Verifiable Composition of Access Control and Application Features

Eunjee Song
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
song@cs.colostate.edu

Raghu Reddy
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
raghu@cs.colostate.edu

Robert France
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
france@cs.colostate.edu

Indrakshi Ray
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
iray@cs.colostate.edu

Geri Georg
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
georg@cs.colostate.edu

Roger Alexander
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
rta@cs.colostate.edu

ABSTRACT

Access control features are often spread across and tangled with other functionality in a design. This makes modifying and replacing these features in a design difficult. Aspect-oriented modeling (AOM) techniques can be used to support separation of access control concerns from other application design concerns. Using an AOM approach, access control features are described by aspect models and other application features are described by a primary model. Composition of aspect and primary models yields a design model in which access control features are integrated with other application features. In this paper, we present, through an example, an AOM approach that supports verifiable composition of behaviors described in access control aspect models and primary models. Given an aspect model, a primary model, and a specified property, the composition technique produces proof obligations as the behavioral descriptions in the aspect and primary models are composed. One has to discharge the proof obligations to establish that the composed model has the specified property.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Languages, Methodologies; D.2.4 [**Software/Program Verification**]: Validation; K.6 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Design, Security, Languages, Verification

Keywords

RBAC, Modeling, Verification, UML

1. INTRODUCTION

Access control policies are constraints that determine the type of access authorized users have on information resources. For example, an access control policy in a banking system can stipulate that only loan managers can create and update computer-based customer loan accounts. From a software design perspective, access control policies are requirements that must be addressed in a design. Design features that enforce access control policies are often spread across and tangled with other functionality in a design. For example, addressing an access control policy can require one to include in each application service an authorization check that is performed before providing the service.

Access control features that crosscut a design are problematic for the following reasons: (1) Changing the access control feature requires making changes in a number of places in the design, (2) Evaluating alternative ways of enforcing access control policies is difficult when the features crosscut a system design, and (3) Understanding a cross-cutting access control feature can be difficult because its description is scattered across a design. These problems can result in software that cannot be trusted to protect sensitive or mission-critical information.

Aspect Oriented Modeling (AOM) techniques allow system developers to separate cross-cutting access control features from other application features. In AOM, an application design consists of one or more aspect models and a primary model. The aspect models describe cross cutting features (e.g., access control features) and the primary model describes other application features. Composing access control aspect models with a primary model produces an application design model in which access control features are integrated with application features. The result is referred to as a *composed model*.

A key issue in applying the AOM approach is determining whether composition of aspect models and a primary model produces a composed model that has specified properties. In this paper, we illustrate how the behavioral views described by an access control aspect model and a primary model can be composed in a verifiable manner. We use the AOM approach to produce an aspect model

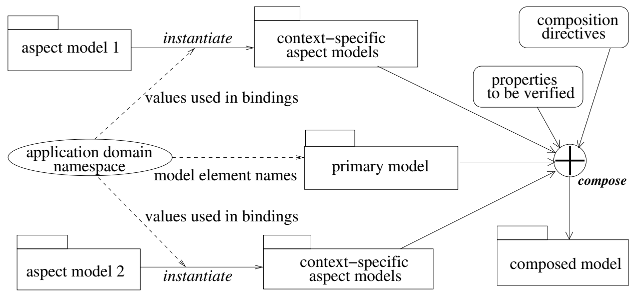


Figure 1: An overview of composition in the AOM approach

describing Role Based Access Control (RBAC) [5] and a primary model describing part of a banking application. Using the banking application primary model, the RBAC aspect model, and a specified property, we illustrate how proof obligations can be generated as the behavioral views of the aspect and primary models are composed. Discharging the proof obligations during composition can help one identify the sources of problems when the obligations do not hold.

The remainder of the paper is organized as follows. Section 2 gives an overview of the AOM approach and Section 3 presents the RBAC aspect model. Section 4 illustrates how the RBAC aspect model can be composed with a banking application primary model. Section 5 illustrates how proof obligations can be produced during composition of behavioral views. Section 6 gives an overview of related work and Section 7 concludes the paper with an overview of our plans to further develop the approach.

2. BACKGROUND

In the AOM approach, features that crosscut a design can be described by aspect models if their distributed parts have common characteristics. In these cases the cross-cutting features can be isolated and described as patterns. Aspect models are descriptions of patterns and composition of aspect and primary models involves incorporating instantiations of the patterns into specified parts of the primary model. An overview of composition in the AOM approach is shown in Fig. 1. An AOM design model consists of the following artifacts: (1) A *primary model* which describes application features not described by aspect models, (2) A set of *aspect models*, where each model describes a pattern that is a generic (parameterized) description of a cross-cutting feature, (3) A set of *bindings* that determines the pattern instantiations that will be produced and composed with the primary model, and (4) A set of *composition directives* that determines how aspect models are composed with the primary model.

Before an aspect model can be composed with a primary model, the aspect model must be instantiated in the context of the application domain. An instantiation is obtained by binding elements in the aspect model to elements in the application domain. The result is called a *context-specific aspect model*. A context-specific aspect model is produced for each part of the primary model into which the aspect feature is to be incorporated. (For further details, refer to [6].)

In our work, primary and context-specific aspect models are expressed in the Unified Modeling Language (UML) [21]. The UML is an Object Management Group (OMG) standard modeling language. A system is described in the UML using multiple diagrams that present different views of the system. In this paper we use only two types of diagrams: Class diagrams specify static structure and sequence diagrams describe how objects interact to accomplish tasks. A UML class diagram consists of a set of classifiers (for example, classes, interfaces) and their relationships (for example, association, generalization). Classes may have attributes and operations. In this paper, operation specifications and constraints on attributes are expressed using the Object Constraint Language (OCL) [23]. A UML sequence diagram presents a behavioral view that focuses on the interactions that take place between class objects when they collaborate to accomplish a specific task. The interactions are expressed in terms of lifelines representing objects and messages that are passed between objects.

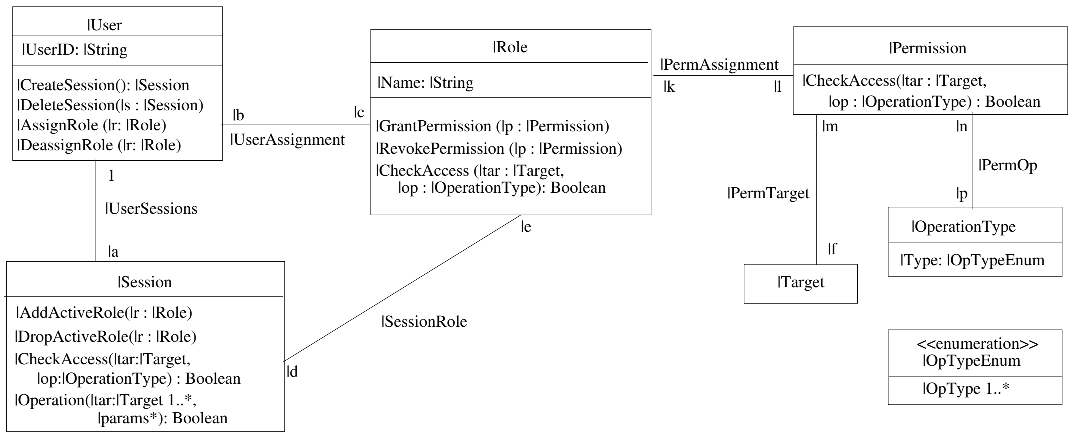
Aspect models consist of template forms of UML diagrams. In this paper, aspect models consist of class and sequence diagram templates. Instantiating an aspect model involves binding template parameters to names in an application domain namespace (see Fig. 1). A class diagram template consists of parameterized elements, such as, relationship templates and class templates that consist of attribute templates and operation templates. Attribute templates can be associated with OCL constraint templates that produce constraints that restrict attribute values when instantiated. Similarly, operation templates can be associated with OCL pre- and postcondition templates that produce operation specifications when instantiated. Examples of aspect models are given in the Section 3.

Composition of aspects and a primary model involves composing a context-specific aspect model's class diagrams and a primary model's class diagrams, and composing their sequence diagrams. The AOM approach uses a basic name-based composition algorithm in which elements with the same name are merged [6]. If an element appears in a primary or aspect model and not in the other, then the element is included in the composed model. Composition directives can be used to modify the base composition algorithm [20]. A composition directive can be used to rename elements and to specify that (1) an element in one diagram overrides a matching element in another diagram, (2) an element must not be present in the composed model, and that (3) a new element is to be included in the composed model. Composition directives are not described further in this paper. For more details and examples of directives see Straw *et al.* [20].

It is sometimes necessary to establish that composition yields a model that is correct with respect to specified properties. In this paper we illustrate an approach that supports verification of behavioral properties. During composition, proof obligations are produced using information that is available in the partially composed model. A proof obligation can be discharged manually or with the help of automated tools.

The types of correctness checks that can be carried out on a model is determined by the types of formally stated properties in the model and the types of derivations that can be supported by the properties. Composed models contain pre and postconditions expressed in the OCL and thus can support checking of behavioral properties that can be stated in terms of before and after states. The correctness properties to be verified using our approach can be derived by composing specifications of operations. As aspect and primary model sequence diagrams are composed, the operation specifications corresponding to the messages in the partially composed sequence diagram are composed in the order specified in the partially composed sequence diagram and the result is used to form a proof obligation.

If it is determined that the proof obligation obtained at a point during sequence diagram composition does not hold, then the composition can be stopped at that point. This approach allows composers to determine the point in the composition at which the property fails to hold. The information that is available when the com-



|User
|UserID: |String
|CreateSession(): |Session
|DeleteSession(|s : |Session)
|AssignRole (|r: |Role)
|DeassignRole (|r: |Role)
|b
|c
|UserAssignment
|Role
|Name: |String
|GrantPermission (|p : |Permission)
|RevokePermission (|p : |Permission)
|CheckAccess (|tar : |Target, |op : |OperationType): Boolean
|PermAssignment
|k
|l
|Permission
|CheckAccess(|tar : |Target, |op : |OperationType) : Boolean
|m
|n
|PermOp
|p
|PermTarget
|f
|Target
|OperationType
|Type: |OpTypeEnum
<<enumeration>> |OpTypeEnum
|OpType 1..*
1
|UserSessions
|a
|e
|SessionRole
|d
|Session
|AddActiveRole(|r : |Role)
|DropActiveRole(|r : |Role)
|CheckAccess(|tar:|Target, |op:|OperationType) : Boolean
|Operation(|tar:|Target 1..*, |params*): Boolean

Figure 2: Class diagram template for the core RBAC aspect model

position is stopped can be used by a developer to determine what needs to be done to correct the situation.

3. AN RBAC ASPECT MODEL

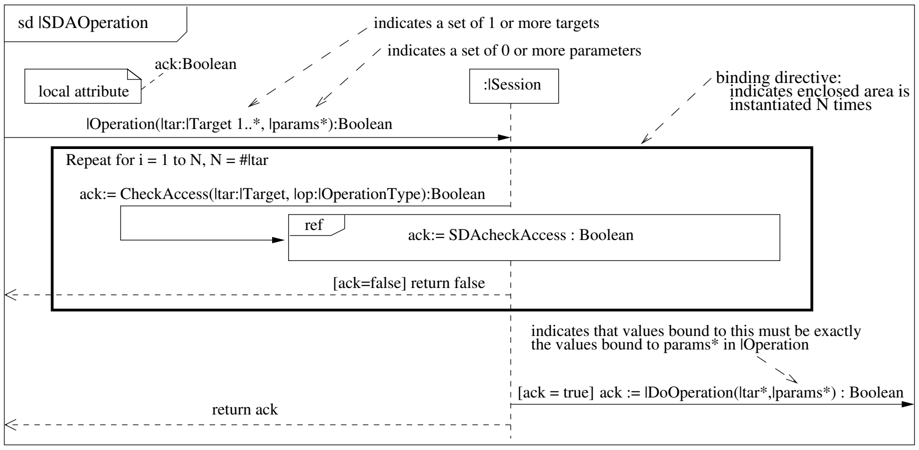
To illustrate our approach we present an aspect model for RBAC [5]. RBAC is used to protect information resources (referred to as targets) from unauthorized access. There are many variations of RBAC, each specifying and enforcing a set of access control constraints. In this paper we focus only on constraints in *Core RBAC*. Core RBAC embodies the essential aspects of RBAC, that is, the constraints are present in any RBAC application. Core RBAC consists of: (1) a set of users where a user is an intelligent autonomous agent, (2) a set of roles where a role is a job function, (3) a set of targets where a target is an entity that contains or receives information, (4) a set of operation types, where an operation describes a service provided by the application, and (5) a set of permissions where a permission is an approval to perform an operation on targets. Users are assigned to roles, roles are associated with permissions, and users acquire permissions by being members of roles. Core RBAC also includes the notion of user sessions. A user establishes a session during which he activates a subset of the roles assigned to him. Each user can activate multiple sessions, however, each session is associated with only one user. The operations that a user can perform in a session depend on the roles activated in that session and the permissions associated with those roles.

Fig. 2 shows a class diagram template in the core RBAC aspect model. In the diagram, we use the symbol "|" to indicate template parameters. A class template consists of two parts: one part consists of attribute templates that produce attributes when instantiated, and the other part consists of operation templates that produce operations when instantiated. For example, the class template *Role* contains an attribute template with a name parameter (*Name*) and three operation templates (*GrantPermission*, *RevokePermission*, and *CheckAccess*). Another class template *OperationType* contains an attribute template with a type parameter (*Type*). Instances of *Type* may be any of the user-defined enumeration literals instantiated from *OpType* which is an attribute template of the enumeration template *OpTypeEnum*.

Association templates, such as *UserAssignment* and *UserSessions* produce associations between instantiations of the class templates they connect. An association template consists of multiplicity parameters (one at each end) that yield association multiplicities (integer ranges) when instantiated. The multiplicity "1" on the *User* end of the *UserSessions* template is strict: a session can only be associated with one user.

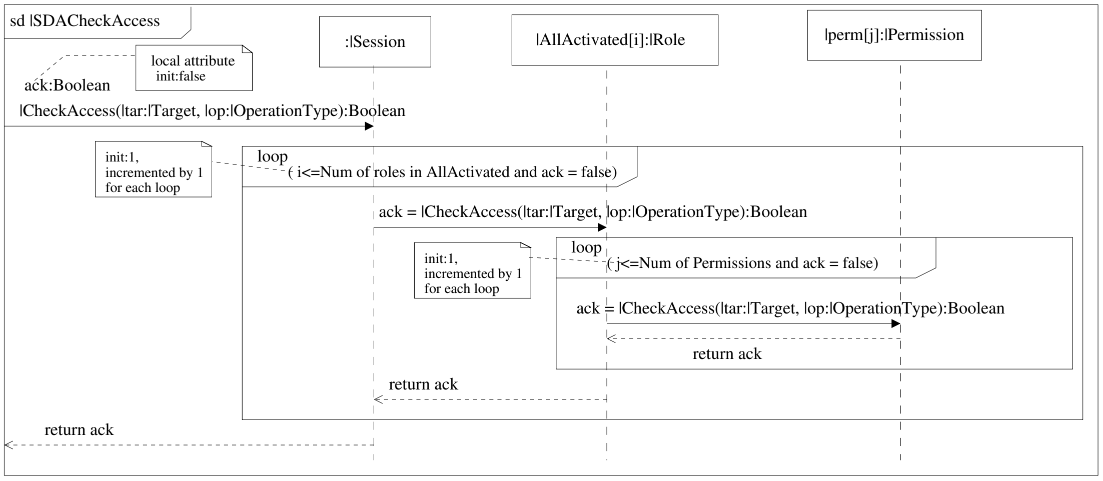
Annotated operation specification templates for *Operation* and *CheckAccess* in the *Session* template are given below. Operation specification templates can include binding directives that determine how context-specific aspect models are produced from templates when simple instantiation is not sufficient.

```
context |Session::|Operation(|tar:|Target 1..*,|params *):Boolean
−− Operation takes 1 or more tar arguments and
−− 0 or more params arguments
pre: true
−− This operation can be invoked in any state
post:
−− The operation returns true if each call to CheckAccess
−− returns true (indicating that the session has permission
−− to perform the operation on the target), and the DoOperation
−− has returned successfully, otherwise it returns false.
let Repeat for i = 1 to N; N = ♯ |tar {
    −− Repeat is a binding directive that causes elements within
    −− its scope to be repeated N times when instantiated.
    −− ♯ |tar returns the number of tar arguments.
    chkAccMsg-i:OclMessage =
        self ˆ |CheckAccess(|tar-i: |Target, |op: |OperationType),
    −− chkAccMsg-i represents the sending of the i-th
    −− CheckAccess message to itself (the session object).
    −− Each CheckAccess message invokes an operation
    −− that checks whether the session has permission
    −− to perform the operation on each target, tar.
}
−− End of Repeat block
    doOpMsg:OclMessage =
        |? ˆ |DoOperation(|tar:|Target *, |params *)
    −− doOpMsg represents the sending of the DoOperation
    −− message to an unknown object (the object is provided
    −− when the template is instantiated).
in
−− Start of constraint in postcondition:
Repeat for i = 1 to N; N = ♯ |tar {
        (chkAccMsg-i.hasReturned() and
            chkAccMsg-i.result() = true) and }
−− End of Repeat block
        (doOpMsg.hasReturned() and doOpMsg.result() = true)
−− End of Operation specification
```



sd |SDAOperation
indicates a set of 1 or more targets
indicates a set of 0 or more parameters
ack:Boolean
local attribute
:|Session
binding directive: indicates enclosed area is instantiated N times
|Operation(|tar:|Target 1..*, |params*):Boolean
Repeat for i = 1 to N, N = #|tar
ack:= CheckAccess(|tar:|Target, |op:|OperationType):Boolean
ref
ack:= SDAcheckAccess : Boolean
[ack=false] return false
indicates that values bound to this must be exactly the values bound to params* in |Operation
[ack = true] ack := |DoOperation(|tar*,|params*) : Boolean
return ack

Figure 3: |*SDAOperation* **sequence diagram template**



sd |SDACheckAccess
:|Session
|AllActivated[i]:|Role
|perm[j]:|Permission
local attribute init:false
ack:Boolean
|CheckAccess(|tar:|Target, |op:|OperationType):Boolean
init:1, incremented by 1 for each loop
loop
(i<=Num of roles in AllActivated and ack = false)
ack = |CheckAccess(|tar:|Target, |op:|OperationType):Boolean
init:1, incremented by 1 for each loop
loop
(j<=Num of Permissions and ack = false)
ack = |CheckAccess(|tar:|Target, |op:|OperationType):Boolean
return ack
return ack
return ack

Figure 4: |*SDACheckAccess* **sequence diagram template**

context |Session::|CheckAccess(|tar:|Target,
|op:|OperationType) : Boolean
pre: true
post:
−− The operation returns true if there exists an assigned
−− role that is associated with at least one permission
−− that grants the operation, op, access to the target, tar,
−− Otherwise, it returns false.
result =
self.|GetAllActiveRoles()→ exists(|r| |r.|Permission
→exists(|p||p.|Target → includes(|tar)
and |p.|OperationType → includes(|op)))
−− End of CheckAccess specification

For lack of space, we show only two sequence diagram templates in the RBAC aspect model: *SDAOperation* describes the interactions that take place when a user invokes *Operation* in *Session*, and *SDACheckAccess* describes what happens when *Session* performs the *CheckAccess* operation. The sequence diagram template *SDAOperation* (Fig. 3) describes the following pattern of behavior:
(1) A sender sends an operation call message (*Operation*(...)) to a session object.
(2) The session object checks whether the user is authorized to invoke the requested operation on each target. This check is described by the referenced sequence diagram shown in Fig. 3 (indicated by the *ref* fragment) and is performed for each target passed in as an argument to *Operation*. If the access is not authorized for a target (i.e., $ack = false$) then the *Session* object returns $false$ to the sender, indicating that access is not granted. The sequence diagram fragment enclosed by the *Repeat* box describes this pattern of behavior. The *Repeat* is a binding directive indicating that the enclosed fragment is repeated *N* times, where *N* is the number of targets given as arguments (indicated by #|*tar*).
(3) If the access is authorized, then the *Session* object requests that the operation be performed, that is, it sends a *DoOperation* message to the target object that performs the operation.

The *SDACheckAccess* sequence diagram template shown in Fig. 4 describes what happens when a *Session* object checks whether it is authorized to perform an operation on a target. Invocation of the *CheckAccess* operation in a *Session* object, results in calls to *CheckAccess* to each role activated for the session. Invocation of a role's *CheckAccess* operation results in the sending of a *CheckAccess* message to each permission associated with the role. If at least one permission returns true, then the *CheckAccess* operation in *Session* returns true; otherwise (i.e., all permissions return false) the *CheckAccess* operation in *Session* returns false.

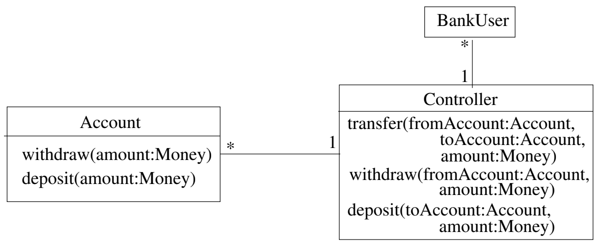

BankUser
*
1
Controller
transfer(fromAccount:Account, toAccount:Account, amount:Money)
withdraw(fromAccount:Account, amount:Money)
deposit(toAccount:Account, amount:Money)
Account
withdraw(amount:Money)
deposit(amount:Money)
*
1

Figure 5: A partial class diagram for a banking application

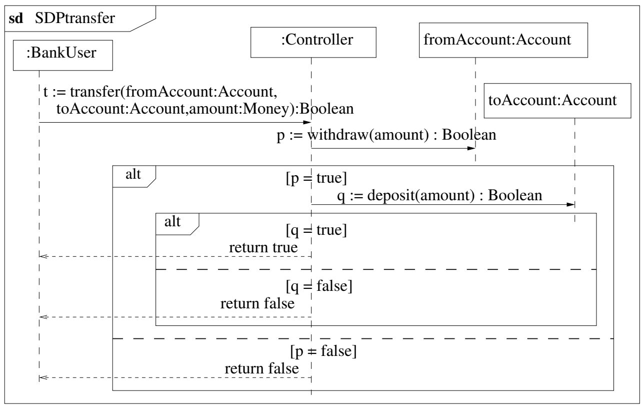


Figure 6: Sequence diagram for the *transfer* operation in a banking application

4. COMPOSING AN RBAC ASPECT WITH A PRIMARY MODEL

In this section we outline how the RBAC aspect model described in the previous section is composed with a banking application primary model.

4.1 A Banking Application Primary Model

Fig. 5 shows a partial class diagram in the banking application primary model. The application allows users to carry out *transfer*, *withdraw*, and *deposit* transactions on accounts. The class *BankUser* describes bank users, *Account* describes bank accounts, and *Controller* describes objects that coordinate transactions on bank entities (in this application, *Controller* has only one instance - the OCL constraint expressing this multiplicity restriction is not shown).

Operation specifications for *transfer*, *withdraw* and *deposit* operations are given below:

SPEC1
context Controller::transfer(fromAccount:Account,
toAccount:Account, amount:Money):Boolean
pre: true
post:
−− The message withdraw sent to fromAccount and
−− the message deposit sent to toAccount have
−− returned and their return values are true
let wdMsg:OclMessage = fromAccount^withdraw(amount),
dpMwg:OclMessage = toAccount^deposit(amount)
in result =
(wdMsg.hasReturned() and wdMsg.result() = true) and
(dpMsg.hasReturned() and dpMsg.result() = true)

SPEC2
context Account::withdraw(amount:Money):Boolean
pre: true
post:
−− If the value of balance before the execution
−− is less than the value of amount, the operation returns false,
−− otherwise, a new balance is obtained
−− by subtracting the amount from the old balance
if balance@pre >= amount
then balance = balance@pre-amount and result = true
else result = false

SPEC3
context Account::deposit(amount:Money):Boolean
pre: true
post:
−− the value of balance after execution is
−− equal to the sum of amount
−− and the value of balance before execution
balance = balance@pre + amount and result = true

We show only the primary model's sequence diagram for the *transfer* operation (see Fig. 6). The sequence diagram reflects directly the message passing specified in the postcondition of the *transfer*, *withdraw*, and *deposit* operations given previously. This style of writing operation pre- and postconditions makes it possible to generate proof obligations using the operation specifications as the sequence diagrams are composed. We illustrate how proof obligations can be generated in Section 5.

4.2 Class Diagram Composition

Composing the RBAC aspect model's class diagram and the banking application's class diagram involves instantiating the RBAC model and composing the resulting context-specific class diagram with the banking application's class diagram.

An instantiation of the class diagram template of the RBAC aspect model is shown in Fig. 7. The bindings used to instantiate the aspect model indicate where in the primary model the context-specific aspect model elements will be incorporated. For example, the bindings, (*BankUser*, |*User*), (*Account*, |*Target*), indicate that the instantiated *User* template class in the aspect model is to be merged with the *BankUser* class in the primary model, and the instantiated *Target* template class in the aspect model is to be merged with the *BankTarget* class in the primary model. The instantiations of class templates *Role* (*BankRole*), and *Session* (*BankSession*) are new model elements that are to be included in the composed model. The *Operation* template in the *Session* class template is instantiated three times to produce the *transfer*, *withdraw* and *deposit* operations in *BankSession*. The enumeration values in *TransactionTypeEnum* (*TRANSFER*, *WITHDRAW*, and *DEPOSIT*) are instantiations of an attribute template *OpType* shown in Fig. 2. The operation specification template associated with the *Operation* template is also instantiated for each of these operations. For example, the *transfer* operation in *BankSession* class of the aspect model is associated with the following instantiation of the *Operation* specification template (only part of the specification is shown; incomplete parts are indicated by ...):

context BankSession::transfer(fromAccount:Account,
toAccount:Account,amount:Money):Boolean
pre: true
post:
let
−− Statement in Repeat block of template is instantiated
−− twice because there are two targets
−− in the argument: fromAccount and toAccount.

```
    chkAccMsg-1:OclMessage =
        self^checkAccess(fromAccount, TRANSFER),
    chkAccMsg-2:OclMessage =
        self^checkAccess(toAccount, TRANSFER),
    doOpMsg:OclMessage =
        Controller^transfer(fromAccount, ...)
in result =
    (chkAccMsg-1.hasReturned() and chkAccMsg-1.result()=true)
    and
    (chkAccMsg-2.hasReturned() and chkAccMsg-2.result()=true)
    and
    (doOpMsg.hasReturned() and doOpMsg.result() = true)
```

Instantiation of the *CheckAccess* specification template produces the following specification for *checkAccess* operation in *BankSession*:

```
context BankSession::checkAccess(tar:Account,
            op:TransactionType):Boolean
pre: true
post: result =
    self.BankRole→exists (r |
        r.Permission→exists(p|p.Account→includes(tar)
        and p.TransactionType→includes(op)))
```

Incorporating the access control behavior into the banking application requires that the *transfer* operation specification in the primary model's *Controller* class (see *SPEC1* in Section 4.1) be modified so that the calls to the *withdraw* and *deposit* operations are authorized before being sent to the target accounts. The needed modifications are defined by composition directives that replace calls to the operations in target accounts by calls to the *withdraw* and *deposit* operations in *BankSession*. The result is the following operation specification that is associated with the operation in the composed model:

```
context Controller::transfer(fromAccount:Account,
            toAccount:Account, amount:Money):Boolean
pre: true
post:
let wdMsg:OclMessage =
        bankSession^withdraw(fromAccount,amount),
    dpMwg:OclMessage =
        bankSession^deposit(toAccount,amount)
in result =
    (wdMsg.hasReturned() and wdMsg.result()=true and
    dpMsg.hasReturned() and dpMsg.result()=true)
```

The result of composing the aspect model's class diagram and the primary model's class diagram is shown in Fig. 8. The basic class diagram composition procedure merges classes with the same name and includes elements that appear in primary or aspect class diagram but not in the other. A detailed description of class diagram composition that uses a more complex example is given in France *et al.* [6].

4.3 Sequence Diagram Composition

The *SDAOperation* sequence diagram template is instantiated three times to produce context-specific sequence diagrams corresponding to the *BankSession* operations *transfer*, *withdraw* and *deposit*. The three sequence diagrams produced from the template are named *SDAtransfer*, *SDAwithdraw*, and *SDAdeposit*. These sequence diagrams are given in the Appendix of this paper together with *SDAcheckAccess* which is the sequence diagram instantiated from *SDACheckAccess* sequence diagram template.

Fig. 9 gives an overview of how the sequence diagram describing the *transfer* operation in the primary model is composed with the sequence diagrams in the context-specific aspect model. Composition should result in a behavior in which calls to *transfer withdraw* and *deposit* operations are carried out only if the *BankSession* object is permitted to carry out the requested operations on the target accounts. The composition procedure that accomplishes this (defined by composition directives not given in this paper) performs the following steps (the numbers shown in Fig. 9 correspond to the steps given below):
(1) The initiating *transfer* message in the primary model sequence diagram is rerouted to the *BankSession* object and the access control behavior described by the *SDAtransfer* sequence diagram is inserted.
(2) If access is granted as a result of carrying out the behavior described by *SDAtransfer*, the *transfer* operation in the *Controller* can be invoked. To reflect this, a composition directive is used to add a *transfer* operation call message directed to the *Controller*. The result of steps (1) and (2) describes a situation in which the *transfer* operation call is intercepted by *SDAtransfer* and passed on to the *Controller* object only if access is granted.
(3) The call to the *withdraw* operation made by the *Controller* during the invocation of the *transfer* operation is intercepted by the *SDAwithdraw* sequence diagram.
(4) If access is granted then a *withdraw* operation call is sent to the account, *fromAccount*.
(5) The call to the *deposit* operation made by the *Controller* during the invocation of the *transfer* operation is intercepted by the *SDAdeposit* sequence diagram.
(6) If access is granted then a *depsoit* operation call is sent to *toAccount*.

5. VERIFYING ACCESS CONTROL PROPERTIES: AN EXAMPLE

In this section we illustrate how composition of the aspect and primary model sequence diagrams can be carried out in a verifiable manner. A desired property of the *transfer* behavior in the composed model is specified and proof obligations are generated as the *SDPtransfer* sequence diagram in the primary model (Fig. 6) is composed with the *SDAtransfer*, *SDAwithdraw*, *SDAdeposit* sequence diagrams in the context-specific aspect model (Fig. 10, Fig. 11, Fig. 12 in the Appendix). The approach requires that operation specifications reference the interactions that take place in corresponding interaction diagrams, that is, they must state the conditions under which messages are sent by the operations.

The behavioral properties used in our approach are those that can be verified by discharging an implication involving operation specifications. Specifically, the proof obligations have the form *P1 implies P2*, where *P1* is the specification of a behavior in the composed model and *P2* is the property to be verified. As the sequence diagrams pertaining to the behavior being analyzed are composed, the proof obligation is evolved by taking into account any new information available in each step of the composition.

The property to be verified during the *transfer* can be stated as follows: *If the transfer operation is authorized on the specified accounts, then, if the source account has enough funds to cover the transfer amount then the funds will have been transferred by the time the transfer operation terminates.*

We express the above, using an extended form of the OCL, as follows:

```
contextBankSession::transfer(fromTarget:Account,
            toTarget:Account, amount:Money):Boolean
```

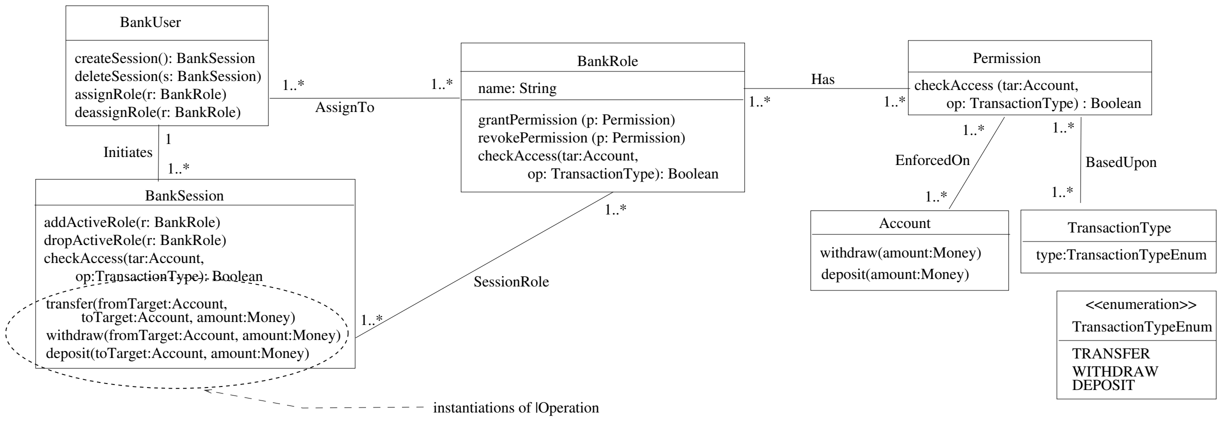

BankUser
createSession(): BankSession
deleteSession(s: BankSession)
assignRole(r: BankRole)
deassignRole(r: BankRole)
BankRole
name: String
grantPermission (p: Permission)
revokePermission (p: Permission)
checkAccess(tar:Account, op: TransactionType): Boolean
Permission
checkAccess (tar:Account, op: TransactionType) : Boolean
Has
AssignTo
Initiates
EnforcedOn
BasedUpon
SessionRole
BankSession
addActiveRole(r: BankRole)
dropActiveRole(r: BankRole)
checkAccess(tar:Account, op:TransactionType): Boolean
transfer(fromTarget:Account, toTarget:Account, amount:Money)
withdraw(fromTarget:Account, amount:Money)
deposit(toTarget:Account, amount:Money)
Account
withdraw(amount:Money)
deposit(amount:Money)
TransactionType
type:TransactionTypeEnum
<<enumeration>>
TransactionTypeEnum
TRANSFER
WITHDRAW
DEPOSIT
instantiations of |Operation
1..*
1

Figure 7: A context-specific RBAC class diagram

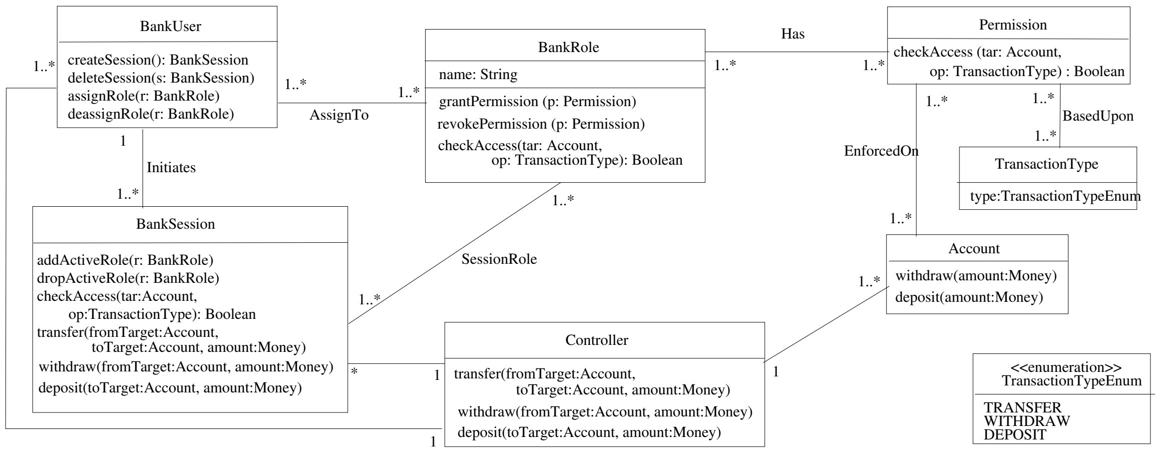

BankUser
createSession(): BankSession
deleteSession(s: BankSession)
assignRole(r: BankRole)
deassignRole(r: BankRole)
BankRole
name: String
grantPermission (p: Permission)
revokePermission (p: Permission)
checkAccess(tar: Account, op: TransactionType): Boolean
Permission
checkAccess (tar: Account, op: TransactionType) : Boolean
Has
AssignTo
Initiates
EnforcedOn
BasedUpon
SessionRole
TransactionType
type:TransactionTypeEnum
BankSession
addActiveRole(r: BankRole)
dropActiveRole(r: BankRole)
checkAccess(tar:Account, op:TransactionType): Boolean
transfer(fromTarget:Account, toTarget:Account, amount:Money)
withdraw(fromTarget:Account, amount:Money)
deposit(toTarget:Account, amount:Money)
Controller
transfer(fromTarget:Account, toTarget:Account, amount:Money)
withdraw(fromTarget:Account, amount:Money)
deposit(toTarget:Account, amount:Money)
Account
withdraw(amount:Money)
deposit(amount:Money)
<<enumeration>>
TransactionTypeEnum
TRANSFER
WITHDRAW
DEPOSIT
1..*
1
*

Figure 8: Class diagram of the composed model

```
verify TransferProp:
let (chkAccMsg-1:OclMessage =
        self^checkAccess(fromAccount, TRANSFER),
    chkAccMsg-2:OclMessage =
        self^checkAccess(toAccount, TRANSFER),
    successful-transfer =
        (if fromAccount.balance@pre >= amount
        then (fromAccount.balance =
                fromAccount.balance@pre-amount
            and toAccount.balance =
                toAccount.balance@pre + amount)))
in
    if (chkAccMsg-1.hasReturned()
        and chkAccMsg-1.result()=true
        and chkAccMsg-2.hasReturned()
        and chkAccMsg-2.result()=true)
    then successful-transfer
```

In the above, the property to be verified is specified in the context of the $transfer$ operation in the *BankSession* class. We introduce the $verify$ construct to the OCL syntax to support the specification of properties to be verified. The OCL statement in the $verify$ section states the property to be verified. The property is named *TransferProp*.

One approach to verifying *TransferProp* is to carry out the composition and then establish that the postcondition of the $transfer$ operation in the composed model's *BankSession* class implies *TransferProp*. The approach described in this section uses a more incremental approach in which a proof obligation is evolved during the composition. The point in the composition where the generated obligation does not hold can yield insights about the sources of the incorrect behaviors.

In what follows we illustrate how a proof obligation for the *TransferProp* property evolves during composition. The property does not hold for the composed model and we will show how this can be revealed during composition.

In steps (1) and (2) of the sequence diagram composition described in Section 4.3, the *SDAtransfer* sequence diagram is incorporated into the primary model's $transfer$ sequence diagram as shown in Fig. 9. At this point, the proof obligation can be expressed as an implication *P1 implies TransferProp*, where $P1$ specifies the condition under which the $transfer$ operation in the *BankSession* object returns true. The postcondition for $transfer$ is repeated below:

```
result =
((chkAccMsg-1.hasReturned() and chkAccMsg-1.result()=true)
    and (chkAccMsg-2.hasReturned() and
            chkAccMsg-2.result()=true)
```

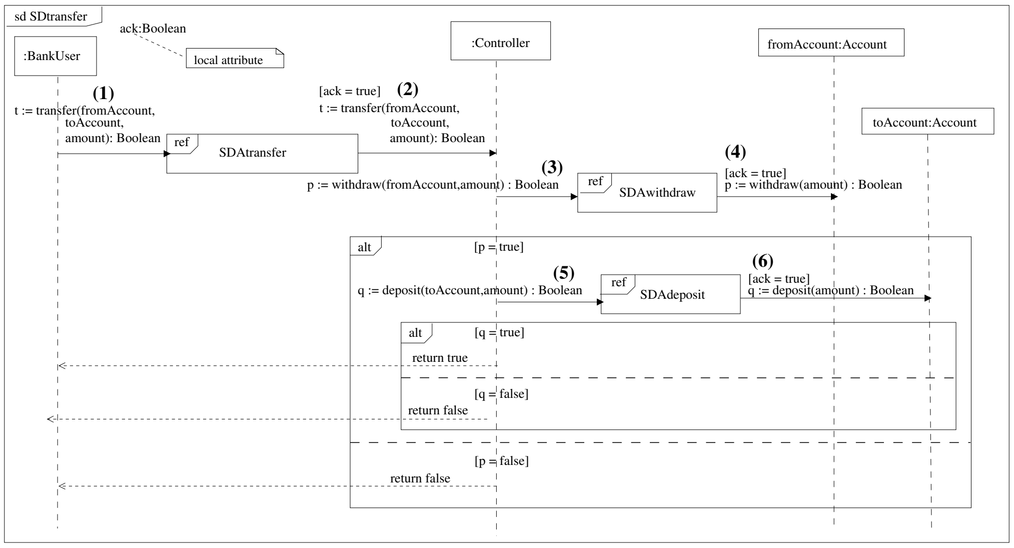


Figure 9: Overview of sequence diagram composition

and (doOpMsg.hasReturned() and doOpMsg.result() = true))

The proof obligation (obtained after simplification) is given below:

−− Proof Obligation PF1
(doOpMsg.hasReturned() and doOpMsg.result() = true)−−(DP)
implies successful-transfer

Discharging PF1 requires information about the conditions under which the condition labeled (DP) in PF1 holds, that is, the conditions under which the *transfer* operation in the *Controller* object (called by the *doOpMsg* message) returns true. This observation lead to a modified proof obligation in which the condition labeled (DP) is replaced by the part of the postcondition that determines when the *transfer* operation in *Controller* returns true. The resulting proof obligation is given below:

−− Proof Obligation PF2
(wdMsg.hasReturned() and wdMsg.result()=true and
  dpMsg.hasReturned() and dpMsg.result()=true) −− (WD)
implies successful-transfer

Discharging proof obligation PF2 requires determining the conditions under which the condition labeled by (WD) holds, that is, the conditions under which the *withdraw* operation in *BankSession* returns true. As was done in the previous steps, the proof obligation is modified by replacing (WD) by the relevant part of the *withdraw* postcondition. This process is continued until a proof obligation that does not hold is produced or until the sequence diagram composition is completed. In this case, a proof obligation that does not hold is obtained after incorporating the *SDAwithdraw* sequence diagram into the primary model's sequence diagram (i.e., after step (4) of the composition). The proof obligation is given below:

```
−− Proof Obligation PF4
let chkAccMsg3:OclMessage =
      self^checkAccess(fromAccount, withdraw),
   doOpMsg:OclMessage =
      fromAccount^withdraw(amount:Money)
in chkAccMsg3.hasReturned() and
   chkAccMsg3.result()= true and
   (if fromAccount.balance@pre >= amount
   then fromAccount.balance =
         fromAccount.balance@pre-amount) and
   dpMsg.hasReturned() and dpMsg.result()=true
implies successful-transfer
```

At this point an inspection of PF4 would reveal that the condition does not hold because of the presence of the access control behavior that checks whether access to the *withdraw* operation is granted. If access to the *withdraw* operation is not granted, then the obligation does not hold. There is no guarantee that this case will never happen (i.e., there are no constraints in the model that preclude this case). At this point the composition can be stopped knowing that it will produce a model that does not have the required property.

This problem can be fixed by incorporating only the *SDAtransfer* (i.e., steps (1) and (2)) sequence diagram during the composition. The result is that the check access operation is only carried out on the *transfer* operation, not on the *withdraw* and *deposit* operations. Another solution is to guarantee access to the *withdraw* and *deposit* operations whenever access is granted to a *transfer* operation by including an invariant on permission objects that precludes the above situation in which the obligation failed to hold.

6. RELATED WORK

Aspect-oriented programming (AOP) supports separation of concerns at the programming level (e.g., see [11, 17]). Researchers have started to address the problem of defining and composing aspects at a higher level of abstraction (e.g., see [4, 9]). Clarke *et al.* [4] propose that a design can be created by synthesizing subjects, where a subject describes how a single required feature is realized in a design. Subjects are expressed as UML model views. Their approach does not currently support verifiable composition of behaviors.

Tidswell and Jaeger [22] propose an approach to visualizing access control constraints. They point out the need for visualizing constraints and the limitations of previous work (e.g., [1, 15, 16]) on expressing constraints. Another effort to graphical specification of RBAC is proposed by Koch *et al.* [13]. In their approach, RBAC policies are represented by graph transformations. A graph consists of nodes and edges. Nodes represent notions such as users and roles. Edges represent relationships between notions. Transformation rules are defined for administration activities such as adding a user to a role and removing a user from a role. Consistency properties such as DSD constraints are also specified graphically. Verification of RBAC policies is carried out by showing that graphical constraints do not occur in the graph specifying RBAC policies. The drawback of these two approaches is that they created a new notation for specifying constraints and it is not clear how the new notation can be integrated with other widely-used design notations. The approach described in this paper utilizes notations from a standardized modeling language and also integrates the policy specification activity with design modeling activities.

There has been some work on using the UML to model security concerns (e.g., see [2, 3, 10, 14]. Chan and Kwok [3] model a design pattern for security that addresses asset and functional distribution, vulnerability, threat, and impact of loss. UML stereotypes identify classes that have particular security needs due to their vulnerability either as assets or as a result of functional distribution. Lodderstedt *et al.* [14] propose SecureUML and define a vocabulary for annotating UML-based models with information relevant to access control. It is based on the model for basic RBAC with support for role hierarchies. An access control policy is realized mainly by using declarative access control. This means that the access control policy is configured in the deployment descriptors of an EJB component. Jürjens [10] model security mechanisms based on the multi-level classification of data in a system using an extended form of the UML called UMLsec. UMLsec is fully described in a UML profile. These approaches mainly focus on extending the UML notation to better reflect security concerns. The approach described in this paper tackles the complementary task of capturing RBAC policies in patterns that can be reused by developers of secure systems.

The proposed aspect modeling approach builds upon the notation and techniques described in our earlier work. [8] shows how security concerns can be localized and then composed with models of system functionality and [12, 19] presents how invalid structures can be captured and expressed using object diagram templates. [6] extends [7, 8] by refining the aspect modeling notation and instantiation process, and refining the notion of composition directives to support resolution and modeling of solution variants. This paper is an extension of [12, 18, 19] in that it illustrates how composition can be carried out in a verifiable manner.

7. CONCLUSION AND FUTURE WORK

In this paper, we give an overview of how AOM can be used to support separation of access control features from other application features. Access control features are described by aspect models which are composed with primary models to produce an application design with access control features. The paper also describes how verifiable composition of access control behaviors can be supported.

The verifiable composition technique was carried out systematically, but manually. The systematic approach is made possible by restricting the form of properties that can be verified. Given an initial proof obligation, its modification during the sequence diagram composition process can be mechanized since it essentially involves replacing specified parts of the proof obligations with parts of operation specifications. We are currently developing tool support for composition that will include support for generating proof obligations. We are also investigating ways of integrating existing proof tools that can be used to assist in the discharge of proof obligations.

The example used to illustrate this approach is a simple banking application. We are currently working on a case study in the military domain that involves a complex primary model and more than one aspect model.

8. ACKNOWLEDGMENTS

This material is based upon work funded by AFOSR under Award No. FA9550-04-1-0102.

9. REFERENCES

[1] G. Ahn and R. Sandhu. The RSL99 Language for Role-Based Separation of Duty Constraints. In *Proceedings of ACM Workshop on Role-Based Access Control*, pages 43–54, 1999.

[2] G. J. Ahn and M. Shin. Role-based authorization constraints specification using object constraint language. In *Proc. of the 10th Int'l. Workshop on Enabling Technologies*, Camebridge, MA, June 2001.

[3] M. T. Chan and L. F. Kwok. Integrating security design into the software development process for e-commerce systems. *Information Management and Computer Security*, 9(2-3):112–122, 2001.

[4] S. Clarke. "Extending Standard UML with Model Composition Semantics". *Science of Computer Programming*, 44(1):71–100, July 2002.

[5] D. Ferraiolo, R. Sandhu, S. Gavrila, and D. R. K. an d R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3), August 2001.

[6] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 151(4), August 2004.

[7] G. Georg, R. France, and I. Ray. An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, Dresden, Germany, 2002.

[8] G. Georg, I. Ray, and R. France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.

[9] G. Georg, R. Reddy, and R. France. Specifying cross-cutting requirements concerns. In *Proceedings of the International Conference on the UML, October 2004*. Springer, 2004.

[10] J. Jurjens. Towards development of secure sytems using umlsec. In *Proc. of the 4th Int'l. Conf. on Fundamental Approaches to Software Engineering*, pages 187–200, Genova, Italy.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[12] D.-K. Kim, I. Ray, R. France, and N. li. Modeling role-based access control using parameterized UML models. In *Proceedings of the 7th Conference on Fundamental Approaches to Software Engineering (FASE 2004)*, 2004.

[13] M. Koch, L. V. Mancini, and F. Parisi-Presicce. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, 2002.

[14] T. Lodderstedt, D. Basin, and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In

5th Int'l. Conf. on the Unified Modeling Language, 2002, 2002.

[15] M. Nyanchama and S. Osborn. The Role Graph Model and Conflict of Interest. *ACM Transactions on Information Systems Security*, 2:3–33, 1999.

[16] S. Osborn and Y. Guo. Modelling Users in Role-Based Access Control. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 31–37, Berlin, Germany, July 2000.

[17] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, Oct. 2001.

[18] I. Ray, R. France, N. Li, and G. Georg. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, 40(9):557–633, 2004.

[19] I. Ray, N. Li, R. France, and D.-K. Kim. Using UML to visualize role-based access control constraints. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT)*, 2004.

[20] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. Bieman. Model composition directives. In *Proceedings of the International Conference on the UML, October 2004*. Springer, 2004.

[21] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0, Final Adopted Specification, OMG, http://www.omg.org, August 2003.

[22] J. E. Tidswell and T. Jaeger. An Access Control Model for Simplifying Constraint Expression. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 154–163, Athens, Greese, November 2000.

[23] J. Warmer and A. Kleppe. *The Object Constraint Language, Second Edition*. Addison-Wesley, 2003.

APPENDIX

Sequence Diagrams

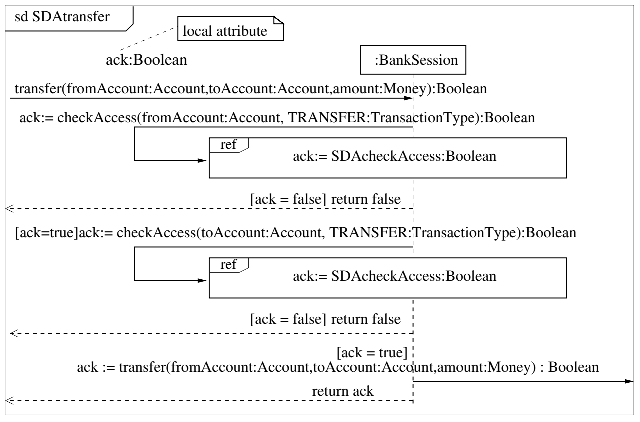

sd SDAtransfer
local attribute
ack:Boolean
:BankSession
transfer(fromAccount:Account,toAccount:Account,amount:Money):Boolean
ack:= checkAccess(fromAccount:Account, TRANSFER:TransactionType):Boolean
ref
ack:= SDAcheckAccess:Boolean
[ack = false] return false
[ack=true]ack:= checkAccess(toAccount:Account, TRANSFER:TransactionType):Boolean
ref
ack:= SDAcheckAccess:Boolean
[ack = false] return false
[ack = true]
ack := transfer(fromAccount:Account,toAccount:Account,amount:Money) : Boolean
return ack

Figure 10: SDAtransfer: Context-specific sequence diagram for the *transfer* operation in *BankSession*

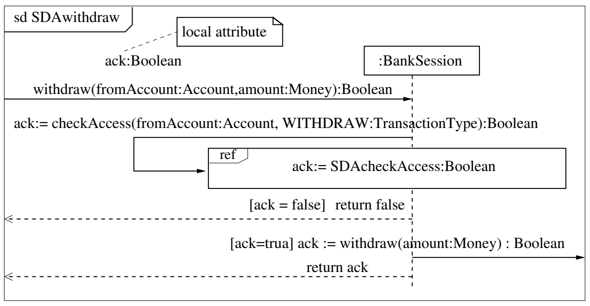

sd SDAwithdraw
local attribute
ack:Boolean
:BankSession
withdraw(fromAccount:Account,amount:Money):Boolean
ack:= checkAccess(fromAccount:Account, WITHDRAW:TransactionType):Boolean
ref
ack:= SDAcheckAccess:Boolean
[ack = false] return false
[ack=trua] ack := withdraw(amount:Money) : Boolean
return ack

Figure 11: SDAwithdraw: Context-specific sequence diagram for the *withdraw* operation in *BankSession*

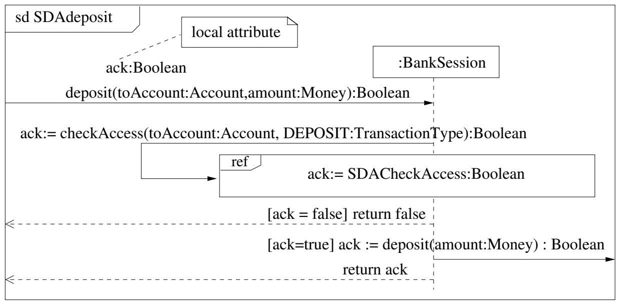

sd SDAdeposit
local attribute
ack:Boolean
:BankSession
deposit(toAccount:Account,amount:Money):Boolean
ack:= checkAccess(toAccount:Account, DEPOSIT:TransactionType):Boolean
ref
ack:= SDACheckAccess:Boolean
[ack = false] return false
[ack=true] ack := deposit(amount:Money) : Boolean
return ack

Figure 12: SDAdeposit: Context-specific sequence diagram for the *deposit* operation in *BankSession*

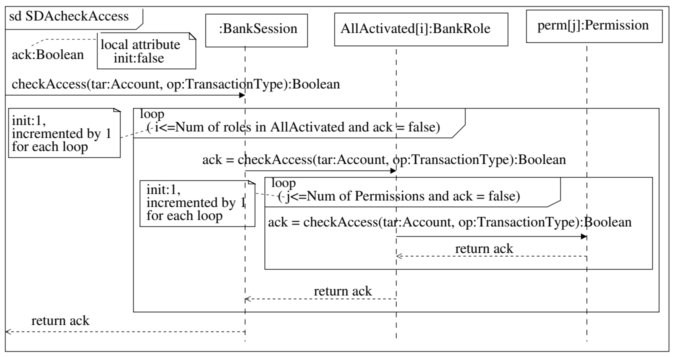

sd SDAcheckAccess
:BankSession
AllActivated[i]:BankRole
perm[j]:Permission
local attribute
init:false
ack:Boolean
checkAccess(tar:Account, op:TransactionType):Boolean
init:1,
incremented by 1
for each loop
loop
(i<=Num of roles in AllActivated and ack = false)
ack = checkAccess(tar:Account, op:TransactionType):Boolean
init:1,
incremented by 1
for each loop
loop
(j<=Num of Permissions and ack = false)
ack = checkAccess(tar:Account, op:TransactionType):Boolean
return ack
return ack
return ack

Figure 13: SDAcheckAccess: Context-specific sequence diagram for the *checkAccess* operation in *BankSession*