# A Verifiable Modeling Approach to Configurable Role-Based Access Control

Dae-Kyoo Kim, Lunjin Lu, and Sangsig Kim

Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309, USA
{kim2,l2lu,skim2345}@oakland.edu

**Abstract.** Role-based access control (RBAC) is a popular access control model for enterprise systems due to its economic benefit and scalability. There are many RBAC features available, each providing a different feature. Not all features are needed for an RBAC system. Depending on the requirements, one should be able to configure RBAC by selecting only those features that are needed for the requirements. However, there have not been suitable methods that enable RBAC configuration at the feature level. This paper proposes an approach for systematic RBAC configuration using a combination of feature modeling and UML modeling. The approach describes feature modeling and design principles for specifying and verifying RBAC features and a composition method for building configured RBAC. We demonstrate the approach by building an RBAC configuration for a bank application.

## 1 Introduction

RBAC [1] is an efficient and scalable access control model that governs access based on user roles and permissions. RBAC consists of a set of features (components), each providing a different access control function. There have been many RBAC features proposed. The NIST RBAC standard [1] presents features of core RBAC, hierarchical RBAC, static separation of duties, and dynamic separation of duties. Researchers have proposed other features such as temporal access control [2] and privacy-aware policies [3]. Not all these features are needed for an RBAC system. Depending on the requirements, one should be able to configure RBAC features by selecting only those that are needed for the requirements. For example, in commercial database management systems, Informix Online Dynamic Server 7.2 does not support static separation of duties, while Sybase Adaptive Server release 11.5 does. Informix, however, supports dynamic separation of duties, while Oracle Enterprise Server Version 8.0 does not [4]. If the requirements involve time-dependent access control (e.g., periodicity, duration), the temporal feature can be chosen.

In this paper, we present a modeling approach that enables systematic and verifiable configuration of RBAC features. This approach is motivated to reduce the development overheads and complexity of application-level RBAC systems

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 188–202, 2010.

(where access control is tightly coupled with application functions) by separating access control from application functions and configuring RBAC features on a need basis. Configured RBAC is used as a base for the functional design of the application. In the approach, RBAC features and their relationships are captured by feature modeling [5]. Rigorous design principles based on the Unified Modeling Language (UML) [6] are presented for specifying RBAC features in a form that facilitates their reuse. The design principles also serve as verification points to ensure the correctness of RBAC feature specifications. The approach defines a composition method for building configured RBAC by composing the features that are necessary for the given requirements. We demonstrate the approach by configuring RBAC for a bank application and show how the configured RBAC can be instantiated to the application.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. Section 3 describes RBAC feature modeling. Section 4 describes the formal basis of design principles for RBAC features. Section 5 presents RBAC feature specifications built upon the design principles. Section 6 describes the composition method for RBAC features. Section 7 demonstrates how RBAC features can be configured for a bank application. Section 8 concludes the paper.

## 2 Related Work

There is some work on using UML to describe access control models. The work can be categorized into two approaches. One is using the UML notation to describe the structure of an access control model and its constraints. Shin and Ahn [7] use UML class diagrams to describe the structure of RBAC and the Object Constraint Language (OCL) [8] to define RBAC constraints. Our previous work [9] uses object diagrams to visualize RBAC constraints. Priebe *et al.* [10] view an access control model as a design pattern and use the Gang-of-Four (GoF) pattern template [11] to describe RBAC. The other approach uses UML profiles, an extension mechanism in the UML, to define access control concepts. Jurjens [12] proposed a UML profile called UMLsec for modeling and evaluating security aspects for distributed systems based on the multi-level security model [13]. Similarly, Lodderstedt *et al.* proposed a UML profile called SecureUML [14] for defining security concepts based on RBAC. Doan *et al.* [15] extend the UML, not by a profile, but by directly incorporating security aspects of RBAC and MAC into UML model elements.

Composition of RBAC features in this work is related to model composition in aspect-oriented modeling (AOD) (e.g., [16,17,18,19]). In AOD, cross-cutting concerns are designed as design aspects that are separated from functional aspects (called primary models). Clarke and Walker [16] proposed composition patterns to compose design aspects described in UML templates with a primary model through parameter binding. Straw *et al.* [19] proposed a set of composition directives (e.g., creating, adding) for aspect composition. Similar to Clarke and Walker's work, Reddy *et al.* [17] use sequence diagram templates for specifying behaviors of design aspects and use tags for behavior composition. An aspect

may include position fragments (e.g., begin, end) which constrain the location of fragment interactions to be inserted in a sequence diagram. The composition method in their work, however, is not rigorously defined, and thus it is difficult to verify resulting models. Their position fragments influenced join points in our work. Song *et al.* [18] proposed a composition method for composing a design aspect with an application design. They verify composed behaviors described in OCL by discharging a set of proof obligations. However, their verification is limited to OCL expressions, and the entire composed model cannot be verified.

## 3 RBAC Feature Modeling

Feature modeling is a design method for modeling commonality and variability of an application family [5]. A feature model consists of mandatory features capturing commonality and optional features capturing variability. Features are organized into a tree-like hierarchy. Fig. 1 shows a simplified feature model for RBAC.

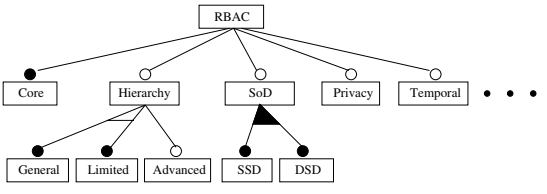


**Fig. 1.** RBAC Feature Model

We design an RBAC feature in the way that it encapsulates those and only those properties that pertain to the functions of the feature. In Fig. 1, filled circles represent mandatory features, while empty circles represent optional features. The empty triangle underneath the *Hierarchy* feature denotes an *alternative* group constraining that only one of the *General* and *Limited* features can be selected from the group. The filled triangle beneath the *SoD* feature denotes an *or* group constraining that at least one of the *SSD* and *DSD* features must be selected from the group.

The *Core* feature captures the essential RBAC functions that all RBAC systems must possess. The *Hierarchy* feature enables roles to be structured in a hierarchy in which permissions are inherited bottom-up and users are inherited top-down. A hierarchy can be either *General* or *Limited*. A general hierarchy allows a role to have more than one descendant, while a limited hierarchy is limited to only one descendant. The optional *Advanced* feature provides administrative functions for managing roles in a hierarchy. The *SoD* feature enforces *Separation of Duty* (SoD) constraints which divide responsibility for accessing sensitive information. SoD constraints are divided into *Static Separation of Duty* (SSD) and *Dynamic Separation of Duty* (DSD). The model can be extended with consideration of other features (e.g., [2,3]).

## 4 Partial Inheritance

The *Core* feature forms the basis of all configurations, and other features (henceforth, referred to as component features) add additional properties to *Core* or redefine its existing properties. This establishes inheritance relationships between *Core* and component features. However, unlike the traditional inheritance where all properties are inherited, component features may inherit only those that are needed for their functions. Partial inheritance for class diagrams and sequence diagrams is defined below.

An operation *op* with name $o$, formal parameter types $p_1, \ldots, p_n$, and return value type $r$ is denoted $o(p_1, \ldots, p_n) \vdash r$. Let $Pre(op)$ and $Post(op)$ be pre-condition and post-condition of *op*. Let $Inv(c)$ be invariant of a class $c$ and $T_1 \subseteq T_2$ denote that $T_1$ is a subtype of $T_2$. An operation $op_p = o_p(p_1, \ldots, p_n) \vdash r$ in class $c_p$ is said to redefine an operation $op_c = o_c(p'_1, \ldots, p'_m) \vdash r'$ in class $c_c$ iff O1: $o_p = o_c$, O2: $n = m$, O3: $\forall\, i \in 1..n, p'_i \subseteq p_i$, O4: $r \subseteq r'$, O5: $Pre(op_c) \wedge Inv(c_p) \Rightarrow Pre(op_p)$, O6: $Pre(op_c) \wedge Post(op_p) \Rightarrow Post(op_c)$ [20].

The cardinality of a relationship *rel* at an end $e$ is an interval of positive integers and it is denoted as $bounds(rel(e))$. The containment relationship between intervals are defined as usual. That is, $\langle l_1, u_1\rangle \subseteq \langle l_2, u_2\rangle$ iff $l_1 \geq l_2$ and $u_1 \leq u_2$. The intersection of two intervals $\langle l_1, u_1\rangle$ and $\langle l_2, u_2\rangle$ is $\langle l_1, u_1\rangle \cap \langle l_2, u_2\rangle = \langle max(l_1, l_2), min(u_1, u_2)\rangle$. The set of traces of a sequence diagram *SD* is denoted $\mathcal{T}(SD)$. A trace $s$ is a sub-sequence of another trace $t$, denoted $s \rhd t$ iff $s$ can be obtained from $t$ by removing zero or more events. We say a class $c_p$ in a component feature $f_p$ is inherited from $f_c$ if $c_p$ has the same name as a class $c_c$ in $f_c$ and we call $c_c$ the parent of $c_p$.

**Definition 1.** A component feature $f_p$ partially inherits the *Core* feature $f_c$ iff

1. At least one class in $f_p$ is inherited from $f_c$.
2. Each group of inherited classes preserves all relationships between their parents. A relationship $rel_c$ in $f_c$ is preserved iff there is a relationship $rel_p$ in $f_p$ that has the same name and the same ends as $rel_c$ and for all relationship end $e$, $bounds(rel_p(e)) \subseteq bounds(rel_c(e))$.
3. For each inherited class $c_c$ and its parent $c_p$, $\exists_{\_\, Y} \bullet Inv(c_p) \Rightarrow \exists_{\_\, Y} \bullet Inv(c_c)$ where $Y$ is the set of properties shared by $c_c$ and $c_p$ and $\exists_{\_\, Y} \bullet P$ is defined as $\exists\, z_1 . \cdots \exists\, z_n \bullet P$ and $\{z_1, \cdots, z_n\}$ contains all those variables in $P$ that are not in $Y$.
4. For each inherited class $c_c$ and its parent $c_p$, each inherited operation $op_p$ in $c_p$ redefines the corresponding operation $op_c$ in $c_c$.
5. If a sequence diagram $SD_c$ in $f_c$ and a sequence diagram $SD_p$ in $f_p$ have the same name, then every trace of $SD_c$ is a sub-sequence of some trace of $SD_p$: $\forall\, t \in \mathcal{T}(SD_c) \bullet \exists\, s \in \mathcal{T}(SD_p) \bullet (t \rhd s)$.

## 5 Specifying RBAC Features

RBAC features are specified based on partial inheritance using the UML. Due to the limited space, we present only the *Core*, *General* hierarchy and *DSD* features where *General* and *DSD* are designed to be partial inheritance of *Core*.

*Core feature.* The *Core* feature defines the properties that are required by all RBAC systems. Fig. 2 shows the structure and behaviors of the *Core* feature. The symbol "|" in the diagram denotes parameters to be instantiated after configuration. Although the operations in the class diagram are self-descriptive, their semantics should be defined clearly. We use the Object Constraint Language (OCL) [8] to define operation semantics. The following defines the semantics of *addActiveRole()*:

**context** Session:: addActiveRole(r:Role)
  **pre** : true
  **post**: **let** auth:OclMessage=User^authorizedRoles() **in**
  Auth: auth.hasReturned() and auth.result() = ars and
  Cond: **if** ars → includes(r) **then** active_in = active_in@pre → including(r)
    **else** active_in = active_in@pre **endif**

The postcondition specifies that an invocation of the operation results in invoking *authorizedRoles()* which returns a set of authorized roles for the user, and the requested role is activated only if it is included in the authorized roles. *Auth* and *Cond* are labels to be used later in this section for proving design correctness.

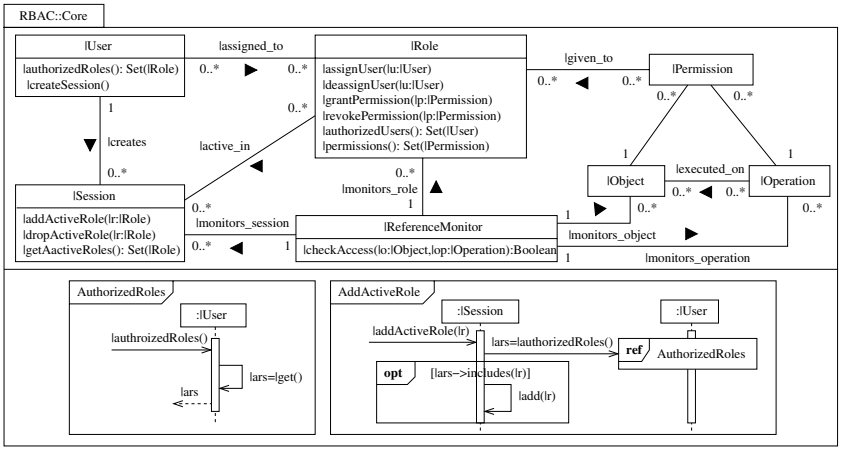


**Fig. 2.** RBAC *Core* Feature

*General hierarchy feature.* General hierarchy allows a role to have one or more immediate ascendants and descendants for inheriting user memberships and permissions from multiple sources. Fig. 3 shows the *General* hierarchy feature. Based on Definition 1, it contains only the properties that are needed for role hierarchy. The *General* hierarchy feature redefines several operations in the *Core* feature which are in bold. For example, *authorizedRoles()* and *createSession()* in the *User* class are redefined to include the roles that are inherited by the directly

assigned roles. *addActiveRole()* in the *Session* class is redefined to activate inherited roles when the requested role is activated in a session. The new semantics of *addActiveRole()* is defined as follows:

**context** Session:: addActiveRole(r:Role)
**pre** : true
**post**: **let** auth:OclMessage=User^authorizedRoles(),
desc: OclMessage = Role^descendants() **in**
Auth: auth.hasReturned() and auth.result() = ars and
Desc: desc.hasReturned() and desc.result() = descnd and
Cond: **if** ars → includes(r) **then** active_in = active_in@pre → including(r)
and descnd → forAll(d| active_in = active_in@pre → including(d)
**else** active_in = active_in@pre **endif**

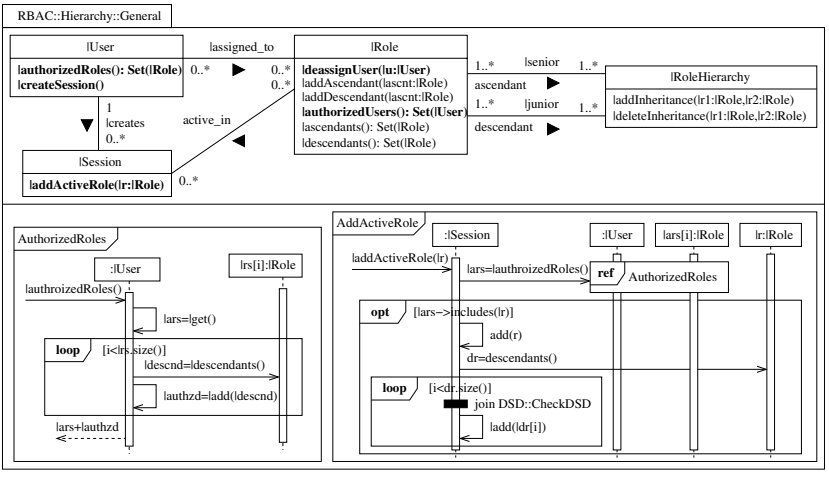


**Fig. 3.** *General* Hierarchy Feature

*deassignUser()* and *authorizedUser()* in the *Role* class are redefined to consider whether inherited roles should be also deassigned when a directly assigned role is deassigned, and whether the user can activate only the directly assigned roles or also inherited roles.

In the analysis of feature behaviors, an interference is found between the *General* hierarchy feature and the *DSD* feature when inherited roles also exist in DSD relations. To avoid the interference, inherited roles should be checked against DSD constraints before they can be activated in the same session. To handle this, we use a join point to designate where in interaction DSD constraints should be checked. The filled rectangle in the **loop** fragment in the *AddActiveRole* sequence diagram denotes a join point. The syntactic definition of join points is defined as follows based on the UML metamodel:

**follows** <Construct>
**in** <FragmentOperator> **join** [<Qualification>]::<Joint>
**precedes** <Construct>

<Construct> ::= Message | InteractionUse | CombinedFragment | "None"
<Joint> ::= Message | Interaction | InteractionUse | CombinedFragment
<FragmentOperator> ::= InteractionOperator
<Qualification> ::= Feature | Feature:<Qualification>

Given the syntax, a join point can be defined between messages, fragments, or combinations of both. If a behavior should be placed at the beginning of a sequence, the *None* construct is used in the *follows* condition. Similarly, *None* is used if a behavior should be placed at the end of a sequence. The *Qualification* construct represents the ownership of the joining construct. That is, the join point is effective only when the feature specified in the qualification is in use.

*DSD feature.* The *DSD* feature enforces DSD relations constraining that two conflicting roles cannot be activated within the same session. Fig. 4 shows the *DSD* feature. In the figure, the *DSDRole* class represents a single DSD relation, and the *cardinality* attribute specifies the number of roles to which a user can be assigned in an DSD relation. The *DSDRoleSet* class represents the set of DSD relations. The multiplicity *n* on the *Role* class denotes the DSD cardinality which must match the value of *cardinality*. *createSession()* in the *User* class and *addActiveRole()* in the *Session* class are redefined to take into account DSD constraints. The new semantics of *addActiveRole()* is defined below, checking if the requested role has an DSD relation with any active role in the session:

**context** Session:: addActiveRole(r:Role)
  **pre** : true
  **post**: **let** auth:OclMessage=User^authorizedRoles(),
  dsd: OclMessage = Role^DSDRoles(),
  violateDSD: active_in → exists (ar|ar.constrained_by_DSD → includes(r)) **in**
  Auth: auth.hasReturned() and auth.result() = ars and
  DSD: dsd.hasReturned() and dsd.result() = dr and
  Cond: **if** ars → includes(r) **then** active_in = active_in@pre → including(r)
    and not violateDSD **else** active_in = active_in@pre **endif**

To verify the correctness of the specifications, their conformance to Definition 1 must be checked. The partial inheritance between *Core* and *DSD* can be verified as follows by discharging the proof obligations in the definition:

- The first proof obligation is proved by the presence of the inherited classes *User*, *Role*, and *Session* in the *DSD* feature and the fact that the relationships *assigned_to*, *creates*, and *active_in* have the same ends and multiplicities.
- There are two classes (*User*, *Session*) in the *DSD* feature that have the same set of properties as the corresponding classes in the *Core* feature, and the second proof obligation can be proved by $Inv(User_{DSD}) \Rightarrow Inv(User_{Core})$

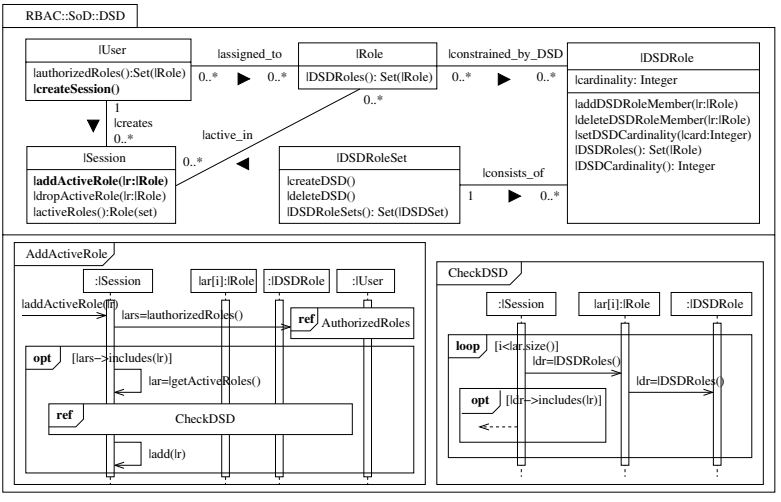


**Fig. 4.** *DSD* Feature

and $Inv(Session_{DSD}) \Rightarrow Inv(Session_{Core})$, which are both trivially true since there is no invariant defined for the *User* and *Session* classes in both *Core* and *DSD*.

- The third proof obligation must be discharged for every operation redefined in the inherited classes. As an example, *addActiveRole()* in the *DSD* feature redefines that of the *Core* feature, since they both have the same name, the same parameter type, and no return type. This satisfies the constraints $O1$, $O2$, $O3$, and $O4$. Based on the OCL semantics of the *addActiveRole()* operations, $O5$ and $O6$ can be proved by (1) $(true \wedge true) \Rightarrow true$, which is trivially true, and (2) $(true \wedge (Auth_{DSD} \wedge DSD_{DSD} \wedge Cond_{DSD})) \Rightarrow Auth_{Core} \wedge Cond_{Core}$. The second condition is proved as follows: Let $a = (ars \rightarrow includes(r))$, $b = (active\_in = active\_in \bullet pre \rightarrow including(r))$, $d = (active\_in = active\_in \bullet pre)$, and $e = violateDSD$. Let $Cond_{DSD} = ((a \Rightarrow (b \wedge \neg e)) \vee d)$ and $Cond_{Core} = ((a \Rightarrow b) \vee d)$. It suffices to prove that $(Auth_{DSD} \wedge Cond_{DSD}) \Rightarrow (Auth_{Core} \wedge Cond_{Core})$ since $(Auth_{DSD} \wedge DSD_{DSD} \wedge Cond_{DSD}) \Rightarrow (Auth_{DSD} \wedge Cond_{DSD})$. Since $(b \wedge \neg e) \Rightarrow b$, we have $(a \Rightarrow (b \wedge \neg e)) \Rightarrow (a \Rightarrow b)$ which implies $Cond_{DSD} \Rightarrow Cond_{Core}$ which in turn implies $(Auth_{DSD} \wedge Cond_{DSD}) \Rightarrow (Auth_{Core} \wedge Cond_{Core})$. Other operations can be proved similarly.
- The fourth proof obligation is concretized as $\forall\, t \in \mathcal{T}(AddActiveRole_{Core}) \bullet \exists\, s \in \mathcal{T}(AddActiveRole_{DSD}) \bullet (t \rhd s)$ for the *AddActiveRole* sequence diagram. There are two traces involved in $\mathcal{T}(AddActiveRole_{Core})$, $<addActiveRole(), authorizedRoles()>$ and $<addActiveRole(), authorizedRoles(), add(r)>$. The first one exists in $\mathcal{T}(AddActiveRole_{DSD})$ and the second one is a subsequence of $<addActiveRole(), authorizedRoles(), getActiveRoles(), DSDRoles(), DSDRoles(), add()>$ which also exists in $\mathcal{T}(AddActiveRole_{DSD})$. Thus, the proof obligation is discharged.

The partial inheritance between the *Core* feature and the *General* hierarchy features can be verified similarly.

## 6 Composition Method

The partial inheritance of RBAC features enables step-wise composition, which allows verification of immediate impact of selected features. The *Core* feature is selected by default as the first configuration. The $n^{th}$ configuration is built upon the $(n-1)^{th}$ configuration by adding or redefining the properties of the selected feature. We view this approach a special kind of multiple inheritance where the elements having the same name get composed rather than renamed as in the traditional multiple inheritance. Fig. 5 illustrates the approach.

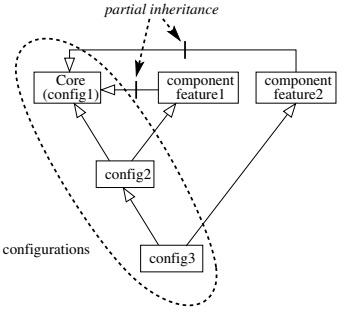


**Fig. 5.** Multiple Inheritance of Configurations

We define feature composition in the view of multiple inheritance as refinement as follows.

*Relationship composition.* In composition of class diagrams $CD_1$ and $CD_2$, relationship $a_1$ from $CD_1$ is composed with relationship $a_2$ from $CD_2$ if $a_1$ and $a_2$ have the same name and same relationship ends. The composed relation denoted $a_1 \oplus a_2$ has the same name and relationship ends as $a_1$ and $a_2$ and $bounds((a_1 \oplus a_2)(e)) = bounds(a_1(e)) \cap bounds(a_2(e))$ for each relationship end $e$. This ensures that the resulting end has the maximal bound interval that conforms to the end of both $a_1$ and $a_2$.

*Operation composition.* Operation $o(p_1, \ldots, p_n) \vdash r$ is said to match with operation $o'(p'_1, \ldots, p'_m) \vdash r'$ iff (1) $o = o'$, (2) $n = m$, (3) $\forall\, i \in 1..n, p'_i \subseteq p_i \vee p_i \subseteq p'_i$, (4) $r \subseteq r' \vee r' \subseteq r$. The composition of two matching operations $op_1$ and $op_2$ is denoted $op_1 \oplus op_2$. Let $op_1 = o(p1_1, \ldots, p1_n) \vdash r_1$ and $op_2 = o(p2_1, \ldots, p2_n) \vdash r_2$. We require that $op_1 \oplus op_2$ be an operation $(o(p'_1, \ldots, p'_n) \vdash r')$ that satisfies

P1 $\forall\, i \in 1..n, p'_i = lub(p1_i, p2_i)$ where *lub* is the least upper bound operation;
P2 $r' = glb(r_1, r_2)$ where *glb* is the greatest lower bound operation;
P3 $Pre(op_1) \wedge Pre(op_2) \wedge Inv(c') \Rightarrow Pre(op')$;
P4 $Pre(op_1) \wedge Pre(op_2) \wedge Post(op') \Rightarrow Post(op_1) \wedge Post(op_2)$.

P1 ensures that composed operations can take any parameter valid for component operations. P2 postulates that the return value of composed operations be of the type that conforms to the return type of component operations. Consistent with P1 and P2, P3 enforces that the precondition of composed operations must not be stronger than that of component operations. P4 constrains that the postcondition of composed operations must not be weaker than that of component operations.

*Class composition.* Class $c_1$ in class diagram $CD_1$ is composed with class $c_2$ in class diagram $CD_2$ if they have the same name. Let $OP(c)$ be the set of operations in class $c$. Class $c' = c_1 \oplus c_2$ is the composition of $c_1$ and $c_2$ iff

C1 $Inv(c') \Rightarrow Inv(c_1) \wedge Inv(c_2)$;
C2 $\forall\, op \in OP(c_1) \cup OP(c_2) \bullet \exists\, op' \in OP(c') \bullet name(op) = name(op')$;
C3 $op_1 \oplus op_2 \in OP(c')$ iff $op_1 \in OP(c_1)$, $op_2 \in OP(c_2)$ and $op_1$ matches $op_2$.

C1 ensures that the invariant of the classes that are composed is preserved in the resulting class. C2 requires that the resulting class have the operations of both the classes composed. C3 ensures that only matching operations can be composed. We are now ready to define composition operations on class diagrams. Let $\mathcal{E}(CD)$ be the set of classes and relationships of class diagram $CD$.

**Definition 2.** An operation $\oplus$ on class diagrams is a composition operation iff

- $\forall\, e_1 \in \mathcal{E}(CD_1) \bullet [\forall\, e_2 \in \mathcal{E}(CD_2) \bullet (name(e_1) \neq name(e_2)$ $\Rightarrow e_1 \in \mathcal{E}(CD_1 \oplus CD_2)]$ and $\forall\, e_2 \in \mathcal{E}(CD_2) \bullet [\forall\, e_1 \in \mathcal{E}(CD_1) \bullet (name(e_1) \neq name(e_2)) \Rightarrow e_2 \in \mathcal{E}(CD_1 \oplus CD_2)]$;
- $\forall\, e_1 \in \mathcal{E}(CD_1) \bullet \forall\, e_2 \in \mathcal{E}(CD_2) \bullet (name(e_1) = name(e_2)$ $\Rightarrow \exists\, e' \in \mathcal{E}(CD_1 \oplus CD_2) \bullet e' = e_1 \oplus e_2)$

An operation on sequence diagrams $\oplus$ is a composition operation if each trace of $SD_1 \oplus SD_2$ can be obtained by interleaving a trace of $SD_1$ and a trace of $SD_2$ and all traces of $SD_1$ and $SD_2$ are used. The interleave of two traces of events is the set of traces obtained by interleaving the two traces in all possible ways. Let $x, y$ be events and $\mu, \nu$ traces. The following definition of the interleave operator $|||$ is adapted from [21].

$$
\begin{aligned}
\epsilon \,|||\, \mu &= \mu \\
\mu \,|||\, \epsilon &= \mu \\
x\,\mu \,|||\, x\nu &= \{x\} \times ((\mu \,|||\, x\nu) \cup (x\,\mu \,|||\, \nu) \cup (\mu \,|||\, \nu)) \\
x\,\mu \,|||\, y\nu &= \{x\} \times (\mu \,|||\, y\nu) \cup \{y\} \times (x\,\mu \,|||\, v) \text{ for } x \neq y
\end{aligned}
$$

Note that the above definition allows us to replace two consecutive occurrences of the same event by a single occurrence if they arise from different traces that are interleaved.

**Definition 3.** A composition operation $\oplus$ on sequence diagrams is defined iff

1. $\forall\, t \in \mathcal{T}(SD_i) \bullet \exists\, t' \in \mathcal{T}(SD_1 \oplus SD_2) \bullet (t \rhd t')$ for $i = 1, 2$;
2. $\forall\, t' \in \mathcal{T}(SD_1 \oplus SD_2) \bullet \exists\, t_1 \in \mathcal{T}(SD_1) \bullet \exists\, t_2 \in \mathcal{T}(SD_2) \bullet t' \in (t_1 \,|||\, t_2)$ where $t_1 \,|||\, t_2$ is the set of traces obtained from interleaving $t_1$ and $t_2$ in all possible ways.

## 7 Configuring RBAC

To demonstrate RBAC configuration, we use a banking application taken from [22]. The application requires the following RBAC policies:

**R1:** A teller can modify deposit accounts.
**R2:** A customer service representative can create or delete deposit accounts.
**R3:** An accountant can create general ledger reports.
**R4:** An accounting manager can modify ledger-posting rules.
**R5:** A loan officer can create and modify loan accounts.
**R6:** The customer service representative role is senior to the teller role.
**R7:** The accounting manager role is senior to the accountant role.
**R8:** A user may be assigned the customer service representative role and the loan officer role, but they cannot be activated simultaneously.

R1-5 describe general authorization requirements for roles, which can be addressed by the *Core* feature. R6-7 describe role hierarchies, which can be satisfied by the *General* hierarchy feature. R8 describes a dynamic SoD requirement to be addressed by the *DSD* feature. The selection is assumed to be in the order of *Core*, *General* hierarchy, and *DSD*, but it can be in any order by partial inheritance. The *Core* feature itself forms the first configuration by the composition method.

### 7.1 Second Configuration

The second configuration is built by composing the *Core* and *General* hierarchy features, which involves 1) adding the *RoleHierarchy* class and its associated relationships to the *Core* feature, 2) composing the co-existing classes of *User*, *Session*, and *Role*, and 3) composing the matching operations in other classes (e.g., *authorizedRoles()*, *addActiveRole()*). Based on the composition method, two operations are composed by conjoining preconditions and postconditions. Due to partial inheritance, the composition of the *addActiveRole()* operations results in the same operation as that of the *General* hierarchy feature. Thus, the composed operation satisfies the constraints $P1 - 4$ in Section 6.

The *AddActiveRole* sequence diagrams are composed by adding the *ars[i]:Role* and *r:Role* lifelines to the *AddActiveRole* sequence diagram in *Core* for checking authorized descendant roles. The **loop** fragment from the *General* hierarchy feature is added to check violation of DSD policies in the descendant roles. The fragment is enabled only when the *DSD* feature is used. Note that the composition results in the same sequence diagram as the one in the *General* hierarchy feature. This is because of the constraint 4 in Definition 1 requiring that a component feature include every trace of the *Core* feature, which is consistent with Definition 3. Thus, the resulting configuration also conforms to the composition method. This is true for every sequence diagram in the second configuration, provided that component features conform to Definition 1. The *AuthorizedRole* operations can be composed similarly.

### 7.2 Third Configuration

The final configuration is built by composing the second configuration *Config*2 with the *DSD* feature. The composition is carried out by 1) adding the *DSDRole*

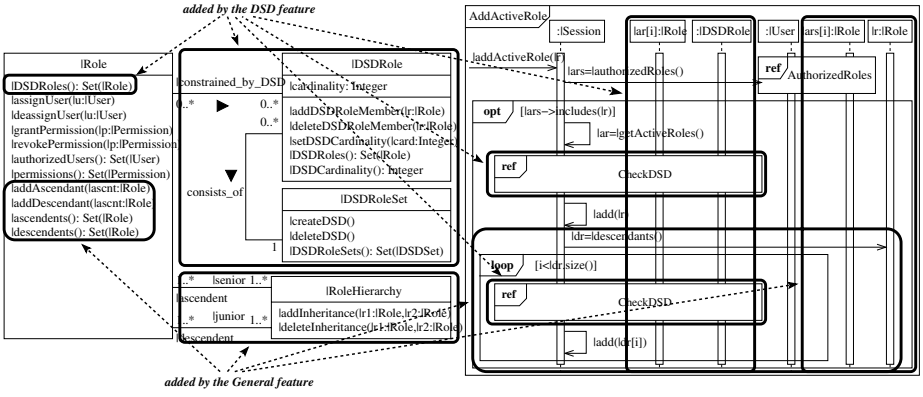


**Fig. 6.** Partial Composition of *Config*2 with *DSD* Feature

and *DSDRoleSet* classes and their associated relationships to *Config*2, and 2) composing the co-existing classes of *User*, *Role*, and *Session*. Fig. 6 shows the *DSD* properties added to the *Config*2 class diagram.

The composition of the *addActiveRole()* operations in *Config*2 and the *DSD* feature results in an operation with the following semantics which checks DSD policies for the requested role and its inherited roles to active roles:

**context** Session:: addActiveRole(r:Role)
**pre** : true
**post**: **let** auth:OclMessage=User^authorizedRoles(),
desc: OclMessage = Role^descendants(),
dsd: OclMessage = Role^DSDRoles(),
violateDSD: active_in → exists (ar|ar.constrained_by_DSD → includes(r)) **in**
Auth: auth.hasReturned() and auth.result() = ars and
Desc: desc.hasReturned() and desc.result() = descnd and
DSD: dsd.hasReturned() and dsd.result() = dr and
Cond: **if** ars → includes(r) **then**
(active_in = active_in@pre → including(r) and
descnd → forAll(d| active_in = active_in@pre → including(d)) and
not violateDSD **else** active_in = active_in@pre **endif**

The resulting semantics conforms to the definition of operation composition in Section 6 as follows:

- $P1$ and $P2$ are trivially true, since the composed operation has the same name and parameter type (*Role*) and no return type as that of *Config*2 and the *DSD* feature.
- $P3$ is true since $(true \wedge true \wedge true) \Rightarrow true$.
- $P4$ is true by $true \wedge true \wedge Auth_{Config3} \wedge Desc_{Config3} \wedge DSD_{Config3} \wedge Cond_{Config3} \Rightarrow (Auth_{Config2} \wedge Desc_{Config2} \wedge Cond_{Config2}) \vee (Auth_{DSD} \wedge DSD_{DSD} \wedge Cond_{DSD})$ where *Config*3 signifies the third configuration.

The *AddActiveRole* sequence diagram in *Config*2 is redefined to enforce DSD policies. The *getActiveRoles()* message, the *CheckDSD* fragment, and the *ar[i]:Role* and *:DSDRole* lifelines of the *DSD* feature are added to *Config*2 to check DSD policies for active roles. The composition also introduces another *CheckDSD* fragment at the place of the join point on the *Session* lifeline, which is responsible for checking DSD policies for the descendant roles of the requested role.

The composed sequence diagram conforms to Definition 3 as follows: Every trace of the sequence diagram in *Config*2 with the join point expanded is a sub-sequence of a trace of the sequence diagram in Fig. 6; and every trace of the *AddActiveRole* sequence diagram of the *DSD* feature is a subsequence of a trace of the sequence diagram in Fig. 6. Thus, the postulate (1) is satisfied. Now consider the postulate (2). Let $m_0 = addActiveRole(r)$, $m_1 = autorizedRoles()$, $m_2 = getActiveRoles()$, $m_3 = add(r)$, $m_4 = descendants()$, $m_5[i] = add(dr[i])$ and $n = dr.size()$. Then the traces of the sequence diagram in Fig. 6 are of these forms: $m_0 m_1$, $m_0 m_1 m_2 \sigma_0$, $m_0 m_1 m_2 \sigma_0 m_3 m_4 \sigma_1$, $m_0 m_1 m_2 \sigma_0 m_3 m_4 \sigma_1 m_5[1] \sigma_2$, ..., $m_0 m_1 m_2 \sigma_0 m_3 m_4 \sigma_1 m_5[1] \ldots m_5[i-1] \sigma_i$ for $2 \leq i \leq n$ and $m_0 m_1 m_2 \sigma_0 m_3 m_4 \sigma_1 m_5^1 \ldots m_5[i-1] \sigma_i \ldots \sigma_n m_5[n]$ where each $\sigma_j$ for $0 \leq j \leq n$ is a trace of the *CheckDSD* sequence diagram. The traces that end with $\sigma_j$ result from violations of the *DSD* constraint. Each of the above traces can be obtained by interleaving a trace of the *AddActiveRole* sequence diagram of *Config*2 with the join point expanded and a trace of the *AddActiveRole* sequence diagram in the *DSD* feature.

Fig. 7 shows partial instantiation of the third configuration in the context of the bank application. The instantiation is carried out based on a mapping between RBAC elements and application concepts. For example, *Object* and

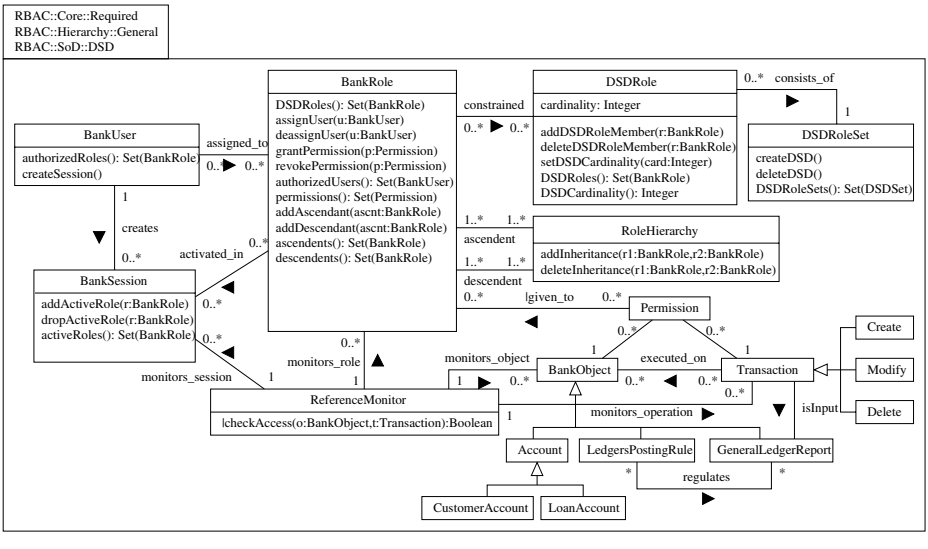


**Fig. 7.** Instantiation of RBAC Configured with *Core*, *General* hierarchy, and *DSD*

*Operation* in RBAC are mapped, respectively, to the hierarchy of bank objects such as *Account* and *LedgersPostingRule* and transaction operations such as *Create* and *Modify*. The instantiation lends itself as an initial design model for the application addressing access control concerns.

## 8 Conclusion

We have described a feature-based modeling approach for configuring RBAC to support the need-based development of access control systems. This approach enables fine-grained configuration of RBAC at the feature level in a systematic manner, which helps to lower development complexity and reduce potential errors by excluding unnecessary features. The composition method allows one to rigorously verify RBAC configurations. We have developed a prototype tool that supports feature selection and composition and instantiation of configurations. The tool is developed as an eclipse plug-in on top of Rational Rose Architects (RSA). We have also used the approach for Mandatory Access Control (MAC) and Discretionary Access Control (DAC). We found the approach less appealing for these models because of their low variability. However, our pilot study shows that the approach is useful for building hybrid models of RBAC and MAC which are often used in the military domain to support polices at different levels of security per role. Configuring a hybrid model requires a comprehensive analysis of both domains to identify possible conflicts in combined use of heterogeneous features.

## References

1. Ferraiolo, D., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. ACM Trans. on Information and Systems Security 4(3) (2001)
2. Bertino, E., Bonatti, P., Ferrari, E.: TRBAC: A Temporal Role-based Access Control Model. ACM Trans. on Information and Systems Security 4(3), 191–223 (2001)
3. Ferraiolo, D., Kuhn, D.R., Chandramouli, R.: Role-Based Access Control, second edition. Artech House (2007)
4. Ramaswamy, C., Sandhu, R.: Role-Based Access Control Features in Commercial Database Management Systems. In: Proc. of the 21st NIST-NCSC Conference (1998)
5. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90TR-21 (1990)
6. The Object Management Group (OMG): Unified Modeling Language: Superstructure. Version 2.1.2 formal/07-11-02, OMG (November 2007), `http://www.omg.org`
7. Shin, M., Ahn, G.: UML-Based Representation of Role-Based Access Control. In: Proc. of IEEE Int. Workshop on Enabling Technologies, pp. 195–200 (2000)
8. Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison Wesley, Reading (2003)
9. Kim, D., Ray, I., France, R., Li, N.: Modeling Role-Based Access Control Using Parameterized UML Models. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 180–193. Springer, Heidelberg (2004)

10. Priebe, T., Fernandez, E., Mehlau, J., Pernul, G.: A Pattern System for Access Control. In: Proc. of Conf. on Data and Application Security, pp. 22–28 (2004)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
12. Jurjens, J.: UMLsec: Extending UML for Secure Systems Development. In: Proc. of the 5th Int. Conf. on the UML, Dresden, Germany, pp. 412–425 (2002)
13. Harrison, M., Ruzzo, W., Ullman, J.: Protection in Operating Systems. Communications of the ACM 19(8), 461–471 (1976)
14. Lodderstedt, T., Basin, D.A., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: Proc. of the 5th Int. Conf. on the UML, Dresden, Germany, pp. 426–441 (2002)
15. Doan, T., Demurjian, S., Phillips, C., Ting, T.: Research Directions in Data and Applications Security XVIII. In: Proc. of the 18th IFIP TC11/WG 11.3 Annual Conf. on Data and Applications Security, Catalonia, Spain, pp. 25–28 (2004)
16. Clarke, S., Walker, R.: Composition Patterns: An Approach to Designing Reusable Aspects. In: Proc. of Int. Conf. on Software Engineering, pp. 5–14 (2001)
17. Reddy, R., Solberg, A., France, R., Ghosh, S.: Composing Sequence Models using Tags. In: Proc. of MoDELS Workshop on Aspect Oriented Modeling (2006)
18. Song, E., Reddy, R., France, R., Ray, I., Georg, G., Alexander, R.: Verifiable Composition of Access Control and Application Features. In: Proc. of the 10th ACM Symp. on Access Control Models and Technologies, Stockholm, Sweden, pp. 120–129 (2005)
19. Straw, G., Georg, G., Song, E., Ghosh, S., France, R., Bieman, J.: Model Composition Directives. In: Proc. of the 7th Int. Conf. on the UML, Lisbon, Portugal (2004)
20. Brady, A.F.: A Taxonomy of Inheritance Semantics. In: Proc. of the 7th Int. Workshop on Software Specification and Design, Redondo Beach, California, pp. 194–203 (1993)
21. Störrle, H.: Semantics of interactions in UML 2.0. In: Proceedings of IEEE Symposium on Human Centric Computing Languages and Environments
22. Chandramouli, R.: Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks. In: Proc. of Workshop on Role-based Access Control (2000)