction or the knowledge of formal methods. The tool includes a built-in theorem prover *simplify*, therefore an automatic program verification can be achieved. ESC/Java2 not only verifies that the implementation satisfies its JML specification, but it also exposes some (possible) implementation warnings/errors which may not be exposed by a normal Java compiler or the JML's runtime assertion checker, such as *Cast*, *Null*, *NegSize*, *IndexTooBig*, and *IndexNegative* warnings. Theoretically, ESC/Java2 is neither sound nor complete, therefore full program verification still cannot be acclaimed.

3. THE AUTHORIZATION ENGINE

In this section, we first present a prototype of a role-based authorization engine that supports the RBAC functions defined in the RBAC standard. After that, we specifically focus on how various authorization constraints associated with each function are specified in JML. Consequently these constraints are enforced by the authorization engine. We also point out how JML tools, specifically ESC/Java2, can ensure that the implementation of the authorization engine satisfies its JML specifications.

3.1 Design and Functionality

In Figure 1, a class diagram of the role-based authorization engine is shown, which consists of the necessary classes and interfaces for core RBAC, hierarchical RBAC, and constrained RBAC. The class EntityHolder describes an in-memory data structure for the creation of RBAC element sets and relations. It defines several Vector variables such as ROLES, USERS, OBS, OPS, PRMS, UA, PA, and SSD representing roles, users, objects, operations, permissions, user-role assignments, role-permission assignments and SSDs (only with a conflicting role set), respectively.

The authorization engine itself is implemented in Java and it supports most of the functionality demanded by the ANSI RBAC standard [2], which provides a set of functions that are required for the creation of RBAC element sets and relations. In particular, we have implemented administrative functions, review functions, and system functions. Administrative functions (e.g., addUser(), assignUser()) are required for the creation and maintenance of the RBAC element sets and relations. Review functions (e.g., assignedRoles(), userPermissions()) can be employed to inspect the results of the actions created by administrative functions. System functions such as createSession(), addActiveRole(), and checkAccess() are required by the authorization engine for session management and making access control decisions.

3.2 Formal Behavioral Specification

One of the important aspects of the authorization engine is to incorporate advanced RBAC concepts, such as authorization constraints, and, in particular, to verify that such constraints have been implemented correctly. Technically, such constraints have to be implemented and enforced by means of the RBAC functions as mentioned before. In the RBAC standard, these functions are defined with sufficient precision to meet the needs of conformance testing and assurance. The RBAC standard formally specifies the requirements that are associated with each function using a subset of **Z** notations [6]. For example, Figure 2 and Figure 3 show the Z specifications of the two RBAC functions createSsdSet() and assignUser(), respectively[2]. These requirements may encompass various constraints such as authorization constraints and integrity constraints.

3.2.1 Question of Linkage between Z and JML

We primarily use the Z specification given in the RBAC standard to translate it into the JML specification to realize the requirements associated with each RBAC function of the authorization engine. From the JML perspective, these requirements are mostly formulated in terms of methods' preconditions, postconditions and class invariants.

As pointed out earlier that JML provides rich specification features, such features include the most important concepts of Z, such as quantifiers, predicates, implications and sets. Thus, a manual translation from the Z specifications into the

[1] An up-to-date list of tools supporting JML can be found at *www.jmlspecs.org*

[2] For the sake of clarity, in Figure 2 we write the second parameter of the createSsdSet() function as $role\_set : 2^{ROLES}$, which, however, is written as $role\_set : 2^{NAMES}$ in the RBAC standard.

$$
\begin{array}{l}
createSsdSet(set\_name : NAME; role\_set : 2^{ROLES}; n : N) \lhd \\
\quad set\_name \notin SSD; (n \geq 2) \wedge (n \leq |role\_set|); role\_set \subseteq ROLES \\
\quad \bigcap_{\substack{r \in subset \\ subset \subseteq role\_set \\ |subset| = n}} assigned\_users(r) = \phi \\
\quad SSD' = SSD \bigcup \{set\_name\} \\
\quad ssd\_set' = ssd\_set \bigcup \{set\_name \mapsto role\_set\} \\
\quad ssd\_card' = ssd\_card \bigcup \{set\_name \mapsto n\} \rhd
\end{array}
$$

Figure 2. Functional Z specification of the createSsdSet() function.

corresponding JML specifications can be achieved that is quite abstract and compact. However, it shall be noted that our translation is not meant to be a formal transformation from Z to JML, which in turns could be verified. Further, to our knowledge, there is no such automated tool available that can verify whether JML specifications written for various methods satisfy their corresponding Z specifications. This type of verification is not the main objective of this paper. The readers interested more in the benefits of linking Z and JML, the translations strategies, and the importance of corresponding verification may be referred to [13].

In the following sections, we provide various excerpts of the JML specifications of the authorization engine, which are mostly translated from their Z specifications.

3.2.2 Specification in JML

The design of the authorization engine allows writing most of the required JML specifications at the interface level. In order to achieve a higher level of specification abstraction the JML model program concepts are used wherever required.

For the demonstration purposes, we focus here mainly on the two methods of the authorization engine, the createSsdSet() and assignUser(), which together are responsible, in particular, for the creation and enforcement of SSD authorization constraints.

createSsdSet() is an administrative function of the constrained RBAC. The available Z specification of this function is shown in Figure 2. The createSsdSet() method creates a named SSD set of conflicting roles and sets the cardinality n of its subsets that cannot have common users. A set of requirements is associated with this function stating what must be satisfied to call this function, and what must be true when it terminates. The informal description of these requirements is: (1) the name of the SSD set is not already in use, (2) all the roles in the SSD set are members of the ROLES data set, (3) n is a natural number greater than or equal to 2 and less than or equal to the cardinality of the SSD role set, (4) the SSD constraint for the new role set is satisfied.

Figure 4 shows the interface IRbacSSD, specifically focusing on the important parts of the JML specification of the createSsdSet() method. For the sake of simplicity, we also skip the details of some of the helper methods and their JML specifications. The first three JML preconditions of the createSsdSet() method specify the requirements (1), (2) and (3), respectively. The interface IRbacSSD also shows a JML model method checkSSD(), which ensures that the SSD constraint for the new role set is satisfied. checkSSD(), which is also a pure (side-effect-free) method, is used as one of the preconditions (line

$$
\begin{array}{l}
assignUser(user, role : NAME) \lhd \\
\quad user \in USERS; role \in ROLES; (user \mapsto role) \notin UA \\
\quad \forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset| = ssd\_card(ssd) \\ us = if\ r = role\ then\ \{user\}\ else\ \phi}} assigned\_users(r) \bigcup us = \phi \\
\quad UA' = UA \bigcup \{user \mapsto role\} \\
\quad assigned\_users' = assigned\_users \setminus \{role \mapsto assigned\_users(role)\} \bigcup \\
\qquad \{role \mapsto (assigned\_users(role) \bigcup \{user\})\} \rhd
\end{array}
$$

Figure 3. Functional Z specifications of the assignUser() function.

13) of the createSsdSet() method, and is responsible to specify the following part of Figure 2:

$$\bigcap_{\substack{r \in subset \\ subset \subseteq role\_set \\ |subset| = n}} assigned\_users(r) = \phi$$

The checkSSD() method implements, in terms of a model method, a complicated part of the specification of the createSsdSet() method. Since it is a specification-only method, we make use of the JML model class library that enables us to write a comparatively more compact specification. For example, we used the JML model class JMLObjectSet, which has methods such as *powerSet()*, *intersection()*, and *union()*. Firstly, we take the powerset of role_set by using JMLObjectSet. Secondly, the **while** loop iterates over the powerset until there are no more elements. Since each element of the powerset is a subset of role_set, it implies that $subset \subseteq role\_set$. Thirdly, the statement if (n >= 2 && subset.int_size() == n) extracts only those subsets such that $|subset| = n$. Finally, intersect computes the intersection of assigned users to roles of subset, which is equivalent to $\bigcap_{r \in subset} assigned\_users(r)$.

The assigned_users() method that is being called returns a set of users assigned to a role r. The assigned_users() method is declared in the interface IRbacCore and implemented by the class RbacCore. Here, we do not go into the details of the JML specification and corresponding implementation of the assigned_users() method.

It can be seen that the overall JML specification in terms of the checkSSD() method is not as short as its corresponding Z specification. The JML specification, however, still has a relevance to its Z specification.

Similarly, we now consider another important administrative function of the RBAC standard, namely, assignUser(). The core RBAC and the constrained RBAC have a different specification for the assignUser() function, which assigns a role to a user. Since we are considering the constrained RBAC for the demonstration purposes, we show and explain the assignUser() method and corresponding specifications from the perspective of constrained RBAC.

The formal Z specification of the assignUser() method for the constrained RBAC is given in Figure 3. The informal description

```
1	package org.tzi.rbac.modules;
2	import java.util.Vector;
3	import org.tzi.rbac.entities.*;
4	//@ model import org.jmlspecs.models.JMLIterator;
5	//@ model import org.jmlspecs.models.JMLObjectSet;
6
7	public interface IRbacSSD extends IRbacHierarchical {
8	    /*@  requires !entityHolder.SSD.contains(new Ssd (set_name));
9	      @  requires (\forall int i; 0 <= i && i < role_set.size();
10	      @                    role_set.get(i) instanceof Role &&
11	      @                    entityHolder.ROLES.contains(role_set.get(i)));
12	      @  requires n >= 2 && n <= role_set.size();
13	      @  requires checkSSD(role_set, n);
14	      @  ensures entityHolder.SSD.contains(new Ssd (set_name));
15	      @*/
16	    public void createSsdSet(/*@ non_null @*/ String set_name,
17	                                      /*@ non_null @*/ Vector role_set, int n);
18
19	    /*@
20	    public model pure boolean checkSSD(non_null Vector role_set, int n) {
21	        JMLObjectSet powerset =
22	                              JMLObjectSet.convertFrom(role_set).powerSet();
23	        JMLIterator itPowerset = powerset.iterator();
24	        while (itPowerset.hasNext()) {
25	            JMLObjectSet subset = (JMLObjectSet)itPowerset.next();
26	            if (n >=2 && subset.int_size() == n) {
27	                JMLIterator itSubset = subset.iterator();
28	                Role r = (Role)itSubset.next();
29	                JMLObjectSet intersect =
30	                                  JMLObjectSet.convertFrom(assigned_users(r));
31
32	                while(itSubset.hasNext()) {
33	                    r = (Role)itSubset.next();
34	                    intersect =
35	                            intersect.intersection(
36	                                  JMLObjectSet.convertFrom(assigned_users(r)));
37	                }
38
39	                if (!intersect.isEmpty())
40	                    return false;
41	            }
42	        }
43	        return true;
44	    }@*/
45	}
```

Figure 4. The interface IRbacSSD, showing the specification of the createSsdSet() method.

is given as follows: (1) the user is a member of the USERS data set, (2) the role is a member of the ROLES data set, (3) the user is not already assigned to the role, and (4) the SSD constraints are satisfied after assignment. In addition, the data set UA is updated just to reflect the assignment. The review function assigned_Users() then returns the updated set of users.

In the authorization engine, the class RbacSSD overrides the assignUser() of core RBAC as shown in Figure 5. The overriding assignUser() method must begin its specification with the JML keyword ***also*** because it should inherit the specification of the overridden method. For the sake of simplicity, however, we here show the inherited specification as part of the extended specification of the overriding method from the perspective of constrained RBAC which is based on the Z specification given in Figure 3. In Figure 5, we only show the JML specification that covers the requirements (1) to (4). We skip the rest of the specification. The notable part of the specification is the postcondition given at line 12, which fulfills the requirement (4) and corresponds to the following part of Figure 3:

$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset| = ssd\_card(ssd) \\ us = if\ r = role\ then\ \{user\}\ else\ \phi}} assigned\_users(r) \bigcup us = \phi$$

Again, in Figure 5 we have a model method named checkSSD(), which is now a part of the postcondition. In principle, this checkSSD() method is similar to the checkSSD() method given

```
1	package org.tzi.rbac.modules;
2	import java.util.Vector;
3	import org.tzi.rbac.entities.*;
4	//@ model import org.jmlspecs.models.JMLIterator;
5	//@ model import org.jmlspecs.models.JMLObjectSet;
6
7	public class RbacSSD extends RbacHierarchicalGeneral implements
8	                                                                  IRbacSSD {
9	    /*@  requires entityHolder.USERS.contains(user);
10	      @  requires entityHolder.ROLES.contains(role);
11	      @  requires !entityHolder.UA.contains(new UserRole(user, role));
12	      @  ensures (\forall Ssd ssd; entityHolder.SSD.contains(ssd);
13	      @                                       checkSSD(ssd, user, role));
14	      @*/
15	    public void assignUser (/*@ non_null @*/ User user,
16	                                           /*@ non_null @*/ Role role) {
17	        super.assignUser(user, role);
18	    }
19
20	    /*@
21	    public model pure boolean checkSSD(Ssd ssd, User user, Role role) {
22	        Vector role_set  = ssd_set(ssd);
23	        JMLObjectSet powerset =
24	                           JMLObjectSet.convertFrom(role_set).powerSet();
25	        JMLIterator itPowerset = powerset.iterator();
26	        while (itPowerset.hasNext()) {
27	            JMLObjectSet subset = (JMLObjectSet)itPowerset.next();
28	            if (!subset.isEmpty() && subset.int_size() == ssd_card(ssd)) {
29	                JMLIterator itSubset = subset.iterator();
30	                Role r = (Role)itSubset.next();
31	                JMLObjectSet us = new JMLObjectSet();
32	                if (r.equals(role))
33	                    us.insert(user);
34
35	                JMLObjectSet intersect =
36	                       JMLObjectSet.convertFrom(assigned_users(r)).union(us);
37	                while(itSubset.hasNext()) {
38	                    r = (Role)itSubset.next();
39	                    us = new JMLObjectSet();
40	                    if (r.equals(role))
41	                        us.insert(user);
42
43	                    intersect = intersect.intersection (
44	                          JMLObjectSet.convertFrom(assigned_users(r)).union(us));
45	                }
46
47	                if (!intersect.isEmpty())
48	                    return false;
49	            }
50	        }
51	        return true;
52	    }@*/
53	}
```

Figure 5. The class RbacSSD showing only the assignUser() method and its specification.

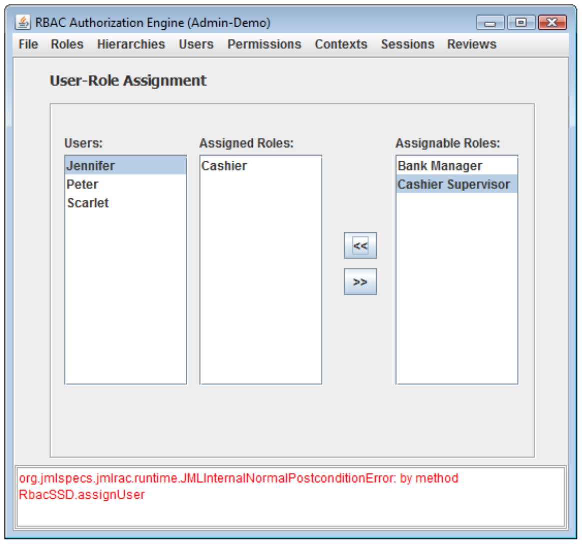
in Figure 4. This time, however, the method is slightly more complex due to its Z specification.

The assignUser() method shown in Figure 5 simply assigns a user to a role. However, Figure 5 does not show the implementation of the postcondition mentioned at line 12. We deliberately leave the implementation incomplete to be discussed in the runtime assertion checking and static checking sections.

3.3 Runtime Assertion Checking: Testing and Enforcement of Authorization Constraints

The authorization engine enforces the constraints that are specified in JML at runtime. In fact, when the authorization engine is compiled with *jmlc*, the JML specifications are converted to normal Java assertions, that is, the compiled bytecode includes runtime assertion checking instructions. Therefore, we realize the enforcement by running the authorization engine as a standalone application that is compiled by *jmlc/jmlrac* and by testing for violations of JML assertions, such as preconditions, postconditions and invariants.

Here, we give a simple example to show how the authorization engine realizes the enforcement of an SSD constraint stating that a user must not be assigned to both “Cashier” and “Cashier



```
org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError: by method
RbacSSD.assignUser
```

Figure 6. Enforcement of an SSD constraint.

Supervisor" roles. Theoretically speaking, such SSD constraints can be realized by first defining the "Cashier" and "Cashier Supervisor" roles as mutually exclusive roles (i.e. conflicting roles), and the constraint itself is enforced at the user-role assignment time. Therefore, in the authorization engine, first we create an SSD (SsdSCR) constraint with a set of conflicting roles comprised of "Cashier" and "Cashier Supervisor", which has an associated cardinality of 2. In fact, in this case, the createSsdSet() is invoked to establish such SSD constraints. The SsdSCR constraint states that a user can be assigned to maximum one role from the set of conflicting roles.

Figure 6 shows the enforcement of SsdSCR constraint when we try to assign the role "Cashier Supervisor" to the user "Jennifer". The assignment cannot be carried out because the role "Cashier" is already assigned to the user "Jennifer", i.e. a postcondition of the assignUser() method would be violated. This situation is depicted in the message view of Figure 6. In fact, in the current scenario, the JML postcondition (Figure 5, line 12) of the assignUser() method is violated, which is responsible for checking that all the SSD constraints are satisfied after a user is assigned to a role. Therefore, when we attempt to assign the role "Cashier Supervisor" to the user "Jennifer", the SsdSCR constraint is violated, and consequently the user-role assignment does not take place. Similarly, the JML specifications of various methods of the authorization engine **ensure the enforcement of various constraints** (e.g., authorization and integrity constraints).

The runtime assertion checking can also be seen from another perspective, that is, whether it can verify that the implementation is correct and if the implementation satisfies its JML specification possibly in all executions paths. For instance, in Figure 5, we deliberately left the implementation of the assignUser() method incomplete, that is, it does not implement its JML specification formulated as a postcondition given at line 12. Now, how can the runtime assertion checking verify that the implementation of the assignUser() is complete w.r.t its JML specification? In Figure 6, we came up with a simple and specific test case where we knew that an SSD violation should occur. Therefore, when a user is assigned to a role, the checkSSD() is violated. This also gives us a clue to look into the implementation of the assignUser() method for correction or completion. However, in this way, it may be difficult to exercise all possible test cases, which can assure us that the implementation always satisfies its JML specification. Hence, the static verification plays a role in this regard, which we will discuss in the following section.

3.4 Static Checking with ESC/Java2

In the previous section, we showed how the authorization engine could enforce authorization constraints by means of JML's runtime assertion checking. However, the runtime assertion checking in principle may not always force us to complete the implementation that satisfies the corresponding JML specification. Although the runtime assertion checking is precise in a sense that it does not generate false positives or false negatives, it may not guarantee that the implementation is correct in all possible execution paths. Hence, in this section, we employ JML in another way assuming that we already have completed the implementation of the authorization engine, which, however, has not been verified yet. In order to have a higher confidence in the correctness of the implementation itself and whether the implementation satisfies its JML specification, we need to apply more formal analysis and verification techniques.

In this paper, so far, we applied only ESC/Java2 as the formal verification tool for the authorization engine. Other works attempting to employ JML and ESC/Java2 in the application security context are presented by Lloyd et al. (biometric authentication system) [17] and by Cataño et al. (electronic purse) [18].

During the verification process of the authorization engine, most of the classes of the authorization engine that have a comparatively simple specifications and corresponding implementations were finally successfully verified by ESC/Java2. However, during the verification process the tool occasionally indicated some warning, such as violations of object invariants, preconditions, postconditions, and null dereferences. Since ESC/Java2 clearly pointed out such problems, we handled them by refining the specification or completing the corresponding implementation.

Although the implementation problems, such as those mentioned above, can be considered quite simple, ESC/Java2 helps in finding such bugs, and we can accordingly improve the overall implementation. However, at the same time, we faced problems when applying ESC/Java2 in situations where the resulting specification and corresponding implementation become considerably complex. Therefore, in the following sections, firstly, we show our overall experience of using ESC/Java2 in the context of the specifications and implementations given in Figure 4 and Figure 5. We briefly report why and how the tool has failed to complete the verification. Secondly, we try to demonstrate a simpler version of the specification and implementation, and we show how the ESC/Java2 still helps to correct the implementation that does not satisfy its specification.

3.4.1 Problems faced with the static checking

We have noted that in a few cases, when the resulting implementation or the specification grows in complexity, especially the specification that makes use of JML model classes, ESC/Java2 fails to complete the normal verification process due to an unexpected termination of underlying *Simplify* theorem

```
public class RbacSSDSimple {
    private final /*@ non_null @*/ EntityHolder entityHolder;

    public RbacSSDSimple() {
        entityHolder = new EntityHolder();
    }

    public /*@ pure non_null @*/ Vector assigned_users(Role  role) {...}
    public /*@ pure @*/ boolean checkSSD_simple(Ssd ssd, User user, Role role) {...}
    public /*@ pure non_null @*/ Vector powerSet(Vector role_set) {...}
    public /*@ pure non_null @*/ Vector intersection(Vector vec1, Vector vec2) {...}
    public /*@ pure non_null @*/ Vector union(Vector vec1, Vector vec2) {...}

    /*@  requires entityHolder.USERS.contains(user);
      @  requires entityHolder.ROLES.contains(role);
      @  requires !entityHolder.UA.contains(new UserRole(user, role);
      @  ensures (\forall Ssd ssd; entityHolder.SSD.contains(ssd);
      @                                            checkSSD_simple(ssd, user, role));
      @*/
    public void assignUser_simple(/*@ non_null @*/ User user,
                                                      /*@ non_null @*/ Role role) {
        entityHolder.UA.add(new UserRole(user, role));
    }
}
```

Figure 7. The class RbacSSDSimple demonstrating a simple case of constrained RBAC.

prover. For instance, executing ESC/Java2 on the class RbacSSD given in Figure 5, ESC/Java2 crashes giving the following error:

Fatal error: Unexpected exit by Simplify subprocess

In principle, assuming that ESC/Java2 does not crash, we can expect that it should always complain that the implementation of the assignUser() method is not complete because it does not satisfy or correctly implement its postcondition. This is to recall that the referenced postcondition is an important SSD that must be enforced by the assignUser() method.

In some other cases, ESC/Java2 also fails complaining that the *verification condition (VC) is too big.*

Since it was difficult for us to determine the exact cause of the problems raised above, we had a short correspondence with one of the developers of ESC/Java2. From the feedback that we received, we can briefly mention, at least, few issues regarding the static checking with ESC/Java2. Firstly, concerning the JML model class library, there are a large number of interdependent classes, which also contain a rich JML specification. Therefore, even if we make use of only the simple model class of this library, one typically ends up depending upon many others, and the verification conditions being generated by ESC/Java2 for the underlying *Simplify* theorem prover grows exponentially in size and complexity. The fact of the matter is that no one has even reasoned about the JML model classes with ESC/Java2 at this time. Secondly *Simplify,* like most automatic theorem provers, is robust only under a restricted set of inputs. Most of the modern provers will crash if they receive a predicate that is too large or too ill-structured. Therefore if *Simplify* receives verification conditions that are outrageous, it crashes sometimes. More details regarding the unsoundness and incompleteness in ECS/Java2 are given by Kiniry et al. [14].

3.4.2 Simplifying the specification

Considering our experience with using ESC/Java2, we have noted that ESC/Java2 is best suited for a modular implementation and, in particularly, for the specifications that do not grow in complexity due to the classes' interdependency or by means of the specification inheritance. Here we demonstrate a simplified version of Figure 5, which does not make use of the JML model classes and, which does not cause ESC/Java2 to crash. However, such simplification has its own downsides that we will also point out here. In Figure 7, the class RbacSsdSimple shows a rudimentary case of an implementation of constrained RBAC. To keep it simple, we have combined the necessary methods in a single class. We mainly focus on the assignUser_simple() method. We skip the specification and implementation details of the helper methods.

The checkSSD_simple() is a normal Java method implementation (not a JML model method), which corresponds to the checkSSD() model method given in Figure 5. Since in the current scenario we are not using JML model classes, we need to explicitly implement various helper methods as well, such as *powerSet()*, *union()*, and *intersection()*. The specification of the assignUser_simple() method serves the same purpose as the specification of the assignUser() method in Figure 5. However, since the checkSSD_simple() method is a normal Java method rather than a JML model method, the overall specification is much closer to the implementation itself. It also shows that, in principle, the specification of the checkSSD_simple() method is not as abstract as of the specification of the checkSSD() method given in Figure 5.

```
public void assignUser_simple (User user, Role role) {
    boolean checkssd = true;
    for (int i = 0; i < entityHolder.SSD.size(); i++) {
        Ssd ssd = (Ssd) entityHolder.SSD.elementAt(i);
        if (!checkSSD_simple(ssd, user, role)) {
            checkssd = false;
            break;
        }
    }

    if (checkssd)
        entityHolder.UA.add(new UserRole(user, role));
}
```

Figure 8. Completing implementation of the method assignUser_simple().

In Figure 7, at first, we deliberately leave the implementation of the assignUser_simple() method incomplete. When we apply ESC/Java2 on the class RbacSsdSimple, it gives a warning that the postcondition possibly not established by assignUser_simple(), which is an expected and correct behavior. Hence, in order to satisfy the ESC/Java2 warning, we complete the implementation of the assignUser_simple() method as given in Figure 8. This time, ESC/Java2 successfully verifies the method assignUser_simple() without any warnings.

4. RUNTIME ASSERTION CHECKING vs. STATIC CHECKING

We now recap a short comparison between runtime assertion checking and static checking with ESC/Java2, which is based on our experience so far.

The runtime assertion checking is precise because it is bound to the test cases carried out by the user, and it does not generate false positives or false negatives. In addition, it is usually not dependent on the complexity of the specification or the implementation. This gives us the freedom to make an extensive use of the JML features to write more abstract and compact specifications, such as using the JML model classes. The downside of runtime assertion checking is that it is almost impossible to exercise all possible execution paths by running the selective user-defined test cases. Therefore, the correctness of the implementation against its JML specification still cannot be fully guaranteed.

Contrary to runtime assertion checking, the advantage of static checking is that possibly all violations within a defined bound can be exposed at once statically. However, ESC/Java2 may produce false alarms. In addition, our experience shows that the richer and more complex JML specifications we write, the more difficult it may become for the static verification tool to fully check whether the implementation satisfies its JML specification. The question may arise why not to keep the specification simple? We can certainly write comparatively simple specifications. However, this may have its own downsides. Firstly, we may be moving away from writing a more compact and abstract specification, that is, the specification itself may look like a concrete implementation. Secondly, if we are considering to formally verify a complete application component, such as the authorization engine, then even with the simple specification, the resulting specification and implementation may be quite complex due to the classes' interdependency which in turn may not be properly handled by the static checker.

5. CONCLUSION

This paper presents a role-based authorization engine and points out its importance within an organization-wide access control system. The paper demonstrates how JML can effectively be used to formally and precisely specifying the functional behavior of the authorization engine, which encompasses various constraints such as authorization constraints and integrity constraints. One of the advantages of using JML as a formal behavioral specification language for the authorization engine is the existence of various JML tools that can be employed to verify the correctness of the implementation of the authorization engine w.r.t its formal specification. We provided excerpts of the JML specifications of the authorization engine, and employed JML runtime assertion checker and ESC/Java2 towards testing and verifying the correctness of the implementation against the JML specifications.

Our experience with ESC/Java2 shows that a static code analysis helps in gradually improving the implementation and its conformance w.r.t to the JML specification. Many of the implementation problems can be detected at an early stage, which otherwise could be difficult to identify by the JML runtime assertion checker. At the same time we learnt that ESC/Java2 lacks in power to handle complex specification and implementation cases.

We believe that using the formal specification languages, such as JML, and successfully employing the formal verification techniques, the high security risks that may arise due to incorrect implementation can be avoided or at least reduced.

6. REFERENCES

[1] R. Sandhu, E. Coyne, H. Feinstein, C. Youman. Role-based access control models, IEEE Computer, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[2] American National Standards Institute Inc. Role Based Access Control, ANSI-INCITS 359-2004, 2004.

[3] D.F. Ferraiolo, D.R. Kuhn, R. Chandramouli, Role-based access control, Artec House, Boston, 2003.

[4] R. Simon, M. Zurko. Separation of duty in role-based environments, In 10th IEEE Computer Security Foundations Workshop (CSFW '97), June 1997, pp. 183–194.

[5] Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model, Part 2: Security functional requirements, Part 3: Security assurance requirements, August 2005 Version 2.3.

[6] J.M. Spivey. The Z Notation. A Reference Manual. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[7] ESC/Java2, http://kind.ucd.ie/products/opensource/ESCJava2/

[8] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT), 7(3):212–232, June 2005.

[9] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML Reference Manual (DRAFT), October 2007.

[10] G. T. Leavens, Y. Cheon. Design by Contract with JML, 2006.

[11] JML toolset release, http://sourceforge.net/projects/jmlspecs/

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for JAVA. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, pp. 234–245, 2002.

[13] L. Freitas, J. Woodcock: Proving theorems about jml classes. In Formal Methods and Hybrid Real-Time Systems, pages 255–279, 2007.

[14] J.R. Kiniry, A.E. Morkan, B. Denby: Soundness and completeness warnings in ESC/Java2. In: SAVCBS'06, New York, NY, USA (2006) 19-24.

[15] G. T. Leavens, A. L. Baker, C. Ruby. JML: A notation for detailed design. In Behavioral Specifications of Businesses and Systems, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[16] G. T. Leavens, A. L. Baker, C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003.

[17] J. Lloyd, J. Jürjens, Security Analysis of a Biometric Authentication System using UMLsec and JML. in: 12th International Conference on Model Driven Engineering Languages and Systems (Models 2009), LNCS 5795, Springer.

[18] N. Cataño and M. Huisman. Formal specification of Gemplus's electronic purse case study. In Proceedings, FME 2002, Vol. LNCS 2391, pp. 272-289, Springer.