DISSERTATION

AN ASPECT-BASED APPROACH TO MODELING ACCESS CONTROL POLICIES

Submitted by

Eunjee Song

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2007

COLORADO STATE UNIVERSITY

August 10, 2006

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY EUNJEE SONG ENTITLED AN ASPECT-BASED APPROACH TO MODELING ACCESS CONTROL POLICIES BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Committee Member: Dr. James M. Bieman

Committee Member: Dr. Sudipto Ghosh

Committee Member: Dr. Joon K. Kim

Adviser: Dr. Robert B. France

Co-Adviser: Dr. Indrakshi Ray

Department Head: Dr. L. Darrell Whitley

ABSTRACT OF DISSERTATION

AN ASPECT-BASED APPROACH TO MODELING ACCESS CONTROL POLICIES

Access control policies determine how sensitive information and computing resources are to be protected. Enforcing these policies in a system design typically results in access control features that crosscut the dominant structure of the design (that is, features that are spread across and intertwined with other features in the design). The spreading and intertwining of access control features make it difficult to understand, analyze, and change them and thus complicate the task of ensuring that an evolving design continues to enforce access control policies.

Researchers have advocated the use of aspect-oriented modeling (AOM) techniques for addressing the problem of evolving crosscutting features. This dissertation proposes an approach to modeling and analyzing crosscutting access control features. The approach utilizes AOM techniques to isolate crosscutting access control features as patterns described by aspect models. Incorporating an access control feature into a design involves embedding instantiated forms of the access control pattern into the design model. When composing instantiated access control patterns with a design model, one needs to ensure that the resulting composed model enforces access control policies. The approach includes a technique to verify that specified policies are enforced in the composed model.

The approach is illustrated using two well-known access control models: the Role-Based Access Control (RBAC) model and the Bell-LaPadula (BLP) model. Features that enforce RBAC and BLP models are described by aspect models. We show how the aspect models can be composed to create a new hybrid access control aspect model. We also show how one can verify that composition of a base (primary) design model and an aspect model that enforces specified policies produces a composed model in which the policies are still enforced.

Eunjee Song
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Spring 2007

ACKNOWLEDGEMENTS

I am greatly indebted to many people for this dissertation. It would not have been possible to complete without their support, help, and patience. I would like to express my gratitude to them for their part in completion of the dissertation.

First, I would like to thank my adviser, Dr. Robert France, for his continued guidance and encouragement, for sharing many of his insights, and for uncountable discussions and revisions of this dissertation. He has shown to me tireless patience and helping so that I may develop my confidence even when I got lost. Second, I want to thank my co-adviser, Dr. Indrakshi Ray for her insightful comments on my work and her help she provided while I was struggling to grab new concepts in software security. She also has been a great counselor for my career as well.

I would like to extend my sincere thanks to Dr. Bieman for his help in my research and his continuous encouragement on my teaching as well. I know it was not possible for me to instruct a distance learning course for three consecutive semesters and to have the honor of Anita Reed Award later without his solid support on my teaching. I am also grateful to other members in my committee; Dr. Sudipto Ghosh for his insightful comments on writing my past publications as well as the crafting of this dissertation; Dr. Joon Kim for his unique comments from non-computer scientist's viewpoint. I would like to offer special thanks to Dr. Carolyn Schauble for providing an understanding ear every time I needed it most. She has been a wonderful mentor and a good friend for my family since we first came to the department.

Thanks to all friendly Computer Science department staffs, especially Wayne Trzyna, Sharon Van Gorder, Carol Calliham, and a former staff Susan Short, for granting me a favor all the time. Thanks to all my colleagues in the Computer Science department, in the past and at present, Dr. Kyungim Baek, Dr. Dae-kyoo Kim, Dr. Raghu Reddy, Devon Simmonds, Dinh Trung, Dr. Emanuel Grant, Eunjong Kim, and Yingchun Cai, for all their support and encouragement that kept me going. I would like to express a special gratitude to Dae-gon Kim who generously agreed to help with my final submission of dissertation. I thank members in the First Korean Church of Fort Collins for all their prayers. My special thanks go to Dr. Sohyun Kwon, Elizabeth Kim, Heekyeong Jung and all other choir members for their continuous support with prayers. And also thank you to all my friends whose names are too numerous to mention.

To my parents, I thank you for your immeasurable support and encouragement. I could not have made this accomplishment done without your support with prayer. I miss you Dad and wish you could be here with us at this time. Another special thanks to my parents-in-law; I know you keep praying for us although I cannot talk with you any more. To all my sisters and brothers, Hongji, Soojee, Sangheon and Bangheon, I thank you all for your cheering me up and giving continuous cares for mom and dad while I have been staying far from them for years. I also thank my sisters-in-law, Youngin and Wonsun, who also have prayed for my family.

My deepest gratitude is saved for my family. I cannot find words to express the depth of my gratitude to my daughter, Seoyoung, and my husband, Hanil, for putting their work and lives on hold while I pursued my goal of earning a Ph.D. Without your love, sacrifice, and endurance, this work would not have been possible to finish. I love you so much.

Finally, I praise my Lord who always sustains me and walks with me every time I am in trouble. I know you have always guided me to this accomplishment and will also show where I will have to go.

DEDICATION

This dissertation is dedicated

to my daughter Seoyoung,

to my husband Hanil,

and to my parents.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Organizations specify and enforce access control policies to protect their valuable information and resources from malicious attacks. These access control policies determine, for example, (1) what programs or persons can access protected resources, (2) under what circumstances these access rights are granted, and (3) what information items and services are made available to authorized users. Developers of applications that must enforce access control policies have to answer the following questions:

- How can one check that an access control feature adequately enforces the policies as intended?
- How can one check that an access control feature interacts with other features to produce behaviors that satisfy specified properties?

The research in this dissertation provides answers to the above questions.

1.1 Research Motivation

Modularization of design in terms of key functional features[1] results in the intertwining of access control features with other features. For example, a decision to modularize the

[1]In this dissertation, a feature is a logical unit of behavior.

design of a banking system in terms of modules that encapsulate banking services (e.g., withdraw and deposit services) results in the scattering of access control design elements across the modules containing resources that are to be protected from unauthorized accesses. In general, crosscutting access control features are problematic for the following reasons:

- Understanding a crosscutting feature is difficult because its description is scattered across a design.
- Changing the crosscutting feature requires making consistent changes in a number of places in the design.
- Evaluating alternative forms of the feature is challenging because it is difficult to replace the feature with an alternative.

The above problems complicate the following tasks:

- **Verifying policy enforcement**: Rigorously establishing that a design enforces specified access control policies requires checking that the design elements describing the access control feature enforce the policies. If the elements are scattered across a design and tangled with other design elements then verification of the policy enforcement becomes difficult.
- **Integrating policies**: Organizations may need to share their access controlled resources with other organizations. This can occur when the organizations need to collaborate on a project or when one organization is acquired by another. In these cases the issue of composing independently developed policies needs to be addressed. However, determining how the features are to be integrated so that they may enforce target policies is difficult when they are intertwined with other application features.

Aspect-oriented modeling (AOM) techniques have been advocated as solutions for making the above tasks easier by isolating crosscutting access control features from other features in the design. However, when isolated access control features are composed with a design, one needs to ensure that the result of composition still enforces the target policy. The goal of this research is to provide a technique for verifying that a design produced by composing access control features and other design features enforces specified policies and has other stated properties.

1.2 Research Overview

We utilize AOM techniques proposed by France *et al.* [31] to localize descriptions of crosscutting access control features in aspect models. In the AOM approach, a design consists of *aspect* models and a *primary* model. Each aspect model describes a crosscutting design feature that addresses a single concern [69]. The primary model describes the core functionality that determines the dominant design structure. An application design in which the concerns are addressed is created by composing the aspect models with the primary model. Composition directives are used to help ensure that the composition produces a desired model.

In this work, access control features that are to be incorporated into a software design (the primary model) are described by *aspect models*. We use Unified Modeling Language (UML) model template notation that is a specialization of the Role-Based Modeling Language (RBML)[2][30] to express aspect models. The techniques described in this dissertation are specifically targeted to designs expressed using the UML, a popular, standard modeling language.

The following techniques are developed in this dissertation: (1) A technique for verifying that an aspect model enforces the given policy, (2) a technique for integrating aspect models to describe different access control mechanisms, and (3) a technique for composing

[2]RBML is a subset of the UML [90] with semantics that support rigorous analysis of policy models

an aspect model and a primary model in a verifiable manner.

1.2.1 Verifying Policy Enforcement in an Access Control Aspect Model

Fig. 1.1 shows our approach to verifying that an access control aspect enforces target access control policies. A *policy model* is a UML description of an access control policy.

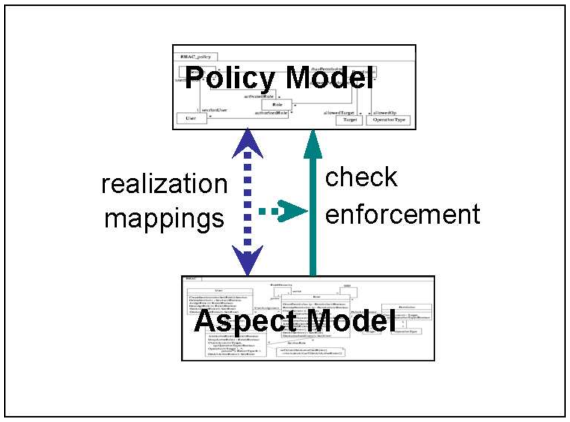


Figure 1.1: Checking policy enforcement in an aspect model

An *aspect model* describes an access control feature. Realization mappings describe how elements in the policy model are realized in the aspect model[3]. Using these mappings we transform the policy constraints defined in a policy model into constraints expressed in terms of concepts in an aspect model. An access control aspect model enforces policies in a policy model if transformed policy constraints hold in an aspect model.

[3]Realization in our work is an abstraction relationship between two sets of model elements, one representing a policy model (policy requirements) and the other represents an aspect model (a set of designs that enforce the given policy model)

1.2.2 Composing Access Control Aspect Models

Complex applications are typically required to enforce multiple access control policies. Instead of composing the aspects one at a time with a primary model, the aspects can be composed *a priori* to create a composed aspect. One advantage of doing this is that it provides a view in which the interactions across the aspects can be understood in isolation. A second advantage is that it reduces the number of times the composition process must be applied, especially when the composed aspect is used in several applications.

In this work, we provide a method for using composition directives to obtain an integrated access control aspect. Fig. 1.2 shows our approach to using composition directives

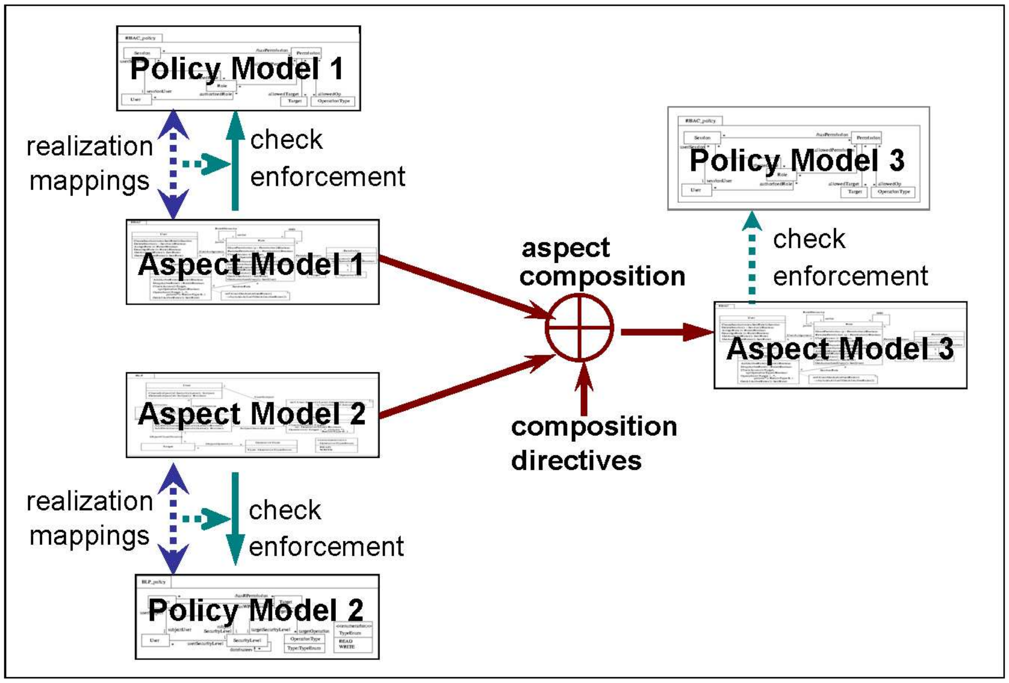


Figure 1.2: Integrating two aspect models to create a composed aspect model that enforces an integrated policy

for integrating two access control aspects to produce an aspect model that enforces a policy (Policy Model 3) that is an integration of the access control policies (Policy Model 1 & 2) enforced by the two access control aspects. Composition directives are used to help one create a composed aspect model that enforces the integrated policy. The developer uses his/her knowledge of how the policies are to be integrated to select appropriate composition directives that will be used to compose the aspect models. The developer is responsible for ensuring that the result enforces the integrated policies. The enforcement technique proposed in this dissertation can be used for this purpose. The approach will be demonstrated using an example from the military domain which requires applications that enforce the integration of two well-known access control policies; Role-Based Access Control (RBAC) and Bell-LaPadula (BLP) policies.

1.2.3 Composing an Access Control Aspect Model and a Primary Model in a Verifiable Manner

Composing an aspect model with a primary model can result in conflicts or compromised behaviors. Therefore, a key issue in applying the AOM approach is determining whether composition of an aspect model and a primary model produces a composed model that has desired properties. To address this issue we provide an approach that allows one to check that the result of a composition is correct with respect to formally stated properties.

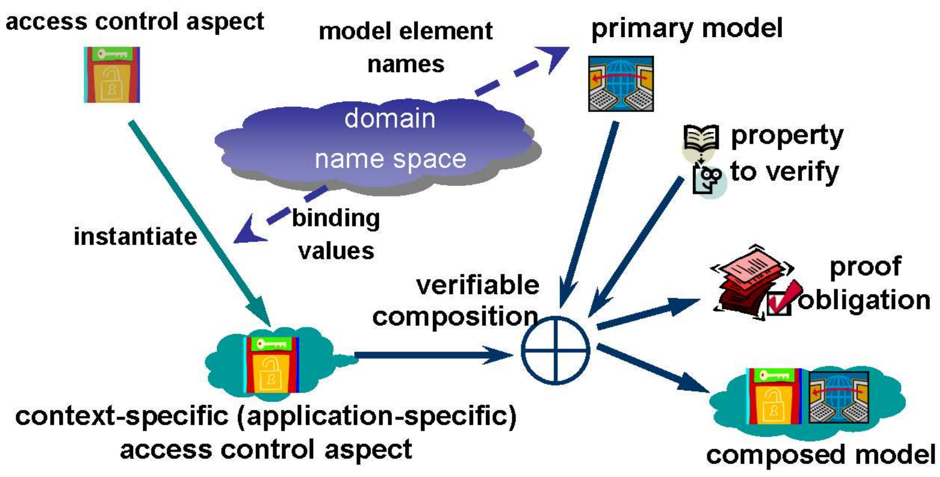


Figure 1.3: Composing an access control aspect model with a primary model

Fig. 1.3 shows how an access control aspect model can be incorporated into an application design in a verifiable manner. When we use the AOM techniques proposed by France *et al.* [31], composition is performed as follows: (1) an *access control aspect* is instantiated in the context of the application to produce a *context-specific* access control aspect that presents an application-specific view of the crosscutting feature (2) this context-specific access control aspect is composed with the primary model.

We extend the previous model composition approach by France *et al.* [31] to support the generation of proof obligations that must be discharged in order to establish that a desired property holds in the composed class model. In our composition approach, class diagrams are merged first, but the entire composition does not end until after the proof obligation is generated and evaluated. When generating the proof obligations, we use the *property to verify* (shown in Fig. 1.3). Verifying that a composed model has the stated property requires one to discharge the generated proof obligation.

If the property does not hold then one can attempt to use composition directives to alter the manner in which the composition is done to help rectify the problem. The properties that are targeted by the technique are those that constrain the effects of sequences of operations on the system state as represented by object structures. To facilitate automation we restrict the forms of operation specifications and target properties.

1.3 Research Scope and Significance

In this research, designs are described using only UML class models and thus the techniques are specific to class model composition. Aspect models in this research consist only of class diagram templates which can be instantiated to produce class diagrams. However, we provide a technique to derive partial sequence diagrams from the class model composition when all message invocations by an operation are stated in the postcondition of its operation specification.

We use a name-based composition approach [31, 86] for composing two aspect models and for composing context-specific aspect models and a primary model. The default procedure in the name-based composition merges model elements that have the same name and syntactic type to produce a single model element in the composed model [86]. This procedure assumes that elements with the same name represent consistent views of the same concept [73]. We use the composition approach to compose an RBAC aspect model with a banking primary model, and to compose RBAC and BLP aspect models to create a hybrid access control aspect model.

The research provides an approach to verifying the enforcement of access control aspect models against the given requirements described in policy models, where policy models are policy specifications represented using the same notation used for specifying access control aspect models. We assume that valid policy models are provided by a developer.

This research is significant in that it provides a methodical approach to verifying composed models. Separating crosscutting access control features as aspect models eases the tasks of analyzing access control features during design. When composing access control aspect models with a primary design model, the verification technique provided in this research allows one to systematically verify that the resulting composed model satisfies specified properties that constrain the effects of operations on system state.

In this research, the verification technique is used to uncover problems and composition directives are used to fix known problems. For example, when an integrated access control policy is required, it may be known that the default merging procedure will not produce the desired result. Composition directives can be used to help ensure that the composition produces the desired result. The result of the composition can be verified using the technique proposed in this dissertation.

1.4 Dissertation Structure Overview

Chapter 2 summarizes the state of the research related to modeling access control policies. Chapter 3 gives the background needed to understand the concepts introduced in this dissertation. Chapter 4 describes how access control features can be modeled as aspect models that enforce the given policy requirements. Chapter 5 shows how two access control policies can be integrated with each other using the AOM composition. Chapter 6 illustrates a verifiable composition approach by showing how an access control aspect model can be incorporated into application features in an automatable and verifiable manner. Chapter 7 concludes the dissertation and outlines directions for future work.

Chapter 2

Related Work

Our work is related to two distinct research areas: aspect-oriented software development (AOSD) and security policy. This chapter gives an overview of relevant work in these two areas and presents related work in the area of specifying access control models using UML.

2.1 Aspect-Oriented Software Development (AOSD)

Aspect-Oriented Software Development (AOSD) supports the *separation of concerns* principle that has proven to be effective at handling complexity [14]. AOSD methods allow developers to represent pervasive design and implementation concerns as aspects. In an AOD approach, a design consists of (1) a primary design or implementation artifact (e.g., a UML model or code) in which the pervasive concerns are not included, (2) a set of aspects, each representing a pervasive design concern that impacts the elements of the primary design artifact, and (3) a composing mechanism that composes aspects with the primary artifact to obtain a view of the design that integrate the primary and aspect model views. Examples of AOSD approaches are (1) aspect-oriented programming (e.g., see [10, 45, 46, 61, 62, 82, 87]) in which the primary design artifacts are code, and aspects are concerns that cross-cut code modules, and (2)aspect-oriented modeling (or design) (e.g. see [20, 21, 28, 38, 88]) in which aspects are design realizations of requirements, and a design is created by composing aspects.

Suzuki *et al.* [88] extend the UML so that it can be used to model code level aspects. Their approach is restricted to secondary system characteristics that can be represented as aspects in an aspect-oriented program. Our approach differs since we do not require aspect-oriented programming techniques. Gray *et al.* [38] use aspects to represent secondary system characteristics in domain-specific models. Their research is part of the Model-Integrated Computing (MIC) initiative that targets embedded software systems specifically. MIC extends the scope and usage of models such that they form the backbone of a development process for building embedded software systems. Requirements, architecture, and the environment of a system are captured in the form of formal high-level models that allow the representation of concerns. Our work can complement their research by providing a UML-based approach for describing aspects.

The Theme/UML approach [1] proposed by Clarke *et al.* [18, 19, 21] and Baniassad *et al.* [5] is a UML-based approach that is closest to the AOM method we use. In their work, a design, called a *theme*, is created for each system requirement. These themes, like context-specific aspect models and primary models, are design views. A comprehensive design is obtained by composing themes. Composition relationships specify how models are to be composed by identifying overlapping concepts in the themes and specifying how models are integrated. Two types of integration strategies are used: override and merge. Override integration is used when existing behavior in a theme needs to be updated to reflect new requirements. Merge integration is used when subjects for different requirements are to be integrated. A theme which has behaviors triggered by other themes is called as an aspect (or a crosscutting theme) in the Theme/UML approach. This type of theme can be parameterized to handle the triggers for its behavior using UML templates.

As part of the early aspects initiative, Rashid *et. al* have targeted multi-dimensional

[1]referred to as subject-oriented design approach with composition patterns in their earlier work

separation beginning early in the software cycle [65, 66, 67, 68]. Their work supports modularization of broadly scoped properties at the requirements level to establish early trade-offs, provide decision support and promote traceability to artifacts at later development stages. Our AOM method compliments this work by supporting aspect specification, composition, and analysis of successively more detailed levels of abstraction needed during system design.

2.2 Security Policies and Access Control Models

In this section we briefly describe some work on security policies and access control models. Damianou's thesis [22] provides a comprehensive survey of important work in this area. A large volume of research exists in the area of access control policies. In this section we describe related work in the area of specifying security policies [6, 7, 12, 16, 23, 39, 41, 42, 43, 49, 57, 59, 74, 92], the area of specifying access control models [1, 4, 8, 9, 11, 17, 35, 55, 75, 78].

Formal logic-based approaches [6, 7, 12, 16, 39, 43, 59] are often used to specify security policies. Jajodia *et al.* [43] propose an authorization specification language (ASL) based on stratified clause form logic. Both negative and positive authorizations can be expressed using this logic. The language also includes *integrity rules* that can be used to specify application-dependent conditions that limit the range of acceptable access control policies. This language provides support for role based access control but no direct support for delegations or obligations. Barker [6] also uses stratified clause form logic to express access control policies with special attention to RBAC. These approaches assume a strong mathematical background which makes one use and understand the specified policies difficult.

In a subsequent work [7], Barker *et al.* show how policies specified in stratified logic can be translated into SQL to protect a relational database from unauthorized read and up-date requests. Ortalo [59] describes a language for specifying security policies based on deontic logic. Researchers [39] at the Cambridge University have defined a language called Role Definition Language (RDL) based on Horn clauses. RDL is based on a set of rules that indicate the conditions under which a client may obtain a name or role. The conditions for obtaining a role depend on the credentials of the client. The notion of delegation in RDL is different in the sense that roles and not access rights are delegated. A client may delegate a role that he himself does not possess. Chen *et al.* [16] propose a language based on set theory for specifying RBAC state related constraints. Bertino *et al.* [12] extends the RBAC model with a temporal model called a Temporal Role-Based Access Control (TR-BAC) model. The language proposed by Bertino can specify periodic activation and deactivation of roles using periodic expressions. They can also specify temporal dependencies among role activation and deactivation using role triggers. Formal logic-based approaches, although, useful for analyzing security policies, are relatively difficult to implement.

Other researchers have used high-level languages to specify policies [41, 42, 57, 74]. Although high-level languages are easier to understand than formal logic-based approaches, they are not analyzable. Ribeiro *et al.* [74] propose a Security Policy Language (SPL) for specifying authorization and obligation policies. Policies are specified using constraint rules. Tower [41] is a language for specifying RBAC policies. The policies are specified using *objects*, *privileges*, *permissions*, *users*, and *roles*. Privileges define a specific access type on an object, permissions are composed of privileges, and roles contain a set of permissions. In addition privileges can also be associated with *conditions* and *actions*. *Conditions* limit the applicability of the privilege. *Actions* are executed when methods associated with the privileges are invoked. The *Organization for the Advancement of Structured Information Standards* (OASIS) technical committee advocates the use of XML for expressing access control policies [57]. They proposed XACML which is an XML specification for expressing policies for information access over the Internet. The policy specification in XACML is very verbose and not aimed for human interpretation.

LaSCO [42] is a graphical approach for specifying policies. The graphical format of LaSCO helps in human interpretation but is not very expressive. Ponder [23] is a specification language that allows various kinds of policies, such as, authorization, obligation, and delegation policies to be specified. Policies are specified in terms of *subject-domain*, *target-domain*, and *access-lists*. The subject-domain specifies the set of subjects that can perform the operations specified in the access-lists on the objects in the target-domain. The authors have also developed a toolkit for policy specification and deployment [24].

The area of conflict analysis of security policy has also received some attention. Lupu and Sloman [52] elaborate on policy conflicts that may occur in large-scale distributed systems and describe a conflict analysis tool that is a part of a Role Based Management Framework. The authors investigate conflicts in authorization and obligation policies. Sibley*et al.* [81, 80] have identified the need for automated tools to specify and analyze policies. They have used both first order logic and an object-oriented approach to represent policy. The policies considered are not limited to access control policies but general rules about the system. Policies are formalized in predicate calculus with the help of enhanced entity relationship diagrams. A theorem prover is used to detect inconsistencies in the specification [53]. Minsky [54] proposes the notion of "law governed systems". These systems implement a common global set of constraints by using filters in every node that ensure that all interactions are consistent with the global law.

Tidswell and Jaeger [92] propose an approach to visualizing access control constraints. They point out the need for visualizing constraints and the limitations of previous work (e.g., [2, 56, 60]) on expressing constraints. Another effort to graphical specification of RBAC is proposed by Koch *et al.* [49]. 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 research utilizes notations from a standardized modeling language and also integrates the policy specification activity with design modeling activities.

Abadi *et al.* [1] presents a calculus for access control which enables one to formally reason about whether access requests should be granted or not. The calculus uses a notion of principals as the sources of requests; a principal can be simple or composite (groups of principals). The model supports delegation of access rights but does not allow one to specify the conditions under which delegation can take place. The proposed calculus is not able to support temporal constraints on authorizations or delegations. Bertino *et al.* [11] propose a formal model for extending authorizations with temporal constraints. They allow the specification of periodic authorizations and authorizations that are valid over specific time periods. The model also allows runtime derivation of new authorizations based on the presence or absence of existing authorizations. Samarati *et al.* [75] suggests adding more general conditions to authorization rules, such as, conditions involving the system state, the state of the object that is being accessed, and on the object's access history. This work also recognizes the need for both positive and negative authorizations. Negative authorizations specify the accesses that should not be granted. The presence of both positive and negative authorizations often lead to inconsistencies which must be detected and resolved.

In our approach, access control policies and access control models are expressed using UML-based notations. Therefore, UML tools can be used to specify them.

2.3 Modeling Access Control Features in UML

Approaches to specifying and analyzing access control features that are based on sophisticated mathematical concepts, formally stated, allow one to check that developed access control features enforce required polices. In practice, however, applying mathematically-based formal specification techniques can be difficult because of the high degree of mathematical skill needed. Therefore, a representation that can be analyzed without sacrificing understandability and usability is desirable.

In this regard, there has been some work on using the UML to model security features (e.g., see [3, 15, 44, 50]. Chan and Kwok [15] 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.* [50] 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. The concepts of RBAC such as User, Role, and Permission are represented as metamodel types. 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.

Steen et al. [85] propose a new language for expressing policies that can be applied over an enterprise that is modeled using UML. The language contains embedded OCL constraints. The constraints cannot specify activation/deactivation of roles or assignment of users or permissions to roles. It also does not allow for the composition of policies.

Jürjens [44] models security mechanisms based on the multi-level classification of data in a system using an extended form of the UML called UMLsec. The UML tag extension mechanism is used to denote sensitive data. Statechart diagrams model the dynamic behavior of objects, and sequence diagrams are used to model protocols. Deployment diagrams are also used to model links between components across servers. 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 research complements the UMLsec by capturing access control policies in patterns that can be reused by developers of secure systems.

France *et al.* [29] and Georg *et al.* [34] have shown how concerns can be modeled as aspects, expressed as structural and behavioral patterns specifications, and composed with designs expressed in the UML (e.g., security concerns [32, 34], and authentication and auditing [33]). The above works have also shown that the order in which the aspects are composed is important.

The aspect composition approach in this research builds upon the techniques described in earlier works (e.g., see [29, 31, 32, 34, 48, 69, 70, 84]). France *et al.* [29] and Georg *et al.* [32, 34] show how security concerns can be localized and then composed with models of system functionality. Kim *et al.* [48] and Ray *et al.* [70] present how invalid structures can be captured and expressed using object diagram templates suggested in the work by France *et al.* [30]. France *et al.* [31] extends the earlier works by George *et al.* [32, 34] by refining the aspect modeling notation and instantiation process, and providing the notion of composition directives to compose a context-specific aspect with a primary model. Ray *et al.* [69] use the AOM based approach for describing access control schemes (e.g., RBAC) and incorporating it into a primary model to produce a composed model and the composed model are analyzed to identify undesirable emergent behaviors. The above earlier works focused on modeling access control features as aspects and incorporating an aspect model into a primary model. Whereas, in the work by Ray *et al.* [71] two access control aspects are integrated, but the composition was not systematic.

This dissertation extends the work by Ray *et al.* [71] by providing a systematic approach to composing aspect models and is also an extension of works by Kim *et al.* [48] and Ray *et al.* [69, 70] in that it illustrates how composition can be carried out in a verifiable manner.

2.4 Model-based Verification

Model-based verification is another area that is close to our approach. As summarized in [93], model-based verification is a process for identifying and correcting errors, which requires integrating established modeling techniques, formal specification methods, and model checking approaches into a systematic software engineering practice. Gluch *et al.* [36, 37] and Engels *et al.* [25] present a model-based verification techniques for software engineering practices. Most of model-based verification work [25, 36, 37, 93] use a model checking tool (e.g., PVS [93] or FDR [51]) for analysis part. Our approach is also in along the same lines in that models are verified at the model level before they are implemented. However, none of the above mentioned approaches addresses identifying problematic compositions during the composition as done in our work.

Chapter 3

Background

This chapter presents background information needed to understand the concepts and notations used in this dissertation.

3.1 Access Control Systems

Access control is the process of ensuring that accesses to protected resources adhere to a set of predefined policies. Access control mechanisms are used to help meet confidentiality[1], integrity[2] and availability[3] goals in software systems [26].

An access control system is typically described in three ways: access control policies, mechanisms, and models [75]. Access control policies are security requirements that describe how access is managed, what information can be accessed and by whom, and under what conditions that information can be accessed [26]. The access control mechanism defines the low-level functions that implement the controls imposed by the policies [75].

[1]Confidentiality refers to the need to keep information secure and private. The condition of confidentiality requires that only authorized users can read information.

[2]The condition of integrity requires that unauthorized persons, processes or devices cannot modify information.

[3]Availability refers to the notion that information is available for use when needed.

These policies are enforced via mechanisms that mediate access requests and grant or deny requested accesses. An access control model specifies an access control system. Access control models must provide ways to reason about the policies they support and prove the security properties of the access control system [40].

3.2 Access Control Policies

There are several well-known access control policies, which can be categorized as discretionary or non-discretionary. Typically, discretionary access control (DAC) policies are associated with identity-based access control and non-discretionary access controls are associated with rule-based controls (for example, mandatory access control (MAC) policy). DAC policies restrict accesses to targets based on the identity of the individual user or group. DAC uses an access matrix model to reason about which subjects can perform which operations on which objects. The rows in an access matrix A correspond to the subjects S, the columns correspond to the objects O, and the entry $A(s,o)$ corresponds to the actions that subject s can perform on the object o. MAC policies enforce access controls on the basis of fixed regulations mandated by a centralized authority, not by the individual owner of an object. For example, MAC occurs in military security, where an individual data owner does not decide who has a *Top Secret* clearance, nor can the owner change the classification of an object from *Top Secret* to *Secret* [63].

Role-based Access Control (RBAC) that was introduced in the late nineties is another well-known type of access control policy. Although RBAC is technically a form of non-discretionary access control [64], RBAC is often considered as one of three primary access control policies together with DAC and MAC. RBAC is an approach to restricting system access to authorized users based on the roles that users play in the organization. In this dissertation, we model access control models that enforce MAC policies and RBAC policies respectively.

Policies can be classified as generic policies and context-specific policies. A generic policy is a statement that can be applied to a set of applications. An example of a generic policy is given below:

Example 1 *"Information items classified as secret can be viewed only by users having a security clearance of secret or higher."*

The above statement is generic in that it implicitly defines the specific information items and users impacted in terms of a property (the "secret" property). Specifically, the policy refers to a category of information items that have the characteristic of being "secret" and it requires a pattern of behavior in which request to view a secret item results in a check of the user's security clearance level.

A context-specific policy is a statement that explicitly identifies the constrained entities and processes. Unlike generic policies, a context-specific policy is specific to an application. A context-specific version of the policy given in Example 1 is given below:

Example 2 *"Project Gemini files classified as secret can be viewed only by Project Gemini Supervisors with a security clearance of secret or higher."*

3.3 Access Control Models

The most cited MAC model is the Bell-LaPadula (BLP) model [9] in which the subjects are cleared at different security levels and the objects are also classified at different security levels. The access privileges that a subject has on an object depends on the security clearance of the subject and the security classification of the object. In the BLP model, the rules under which the subjects can read or write objects are defined in terms of the security levels of subjects and objects. The BLP is aimed at providing confidentiality. A similar model called the Biba model was proposed for integrity [13]. In RBAC models, accesses to protected information is determined by the operations that users playing roles execute in a session. A role is defined as the set of access rights associated with a particular position in the organization. Access rights are not specified with respect to users, but with respect to roles.

In the following subsections, we describe BLP and RBAC models in more details.

3.3.1 Bell-LaPadula (BLP) Model

The BLP model [9] is defined in terms of a security structure $(L, \geq)$, where L is a set of security levels (e.g., Top Secret, Secret) and $\geq$ is a dominance relation between these levels. The main components of this model are objects, users, and subjects. Objects contain or receive information. Each object in the Bell-LaPadula model is associated with a security level which is called the classification of the object. Each user is associated with a security level that is referred to as the clearance of the user. Each user is also associated with one or more subjects. Subjects are processes that are executed on behalf of some user logged in at a specific security level. The security level associated with a subject is the same as the level at which the user has logged in.

The access control policies enforced in the Bell-LaPadula model are specified in terms of subjects and objects. The policies for reading and writing objects are given by two properties stated below:

- *Simple Security Property*: A subject S may have read access to an object O only if the security level of the subject $L(S)$ dominates the security level of the object $L(O)$, that is, $L(S) \geq L(O)$.
- $\star$ *Property*: A subject S may have write access to an object O only if the security level of the subject $L(S)$ is dominated by the security level of the object $L(O)$, that is, $L(O) \geq L(S)$.

The restricted star property given below provides integrity as well as confidentiality:

- *restricted-$\star$ Property*: A subject S may have write access to an object O only if the security level of the object $L(O)$ is the same as the security level of the subject $L(S)$, that is, $L(O) = L(S)$.

3.3.2 Role-Based Access Control (RBAC)

RBAC [27] is used to protect information targets (henceforth referred to simply as targets) from unauthorized users. To achieve this goal, RBAC specifies and enforces different kinds of constraints. Core RBAC defines the properties that must be present in any RBAC application. Core RBAC requires that users be assigned to roles, roles be associated with permissions (approval to perform an operation on a target), and that users acquire permissions through their associated roles. For example, in a banking application, users can be assigned to roles such as loan officer and teller, where a loan officer has permission to issue loans to customers.

Sandhu *et al.* [77] have specified four conceptual RBAC models. Core RBAC ($RBAC_0$) is the most basic model. In core RBAC, a user can establish a session to activate a subset of roles to which the user is assigned. Hierarchical RBAC ($RBAC_1$) includes $RBAC_0$ and introduces *role hierarchies*. Hierarchies structure roles to reflect an organization's lines of authority and responsibility and they are specified using inheritance of roles. $RBAC_2$ includes $RBAC_0$ and introduces constraints to restrict the assignment of users or permissions to roles, or the activation of roles in sessions. Constraints are used to specify application dependent conditions, such as, separation of duties. $RBAC_3$ combines both $RBAC_1$ and $RBAC_2$, thus providing role hierarchies as well as constraints.

Core RBAC does not place any constraint on the cardinalities of the user-role assignment relation or the permission-role association. In core RBAC 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.

Hierarchical RBAC adds role hierarchies to Core RBAC. Role hierarchies define inheritance relation among the roles in terms of permissions and user assignments. If role *r1* inherits role *r2* then all permissions of *r2* are also permissions of *r1* and all users of *r1* are also users of *r2*. There are no cardinality constraints on the inheritance relationship. The inheritance relationship is reflexive, transitive and anti-symmetric.

Static Separation of Duty (SSD) relations are used to define conflicting roles: If a user is assigned to roles that conflict then there is a conflict of interest with respect to permissions assigned to the user via the roles. SSD relations between roles constrain how users are assigned to roles: Membership in one role that takes part in an SSD relation prevents the user from being a member of the other role. The SSD relationship is symmetric, but it is neither reflexive nor transitive. SSD relations may exist in the absence of role hierarchies (referred to as SSD RBAC or RBAC$_2$), or in the presence of role hierarchies (referred to as hierarchical SSD RBAC or RBAC$_3$). The presence of role hierarchies complicates the enforcement of the SSD relations: Before assigning users to roles not only should one check the direct user assignments but also the indirect user assignments that occur due to the presence of the role hierarchies.

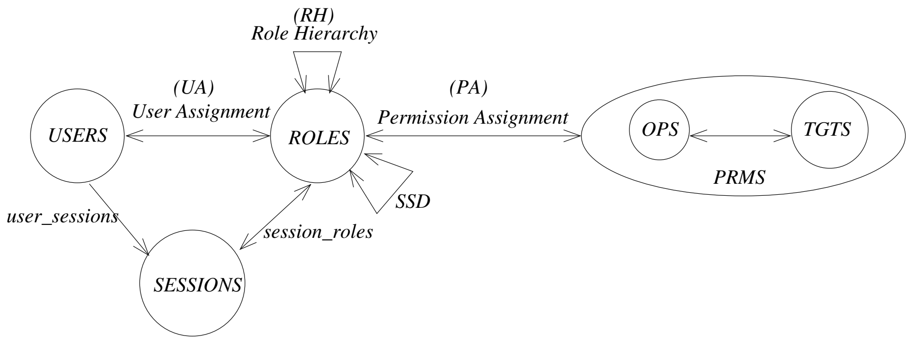


Figure 3.1: Hierarchical SSD Role-Based Access Control (taken from [27])

Fig. 3.1 shows a model of hierarchical SSD RBAC that consists of: (1) a set of users (*USERS*) where a user is an intelligent autonomous agent , (2) a set of roles (*ROLES*) where a role is a job function , (3) a set of sessions (*SESSIONS*) where a user establishes a session during which he/she activates a subset of the roles assigned to him/her, (4) a set of targets (*TGTS*), where a target is an entity that contains or receives information, (5) a set of operations types (*OPS*) where an operation describes a service provided by the application, and (6) a set of permissions (*PRMS*) where a permission is an approval to perform an operation on targets. The cardinalities of the relationships are indicated by the absence (denoting one) or presence of arrow heads (denoting many) on the corresponding associations. For example, the association of user to session is one-to-many. All other associations shown in the figure are many-to-many. The association labeled *Role Hierarchy* defines the inheritance relationship among roles. The association labeled *SSD* specifies conflicting roles.

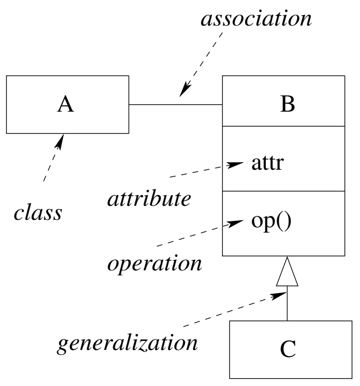
3.4 Unified Modeling Language

The UML [90] is a widely-used standard modeling language for object-oriented systems maintained by the Object Management Group (OMG) (see www.omg.org), a standards body for the object-oriented community. The UML prescribes a standard set of diagrams and notations for modeling object-oriented systems, and describes the underlying semantics of what these diagrams and symbols mean. It began as a consolidation of the work of Grady Booch, James Rumbaugh, and Ivar Jacobson, creators of the most popular object-oriented methodologies. UML 1.0 was proposed by the UML Partners, a consortium of several organizations, in 1997 in response to an OMG's request for proposals for a standard object-oriented analysis notation and semantic metamodel. Several revisions have been produced since the UML 1.0, and the most recent work, version 2.0, was approved by the OMG in 2005 (refer to [90]).

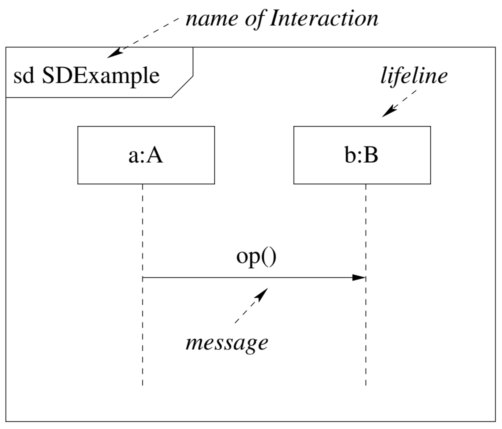
UML 2.0 offers thirteen types of diagrams to model systems, including use case, activity, class, sequence, and statechart diagrams [90]. Each diagram describes a different view of the system being modeled. Constraints on structure and behavior are stated using the

Object Constraint Language (OCL) [91]. In this dissertation we use the following three UML diagram types:

- Class Diagram: A diagram that describes classifiers and their relationships. Properties are specified in the form of invariants and operation pre- and postconditions using the OCL [91].
- Sequence Diagram: A diagram that describes how instances interact to accomplish a task.
- Activity Diagram: A diagram that describes the flow of control (and optionally data) through steps of a computation.



(a) A class diagram example



(b) A sequence diagram example

Figure 3.2: Class diagram and sequence diagram examples

Fig. 3.2 shows examples of a class diagram and a sequence diagram. A class diagram describes classifiers (e.g., classes, interfaces, types) and their relationships. A class is a classifier that characterizes a family of objects in terms of attributes and operations that are common to the objects. An operation can be specified using pre- and postconditions expressed in the OCL. Links between class objects are specified by associations between classes. The ends of associations, referred to as *association-ends* in UML metamodel, have properties such as multiplicity and navigability. The class diagram in Fig. 3.2(a) shows three classes *A*, *B*, and *C*, and their relationships. A class *B* has a generalization/specialization relationship with a class *C*, which specifies that *C* inherits the features of *B*. The association between *A* and *B* declares that there can be links between the instances of *A* and *B*.

A sequence diagram describes how objects interact to accomplish a task [47]. An interaction is expressed in terms of *lifelines* and *messages*. A lifeline is a participant in an interaction. In this dissertation, messages represent operation calls. For example, the sequence diagram in Fig. 3.2(b) shows that *a:A*, a lifeline of a class *A* object, sends a message to *b:B*, a lifeline of a class *B* object, to carry out a specific goal. The sequence diagram notation can be used to specify alternative sets of interactions and iterations over interactions.

OCL expressions are used to formalize invariants for classes, preconditions and postconditions for operations. For example, we can add the following OCL expression to the class *Company* shown in Fig. 3.3:

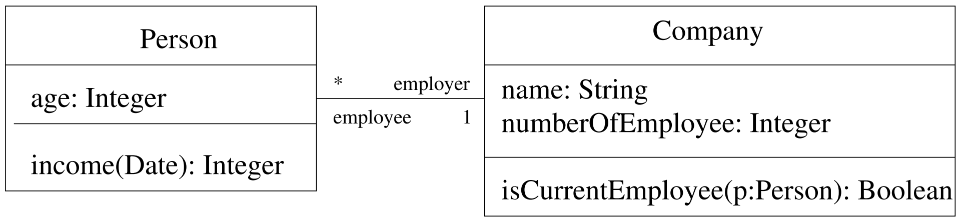


Figure 3.3: Another class diagram example

context Company **inv:**
self.numberOfEmployees $>$ 50

The above constraint defines an invariant of the class *Company* stating that the value of an attribute *numberOfEmployees* must be greater than 50 in all consistent states of the system[4]. Each OCL expression is evaluated in the context of an instance of a specific type and the reserved keyword *self* is used to refer to the instance. For example, *self* represents an instance of the type *Company* in the above invariant. The attributes, association ends and operations of an instance can be accessed using "." (*dot*). In the above example, *self.numberOfEmployees* denotes the attribute *numberOfEmployees* of the company.

Preconditions and postconditions are constraints that specify applicability and effect of an operation without stating an algorithm or implementation [94]. The following constraint can be added to the class *Company* shown in Fig. 3.3 as well:

context Company::isCurrentEmployee(p:Person):Boolean

pre: true

post: result = self.employee→includes(p)

In the above example, there is no precondition for this operation, so the constrain always holds. For the postcondition, the object that is returned by the operation can be referred to by the reserved keyword *result*. When the multiplicity of an association end is grater than 1, a navigation results in a collection of objects. In our example expression, the navigation from a company to associated employees results in a set of employee objects. OCL has some built-in primitive types (e.g., *Boolean*, *String*, *Integer*) and collection types (e.g., *Set*, *Bag*). Collections have many predefined operations on them (e.g., *includes*, *excludes*, *includesAll*, *excludesAll*, *isEmpty*). To access this type of operations, an arrow symbol is used in OCL instead of dot symbol. For example, the OCL type *Set* has the operation *includes* of type *Boolean* that tests whether the object passed as a parameter is an element of the collection. In our example expression, the operation *isCurrentEmployee*

[4]While the system is, for instance, executing an operation, it is not in a consistent state, and the invariant need not be true. Of course, when the execution has finished, the invariant must again be true [91].

checks whether the set of persons referenced by the association end *employee* contains a *Person* object *p* given as a parameter or not. Refer to [94] for more detail.

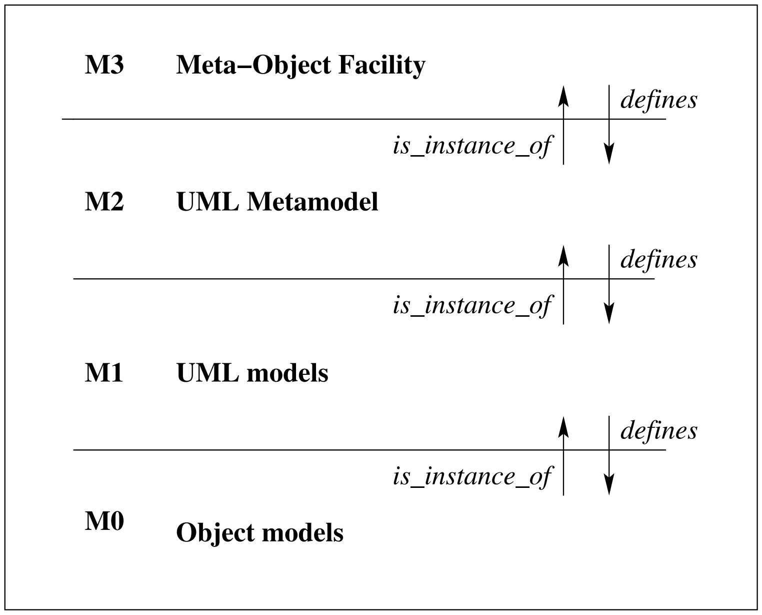


Figure 3.4: UML four layer metamodel architecture

The UML infrastructure is defined as a four-layer architecture (see Figure 3.4).

- Level M3 (meta-metamodel layer) defines a language for specifying metamodels. The Meta Object Facility (MOF) [58] is an example of meta-metamodel.
- Level M2 (metamodel layer) contains models that specify modeling languages. The UML metamodel and the Common Warehouse Metamodel (CWM) [89] are examples of metamodels.
- Level M1 (model layer) contains models that describe semantic domains. The model layer consists of models expressed in languages specified by the metamodel at level M2 .
- Level M0 (instance layer) consists of actual instances (objects) of the running system specified by the models at level M1.

3.5 Aspect-Oriented Modeling (AOM)

In this section we give an overview of the AOM approach [31, 69] on which our work is based. Aspect models in the AOM approach describe crosscutting features. A crosscutting feature can be isolated if its distributed elements have common structural and behavioral characteristics. A generalized form of the solution can then be represented as a pattern, where the pattern describes common characteristics of the distributed solution parts. A pattern view of crosscutting solutions screens out context-specific details and makes it possible to conceive, describe, and understand the solutions in isolation. In our AOM approach an aspect model is a pattern that characterizes a family of features. The patterns are described using UML model templates. UML model template notation is an adaptation of a UML-based pattern language, called the Role-Based Metamodeling Language (RBML) [30].

3.5.1 An Overview of the AOM Approach

The AOM approach utilizes the following items [31]:

1. A *primary (base) model*: The primary model describes the core functionality that determines the dominant design structure. It is described using the Unified Modeling Language (UML) [90].

2. A set of *aspect models*: Each aspect model describes a feature that crosscuts the dominant structure described in the primary model. The crosscutting features are described as patterns in aspect models.

3. A set of *bindings*: A binding associates an application-specific value to a template parameter. Applying the bindings to an aspect model produces a *context-specific aspect model* that describes how the feature is to be realized in the primary model.

4. A *basic model merging procedure*: A name-based procedure is used to merge a

context-specific aspect and a primary model. Elements with the same name are merged in the composed model.

5. A set of *composition directives*: Composition directives are used to help ensure that composition produces desired models.

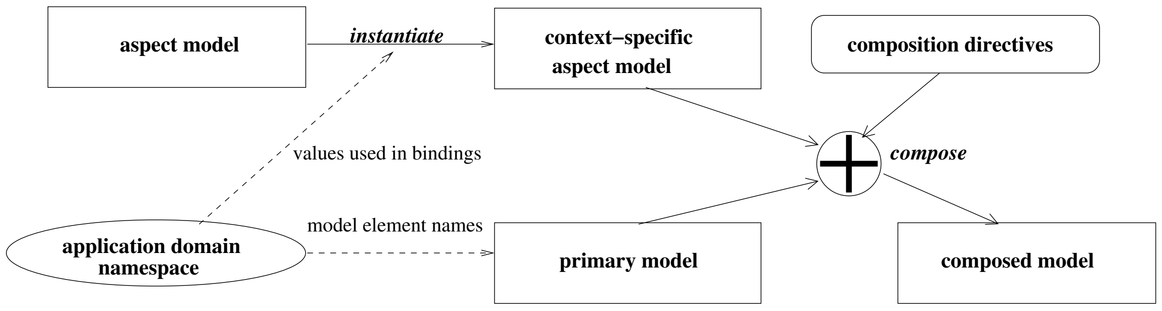


Figure 3.5: An overview of composition in the AOM approach [84, 31]

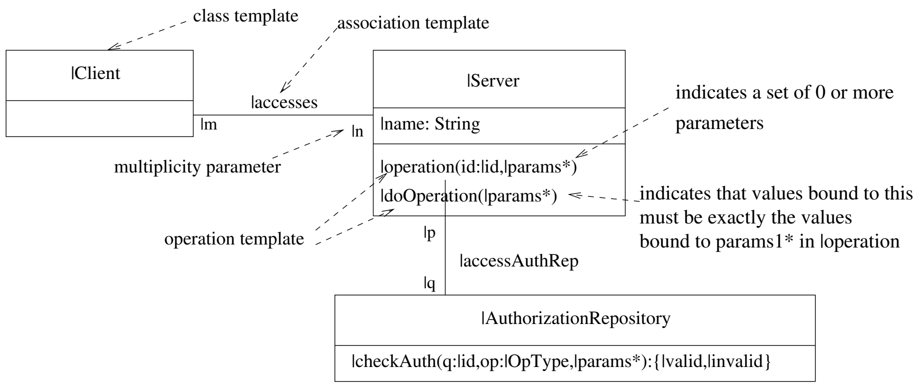
Fig. 3.5 gives an overview of composition in the AOM approach [31, 84]. The first step is to identify the bindings needed to generate a context-specific aspect. The context-specific aspect model is then composed with the primary model to produce a composed model. Composition directives are often needed to ensure that composition produces required results [73, 86]. They can be used to (1) determine the order in which multiple aspects are composed with the primary model, (2) modify models before they are merged, (3) override specific parts of the basic merge procedure, and (4) modify the model produced by the basic merge procedure.

3.5.2 Representing Aspect Models

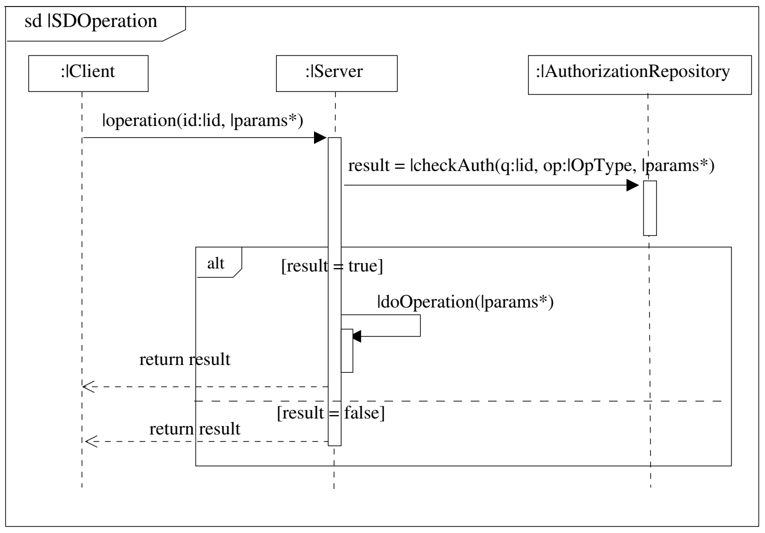
Aspect models can be represented as UML diagram templates representing patterns of features. The diagram templates used in this dissertation produce UML design diagrams when instantiated. In the UML, template models are described by parameterized packages that explicitly list the parameters in the package header. However, this notation is unwieldy when a large number of parameters are involved. France *et al.* [30] and Kim [47] developed a specialized form of the Role-Based MetaModeling Language (RBML) to describe aspect models. RBML is a UML based language that supports rigorous specification of pattern solutions. The specialized RBML is used to create RBML models consisting of a set of diagram templates and OCL templates. Aspect models consist of class diagram templates and sequence diagram templates. Since RBML uses UML syntax, UML tools can be used to create RBML models. The aspect model consists of two diagram templates: A *class diagram template* that describes structural properties of the features and a *sequence diagram template* that describes interactions among feature elements. Class diagram templates and sequence diagram templates have template model elements that are explicitly marked using the "|" symbol. 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. Operation templates may be associated with template forms of pre- and postconditions, referred to as *operation specification templates*, that produce OCL specifications when instantiated. These operation specification templates are presented separately from the diagrams to reduce diagram clutter. For example, Fig. 3.6 shows an aspect model, $Authorization$, characterizing features in which access to a service is restricted to authorized clients.

Instantiating the class diagram template shown in Fig. 3.6(a) results in a class diagram that consists of composite classes representing logical architectural views of clients, servers with services under access control, and authorization repositories. The class template *Server* contains an attribute template with a name parameter (i.e., *name*) and two operation templates (i.e., *operationi* and *doOperation*). A service under access control is represented by these two operations in a server class:

- An operation that checks whether a client that requests the service is authorized to execute the service. The operation signature is obtained by instantiating the operation template $|operation$. The operation takes in as arguments the client's identifier


class template
association template
|Client
|Server
|accesses
|m
|n
|name: String
indicates a set of 0 or more parameters
multiplicity parameter
|operation(id:|id,|params*)
|doOperation(|params*)
indicates that values bound to this must be exactly the values bound to params1* in |operation
operation template
|p
|accessAuthRep
|q
|AuthorizationRepository
|checkAuth(q:|id,op:|OpType,|params*):{|valid,|invalid}

(a) Class Diagram Template for an Authorization Aspect Model



(b) Sequence Diagram Template for an Authorization Aspect Model

Figure 3.6: An Authorization-based Access Control Aspect Model (taken from [31] and modified)

(represented by the operation argument template $id : |id$) and zero or more values needed by the service (represented by the argument template $|params*$). The template parameter $params*$ is referred to as a *collection parameter* indicating that it must be bound to a collection of values.

- An operation that performs the required service. This operation is obtained by instantiating the operation template $|doOperation$. The use of the $|params*$ collection parameter in both the $operation$ and $doOperation$ templates indicates that the same value (i.e., the same set of arguments) must be used to instantiate the collection parameter in both of the templates.

The class template $|AuthorizationRepository$ contains the operation template $|checkAuth$ that produces an operation that performs authorization checks when instantiated. A $|checkAuth$ operation uses the client identifier (represented by $q : |id$), an operation identifier (represented by $op : |OpType$), and possibly other information passed in as arguments (represented by the collection parameter $|params*$), to determine whether the client is authorized to access the operation or not. If the client is authorized the operation returns a value that is an instantiation of $|valid$, otherwise it returns a value that is an instantiation of $|invalid$. The following is the annotated operation specification template associated with the $|operation$ template:

context |Server::|operation(id:|id,|params*)
  pre: true
  −− This operation can be invoked at any time.
  post:
  −− The service is carried out if and only if the client is
  −− authorized to invoke the service.
  let authmessage : OclMessage =

```
    |AuthorizationRepository^|checkAuth(q:|id,op:|opType,|params*)
in
  (authmessage.hasReturned() and authmessage.result() = true
    implies |Server^|doOperation(params*)) and
    (|Server^|doOperation(params*) implies
    authmessage.hasReturned() and authmessage.result() = true)
```

An association template consists of multiplicity parameters (one at each end) that yield association multiplicities (integer ranges) when instantiated. The multiplicity in an alphabet letter (e.g., “a” on the *Session* end of the *UserSessions* template) is an unconstrained multiplicity parameter, that is, any integer range of multiplicity (e.g., “1..*”, “*”, “1..4”) can be instantiated from it.

The sequence diagram template shown in Fig. 3.6(b) consists of template forms of participants (i.e., $: |Client$, $: |Server$, and $: |AuthorizationRepository$) and messages (e.g., $|operation(|id, |params*)$). Instantiating a participant template produces either a named or anonymous participant, for example, binding $UserMgmt$ to the parameter *Server* in the $: |Server$ participant template produces the anonymous participant $: UserMgmt$. In a participant template, the type parameter (e.g., $|Server$ in $: |Server$) must be a classifier template in a corresponding classifier diagram template. Participant type parameters and the corresponding classifier templates must be instantiated with the same value.

Message templates consist of parameterized message expressions. For example, a message template *result := |checkAuth(q:|id,op:|opType,|params*)* is a parameterized message expression that includes three mandatory parameters *checkAuth*, *id*, and *opType*, and an optional set of arguments indicated by the collection parameter $params*$. The message expression $result := IDcheck(q : Userid, op : UpdateOp, userstatus : Status, usersession : Session)$ can be obtained from this template using the following bindings: $IDcheck \mapsto checkAuth$, $Userid \mapsto id$, $UpdateOp \mapsto opType$, and $<(userstatus \mapsto params)$, (*usersession* $\mapsto$ *params*)$>$.

For readability, we do not show the template parameter indicator symbol "|" in the following chapters of this dissertation. In such cases, all the user-defined names in the diagram are template parameters that must be bound to values when instantiating the aspect model. For example, the following values are examples of ones that are not template parameters: UML keywords such as *Boolean* and *enumeration*, UML stereo-typed name such as *prohibited*, OCL keywords such as *self* and *includesAll*, and all variable names.

In this dissertation aspect models specified using UML diagram template notation are defined at the M2 (matemodel layer) level since aspect models are generic descriptions of model families. The context-specific aspect models and primary model are defined at the M1 (model layer) level.

Chapter 4

Enforcing an Access Control Policy in an Aspect Model

From a software design perspective, access control policies are requirements that must be addressed in a design. For example, access control policies are constraints that determine the type of access authorized users have on information resources. In this chapter, we show how one can formulate access control policies as a policy model, formulate an access control aspect model that enforces policies as an aspect, and verify whether the aspect model enforces the given policies or not. We show two access control policy examples, RBAC and BLP.

4.1 Formulating Access Control Policies

Policies are expressed in terms of UML class diagrams with OCL constraints. We define this form of diagrams with constraints as a policy model. A policy model is obtained by analyzing the given policy statement as illustrated in Fig. 4.1. For example, the following shows how the Role-Based Access Control (RBAC) policy can be expressed in a policy model. We use the Proposed NIST standard for role-based access control [27] as RBAC requirements. The access control policy statement must describe under what condition a user does or does not have permission to access a target to perform a certain type of operations in a session. The RBAC policy is stated as follows:

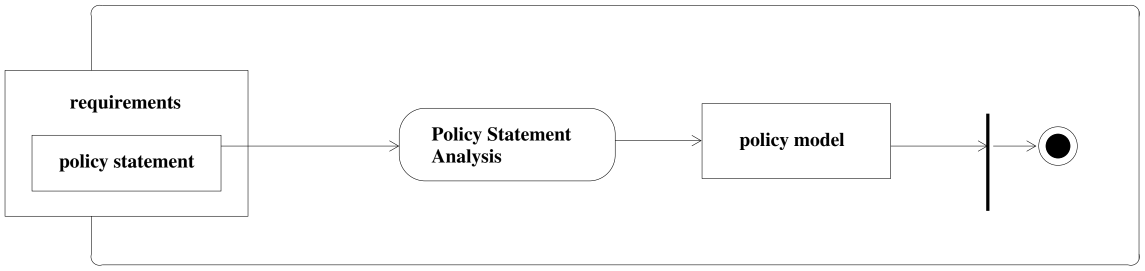


Figure 4.1: Obtaining a policy model from requirements

P$_{RBAC-1}$: If a user *u* has permission to access a target *t* to perform operations of type *op* in a session *s*, then there exists a role *r* with the following properties:

- *r* has permission to access *t* to perform operations of type *op*,
- *r* is an authorized role for *u*, and
- *r* is currently activated in *s*.

P$_{RBAC-2}$: Roles activated in a session must be a subset of the roles assigned to the user of the session.

A class model that is required to describe the RBAC policy is shown in Fig. 4.2. The

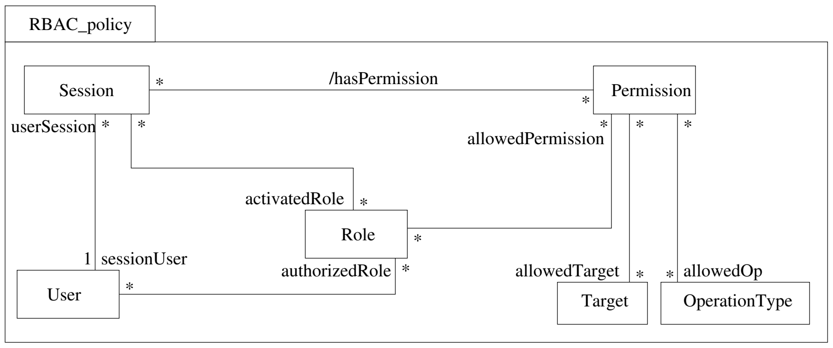


Figure 4.2: An RBAC policy model

User class and *Session* class in the policy model represent a set of users and a set of sessions respectively. The *Role* class in the policy model represents a set of roles that a user can play.

The *Permission* class in the policy model represents pairs of sets where one part of a pair is a set of *Target* instances and the other is a set of *OperationType* instances.

To formally specify the first part of the policy statement, P_{RBAC-1}, we define the derived *hasPermission* relationship between *Session* and *Permission* that links sessions with their permissions as stated in the RBAC policy P_{RBAC-1}. The OCL statements that define the RBAC policy P_{RBAC-1} are given below:

P$_{RBAC-1}$:

```
context Session inv:
hasPermission→forAll(p:Permission|
    activatedRole→exists(r|r.allowedPermission→includes(p)) and
    sessionUser.authorizedRole→exists(r|r.allowedPermission→includes(p)))
```

where a derived association *hasPermission* is defined as follows:

```
context Session:: hasPermission : Set(Permission)
derive: activatedRole.allowedPermission
```

The policy statement P_{RBAC-2} is expressed in the OCL as follows:

P$_{RBAC-2}$:

```
context Session inv:
sessionUser.authorizedRole→includesAll(self.activatedRole)
```

These two policy properties must be true in the model that enforces the RBAC policy. The policy model described above is used for showing the policy enforcement that will be described later in Section 4.3.

As another example, we show the case of BLP model. The BLP policy can be expressed as follows:

P$_{BLP-1}$**:** If a user u has permission to read from or write to a target t in a subject s by performing an operation of type op, then all of the following are satisfied:

- op is a read type and the security level of s dominates the security level of t, or op is a write type and the security level of s is equal to the security level of t,
- s is a subject for u and the security level of u dominates the security level of s

P$_{BLP-2}$: the security level of a BLP subject must be dominated by the security level of its user

Fig. 4.3 shows a class model of the BLP policy as stated above.

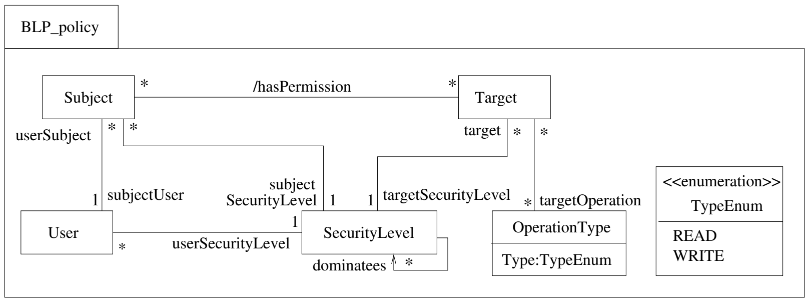


Figure 4.3: A BLP policy model

The following OCL statements expresses the BLP policies P$_{BLP-1}$ and P$_{BLP-2}$:

P$_{BLP-1}$:
```
  context Subject inv:
  hasPermission→forAll(t:Target|
    t.targetOperation→forAll(op:OperationType|
      ((op.Type = TypeEnum::READ and
        subjectSecurityLevel.dominatees→includes(t.targetSecurityLevel))
        or
        (op.Type = TypeEnum::WRITE and
        subjectSecurityLevel = t.targetSecurityLevel))
      and
      subjectUser.userSecurityLevel.dominatees→includes(subjectSecurityLevel)))
```

P$_{BLP-2}$:
```
  context Subject inv:
   self.subjectUser.userSecurityLevel.dominatees→includes(self.subjectSecurityLevel)
```

These two BLP policy properties must be satisfied in the aspect model that enforces the BLP policy.

4.2 Formulating Access Control Models As Aspect Models

Fig. 4.4 shows the class diagram template of the hierarchical SSD RBAC aspect model that was illustrated in Fig. 3.1. The class diagram template of the hierarchical SSD RBAC

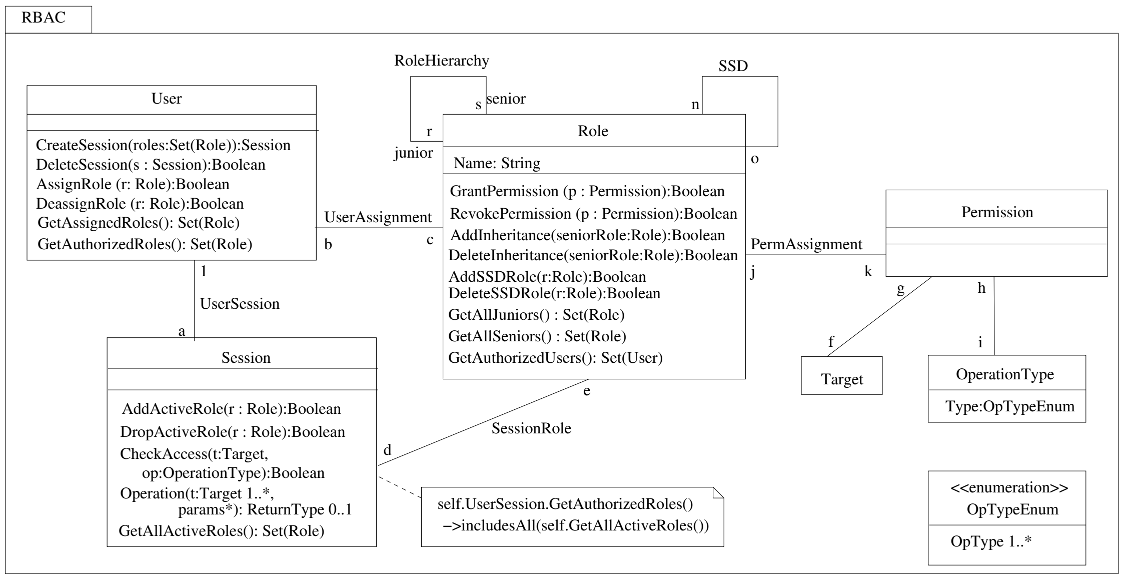

RBAC
RoleHierarchy
SSD
User
CreateSession(roles:Set(Role)):Session
DeleteSession(s : Session):Boolean
AssignRole (r: Role):Boolean
DeassignRole (r: Role):Boolean
GetAssignedRoles(): Set(Role)
GetAuthorizedRoles(): Set(Role)
Role
Name: String
GrantPermission (p : Permission):Boolean
RevokePermission (p : Permission):Boolean
AddInheritance(seniorRole:Role):Boolean
DeleteInheritance(seniorRole:Role):Boolean
AddSSDRole(r:Role):Boolean
DeleteSSDRole(r:Role):Boolean
GetAllJuniors() : Set(Role)
GetAllSeniors() : Set(Role)
GetAuthorizedUsers(): Set(User)
senior
junior
UserAssignment
PermAssignment
Permission
UserSession
Session
AddActiveRole(r : Role):Boolean
DropActiveRole(r : Role):Boolean
CheckAccess(t:Target, op:OperationType):Boolean
Operation(t:Target 1..*, params*): ReturnType 0..1
GetAllActiveRoles(): Set(Role)
SessionRole
self.UserSession.GetAuthorizedRoles() ->includesAll(self.GetAllActiveRoles())
Target
OperationType
Type:OpTypeEnum
<<enumeration>> OpTypeEnum
OpType 1..*
a b c d e f g h i j k l n o r s

Figure 4.4: The class model template view of the RBAC aspect model

aspect model consists of a set of users, a set of roles, a set of user sessions, a set of targets, a set of operation types, and a set of permissions as described earlier in Section 3.3. Users are assigned to roles, roles are associated with permissions, and users acquire permissions by being members of roles. Association templates, such as *UserAssignment* and *UserSessions* produce associations between instantiations of the class templates they connect. The multiplicity “1” on the *User* end of the *UserSessions* template is strict: a session can only be associated with one user. Only roles that are assigned to the user of a session can be activated for that user in the same session. The following invariant shown in Fig. 4.4 specifies the above constraint:

context Session **inv**:

self.UserSession.GetAuthorizedRoles()$\rightarrow$includesAll(self.GetAllActiveRoles())

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. The operation template $Operation$ in the *Session* template represents operations under access control. Instantiating $Operation$ produces an operation that describes a behavior that is performed on target elements (e.g., a withdraw operation on an account element in a banking system). The 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*. Table 4.1 gives an overview of operation templates in each class template shown in Fig. 4.4.

All the user-defined names[1] in the diagram are template parameters that must be bound to values when instantiating the aspect model. Note that we used the symbol "|" to indicate template parameters in Section 3.5. From this chapter, however, we do not show the template parameter indicator symbol "|" for the readability. For example, the parameter template *params* in the $Operation$ template shown in *Session* of Fig. 4.4 must be bound to a list of zero or more operation arguments as indicated by the "$*$" following the parameter name.

An instantiation of $Operation$ can have one or more arguments representing target elements (*t:Target 1..**) and zero or more other arguments (*params **). For example, the operation *transfer (from:Account, to:Account, amount:Integer)* can be obtained from $Operation$ using the following bindings for template parameters: $<transfer \mapsto Operation, Account \mapsto Target, amount : Integer \mapsto params>$, where $(value \mapsto parameter)$ represents a binding

[1]In Fig. 4.4, for example, the following values are examples of ones that are not template parameters: UML keywords such as Boolean, String, Set, and enumeration, OCL keywords such as self and includesAll , and all variable names such as s for a Session object, t for a Target object.

that is done by providing a value for a parameter. More detailed specification of *Operation* and its instantiation examples will be described later in Chapter 6. Table 4.1 gives an overview of operation templates in each class template.

The *CheckAccess* operation template in *Session* is intended to enforce the RBAC policy P_{RBAC}. The OCL specification associated with *CheckAccess* is given below:

```
context Session::CheckAccess(t:Target, op:OperationType):Boolean
  pre: true
  post: result =
        self.GetAllActiveRoles().Permission → exists (p |
           p.Target → includes(t) and p.OperationType → includes(op))
```

The postcondition of *CheckAccess* states the following: *If there exists an activated role with the required permission, the CheckAccess operation returns true, otherwise it returns false.*

The behavior described by *CheckAccess* is called whenever an operation under access control (represented by *Operation*) is invoked. When the behavior described by *Operation* is invoked it first checks whether the caller has permission to perform the operation on the target objects (the set of targets represented by the parameter t in the *Operation* template) by invoking the behavior described by *CheckAccess* for each target object. Invariants and operation specifications associated with other elements in the RBAC aspect model are given in the Appendix A.

Fig. 4.5 shows the class diagram template of the BLP aspect model. Access control policies in BLP are specified in terms of dominance relation between security levels of a subject and an object. The dominance relation between security levels in BLP (e.g., $L(S)$ dominates $L(O)$) is expressed in terms of the query operations *GetAllDominatees* and *GetAllDominators*. *AddDominatee* in *SecurityLevel* adds a link between the current

Table 4.1: The list of operation templates defined in RBAC.

Operation Template	Description
User Class Template	
CreateSession	creates a new session and activates a default role set; creates a *UserSession* link between the user and the session
DeleteSession	deactivates roles that are activated in the given session and deletes that session; deletes a *UserSession* link
AssignRole	creates a *UserAssignment* link between the user and the given role
DeassignRole	deletes a *UserAssignment* link
GetAssignedRoles	returns the set of roles directly assigned to the user as well as those roles that are inherited by the directly assigned roles
GetAuthorizedRoles	returns the set of roles directly assigned to the user as well as those roles that are inherited by the directly assigned roles (junior roles)
Session Class Template	
AddActiveRole	creates a new *SessionRole* link between the session and the role that is one of roles in the authorized role set.
DropActivatedRole	deletes a *SessionRole* link between the session and the given role.
GetAllActiveRoles	returns a set of all roles which are activated for that session and all their junior roles
CheckAccess	determines whether an access should be granted or not
Operation	invokes the operation under access control if the caller has permission to each target
Role Class Template	
GrantPemission	creates a new *PermAssignment* link between the role and the given
RevokePemission	permission deletes a new *PermAssignment* link
AddInheritance	adds a link *RoleHierarchy* to a senior role
DeleteInheritance	deletes a link *RoleHierarchy* to a senior role
AddSSDRole	creates a new *SSD* link between roles
DeleteSSDRole	deletes an *SSD* link
GetAllJuniors	returns a set of roles which consists of direct junior roles and all other junior roles acquired by the transitive closure relation in a role hierarchy
GetAllSeniors	returns a set of roles which consists of direct senior roles and all other senior roles acquired by the transitive closure relation in a role hierarchy
GetAuthorizedUsers	returns the set of users that are assigned to the role and its senior
CheckAccess	roles determines whether an access should be granted or not
Permission Class Template	
CheckAccess	determines whether an access should be granted or not

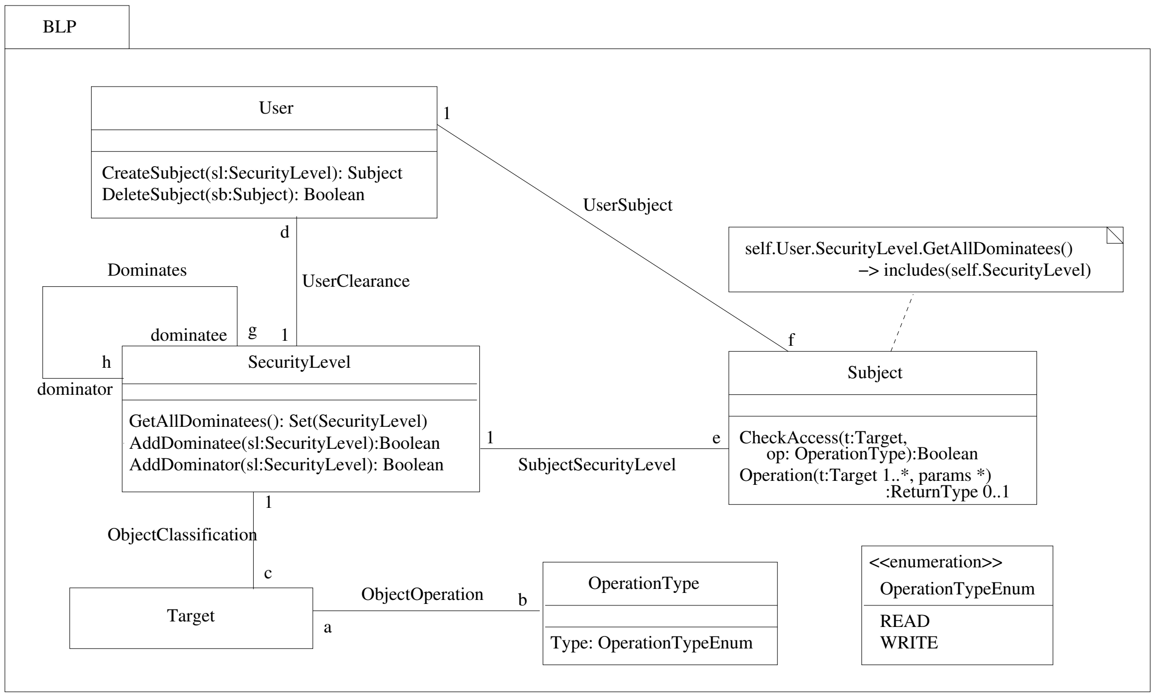

BLP
User
CreateSubject(sl:SecurityLevel): Subject
DeleteSubject(sb:Subject): Boolean
1
UserSubject
d
UserClearance
self.User.SecurityLevel.GetAllDominatees()
-> includes(self.SecurityLevel)
Dominates
dominatee
g
1
f
h
SecurityLevel
Subject
dominator
GetAllDominatees(): Set(SecurityLevel)
AddDominatee(sl:SecurityLevel):Boolean
AddDominator(sl:SecurityLevel): Boolean
1
e
SubjectSecurityLevel
CheckAccess(t:Target,
op: OperationType):Boolean
Operation(t:Target 1..*, params *)
:ReturnType 0..1
1
ObjectClassification
c
<<enumeration>>
OperationTypeEnum
READ
WRITE
Target
ObjectOperation
b
a
OperationType
Type: OperationTypeEnum

Figure 4.5: The class model template view of the BLP aspect model

security level (dominator) and the given security level (indicated by the operation parameter *sl*), and *CreateSubject* in *User* creates a subject with a given security level *sl*. According to the invariant associated with *Subject* (see Fig. 4.5), a new subject can be created when the security of the user dominates the security level of the subject to be created. This constraint is checked in the precondition of *CreateSubject*.

The access control privileges are enforced by making all operations under access control invoke the *CheckAccess* operation in a session. If the operation is a write operation and the security level of the target is equal to the security level of the subject, the *CheckAccess* operation returns true; if the operation is a read operation and the security level of the target is equal to or dominated by the security level of the subject, the *CheckAccess* operation also returns true; otherwise the *CheckAccess* operation returns false. The specification template associated with *CheckAccess* is given below:

context Subject::CheckAccess(t:Target, op:OperationType):Boolean

pre: true

post: result =

    t.OperationType→includes (op)
    and
    ((op.Type = OperationType::READ and
        self.SecurityLevel.GetAllDominatees()→includes (t.SecurityLevel))
        or
    (op.Type = OperationType::WRITE and
        self.SecurityLevel = t.SecurityLevel))

Specification templates for the other operations and specification templates describing other invariant properties of BLP elements are given in the appendix B Appendix B.

4.3 Verifying Policy Enforcement

In this section, we illustrate how one can apply our approach to verifying the policy enforcement of an aspect model using RBAC and BLP aspect model examples. Fig. 4.6 shows the policy enforcement verification process we developed.

policy model and aspect model : These models are two inputs to the verification process. Note that an aspect model is a generic description of model families and it is specified using UML diagram template notation while a policy model is stated in terms of UML class diagram concepts with constraints that express restrictions given in a policy statement. [2]

realization mapping rules : To verify that an aspect model describing an access control

[2]In other words, an aspect model in this dissertation is defined at the M2 (metamodel) level of the four UML layers while a policy model is defined at the M1 (model) level.

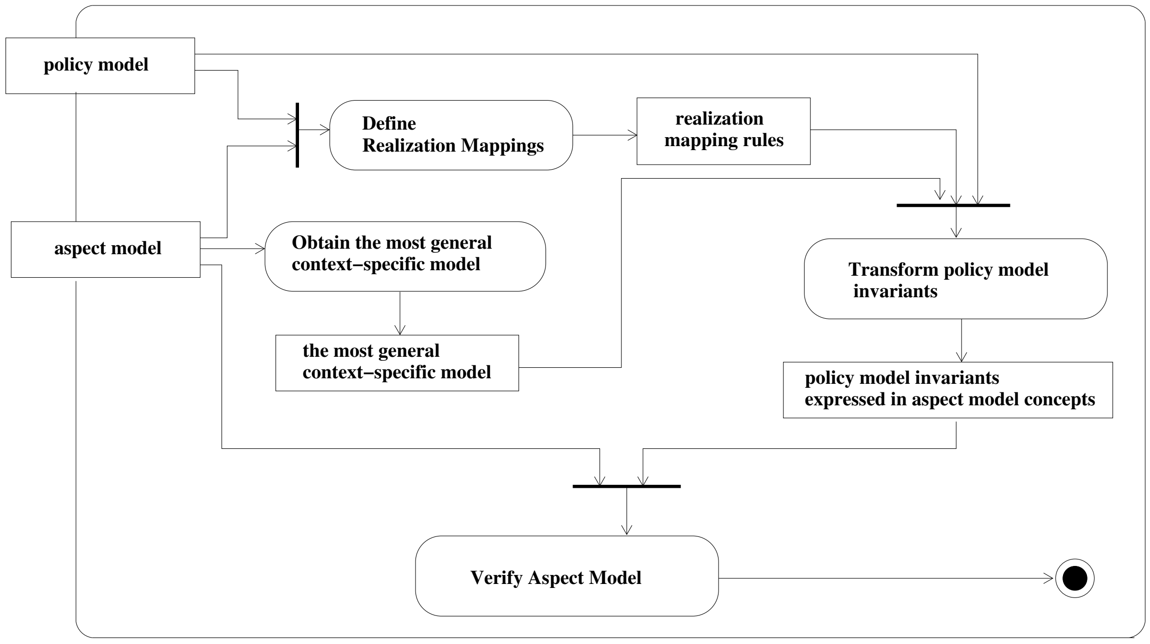

policy model
Define Realization Mappings
realization mapping rules
aspect model
Obtain the most general context–specific model
Transform policy model invariants
the most general context–specific model
policy model invariants expressed in aspect model concepts
Verify Aspect Model

Figure 4.6: Verifying a design aspect model against its policy model.

feature enforces targeted access control policies, we define realization mapping rules. A realization mapping in our approach is a set of pairs that defines how elements in the policy model are realized in the generalized aspect model. These mapping rules are used to transform the OCL invariants in the policy model into invariants expressed in terms of aspect model concepts.

the most general context-specific aspect model : We obtain a context-specific aspect model that is described at the M1 level so that OCL transformations can be performed between two M1-level models (i.e., one is a policy model and the other is a context-specific aspect model). Note that a context-specific aspect model is obtained by providing one or more domain-specific parameter values for each parameter in an aspect model (refer to Fig. 3.5). For example, for a banking application domain, *BankSession* and *BankRole* can be provided for the class templates *Session* and *Role* in Fig. 4.4 respectively. Since no domain-specific value set is available yet, we obtain the most general context-specific aspect model of an aspect model by providing a general set of parameter values for its template parameters as described below:

- provide a current parameter name as a parameter value for each template (e.g., class, attribute, operation, association templates) except multiplicity parameters on ends of association templates.
- provide a multiplicity indicator “$*$” that represents “zero or more” for each multiplicity template in an alphabet letter (i.e., the weakest form of multiplicity for an unconstrained multiplicity parameter).

transformed policy model invariants : Using realization mapping rules, invariants of an policy model are transformed into invariants expressed in terms of aspect model concepts.

enforcement verification : Verifying that the aspect model enforces the policy model involves establishing that transformed invariants hold in the aspect model.

When an RBAC policy model in Fig. 4.2 and an RBAC aspect model in Fig. 4.4, for example, are given as inputs to our verification procedure, realization mapping rules are defined in the form of set of realization pairs. A realization pair can be defined explicitly or implicitly. An explicit pair has the form {`policyElem`, *aspectElem*}, indicating that the policy model element `policyElem` is realized by an element *aspectElem* in the aspect model. The explicit realization pairs used in the verification of the RBAC aspect model reflect the simple one-to-one relationship between the major concepts shown in the policy model and the aspect model.

The following shows explicitly defined in realization pairs in the RBAC realization mapping:

{`Session`, *Session*}, {`User`, *User*}, {`Role`, *Role*}, {`Permission`, *Permission*}, {`Target`, *Target*}, {`OperationType`, *OperationType*}.

In the implicit form, one or both items in the pair are expressions that are evaluated. For example, the following is the realization pair that describes how the `UserSession` association in the RBAC policy model is realized in the aspect model:

{`u:User.userSession`, *u:User.UserSession*}.

In the above `u:User.userSession` represents the set of session objects associated with a user *u* in the policy model, and *u:User.UserSession* represents the set of sessions associated with the realization of *u* in the RBAC aspect model.

The following are implicitly defined in realization pairs in the RBAC realization mapping:

{`u:User.authorizedRole`, *u:User.GetAuthorizedRoles()*},

{`r:Role.allowedPermission`, *r:Role.Permission*},

{`s:Session.sessionUser`, *s:Session.UserSession*},

{`s:Session.activatedRole`, *s:Session.GetAllActivatedRoles()*},

{`p:Permission.allowedTarget`, *p:Permission.Target*},

{`p:Permission.allowedOp`, *p:Permission.OperationType*}.

The permissions associated with a session via the `hasPermision` association end in the policy model, is realized by the set of permissions associated with roles activated in a session. If we model the relationship between permissions and sessions in the RBAC aspect model as a derived association between *Session* and *Permission* that is named *SPermission*, then the following realization pair maps `hasPermission` links to *SPermission* links: {`s:Session.hasPermission`, *s:Session.SPermission*}.

Fig. 4.7 shows the most general context-specific RBAC class model that is obtained from the RBAC aspect model given in Fig. 4.4. *Session* and *Role*, for example, are provided as parameter values to instantiate the class templates *Session* and *Role* respectively and *CheckAccess* is provided as a parameter value for an operation template *CheckAccess*.

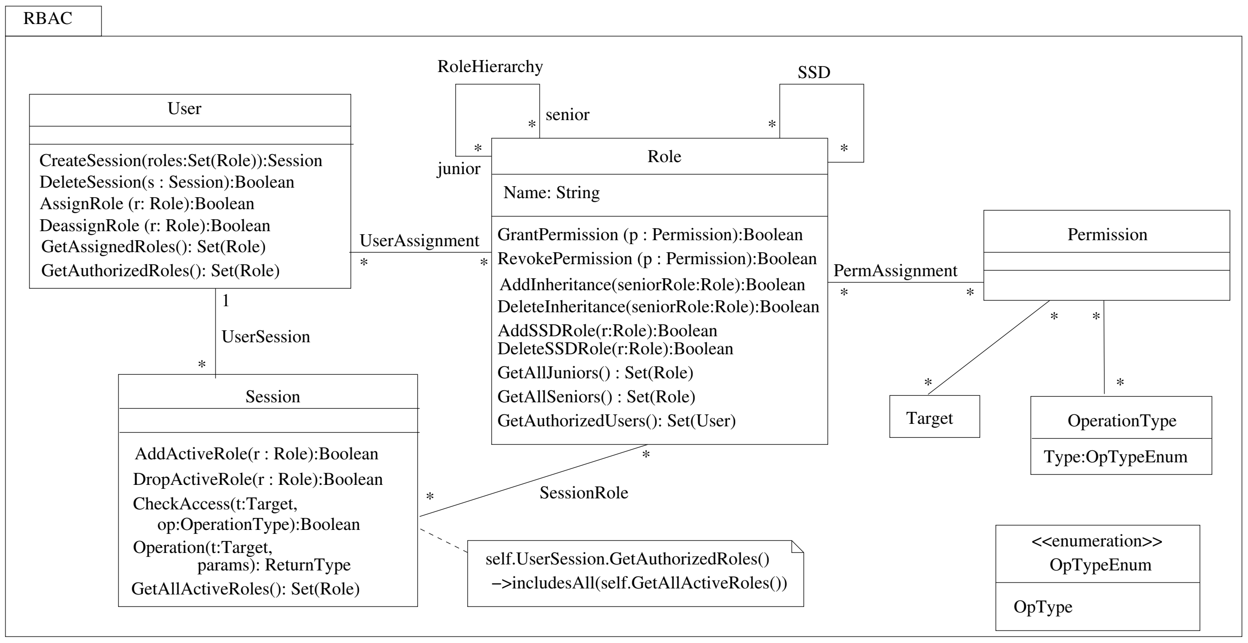

RBAC
RoleHierarchy
senior
junior
SSD
User
CreateSession(roles:Set(Role)):Session
DeleteSession(s : Session):Boolean
AssignRole (r: Role):Boolean
DeassignRole (r: Role):Boolean
GetAssignedRoles(): Set(Role)
GetAuthorizedRoles(): Set(Role)
UserAssignment
Role
Name: String
GrantPermission (p : Permission):Boolean
RevokePermission (p : Permission):Boolean
AddInheritance(seniorRole:Role):Boolean
DeleteInheritance(seniorRole:Role):Boolean
AddSSDRole(r:Role):Boolean
DeleteSSDRole(r:Role):Boolean
GetAllJuniors() : Set(Role)
GetAllSeniors() : Set(Role)
GetAuthorizedUsers(): Set(User)
PermAssignment
Permission
UserSession
Session
AddActiveRole(r : Role):Boolean
DropActiveRole(r : Role):Boolean
CheckAccess(t:Target, op:OperationType):Boolean
Operation(t:Target, params): ReturnType
GetAllActiveRoles(): Set(Role)
SessionRole
self.UserSession.GetAuthorizedRoles() −>includesAll(self.GetAllActiveRoles())
Target
OperationType
Type:OpTypeEnum
<<enumeration>>
OpTypeEnum
OpType

Figure 4.7: The most general context-specific RBAC aspect model

Therefore, the most general context-specific RBAC aspect model has an operation named *CheckAccess* in a class *Session*. For each multiplicity parameter expressed in an alphabet letter (e.g., the multiplicity parameter *a* on *Session* end of *UserSession* association of Fig. 4.4), we provide the multiplicity indicator "∗".

Using realization mappings shown above, the RBAC policy constraints P_{RBAC-1} and P_{RBAC-2} are transformed to the following:

$\mathbf{P}_{RBAC-1-transformed}$:

context Session **inv**:

SPermission→forAll(p:Permission|

GetAllActiveRoles()→exists(r|r.Permission→includes(p)) and

UserSession.GetAuthorizedRoles()→exists(r|r.Permission→includes(p)))

where a derived association *SPermission* is defined as follows:

context Session:: SPermission : Set(Permission)

derive: self.GetAllActiveRoles().Permission

P$_{RBAC-2-transformed}$:

context Session **inv**:

UserSession.GetAuthorizedRoles()→includesAll(GetAllActivatedRoles())

The behavior described by the *CheckAccess* template must enforce the transformed RBAC policy properties, P$_{RBAC-1-transformed}$ and P$_{RBAC-2-transformed}$. We show the policy enforcement below:

- From the definition of *SPermission*, if there is a permission in the set of permissions represented by *SPermission* that grants the access, an invocation of a *CheckAccess* operation with an argument *t* representing the target operation and an argument *op* representing the operation type, must return true; otherwise it returns false.

- Note that $s.CheckAccess(t, op) = true$ when the following expression in the postcondition of *CheckAccess* is true:

  self.GetAllActiveRoles().Permission → exists (p |

  p.Target → includes(t) and p.OperationType → includes(op))

- P$_{RBAC-2-transformed}$ holds because of the invariant shown in Fig. 4.7.

- Showing that P$_{RBAC-2-transformed}$ holds in the RBAC aspect model requires one to show that the following expression is true for all permissions in the set of permissions represented by *SPermission* (i.e., permissions that grant the access):

  GetAllActiveRoles()→exists(r|r.Permission→includes(p)) $--$ (1)

  and

  UserSession.GetAuthorizedRoles()→exists(r|r.Permission→includes(p)) $--$ (2)

  The postcondition of *CheckAccess* must return true for each permission *p* that grants the access. Therefore, the following expression holds:

GetAllActiveRoles().Permission→exists(p) $--(3)$

which is equivalent to

GetAllActiveRoles().exists(r|r.Permission→includes(p))

Therefore, (1) holds.

From the expression (3) and $P_{RBAC-2-transformed}$, we know the following expression holds:

UserSession.GetAuthorizedRoles().Permission→exists(p)

which is equivalent to

UserSession.GetAuthorizedRoles()→exists(r|r.Permission→includes(p))

Therefore, (2) holds.

- Therefore, two transformed RBAC policy properties, $P_{RBAC-1-transformed}$ and $P_{RBAC-2-transformed}$, holds in RBAC aspect model.

Verifying the BLP aspect model against its policy model proceeds as described for RBAC. We do not show the details of this verification because it uses techniques already covered in our discussion on verifying the RBAC aspect model.

4.4 Summary

In this chapter, we have given an AOM approach to modeling access control features that enforce access control policies separately as aspects and checking whether modeled aspects enforces given policies. To check that an aspect model enforces expected access control policies, access control policies need to be translated into the forms that access control aspects are specified in. In our approach, we propose to create a *policy model* that is a set of policies stated in terms of UML class diagram concepts with constraints and to obtain a context-specific aspect model that is the most general form of instantiation from the aspect model. The realization relationships between an aspect model and a policy model are described in the form of realization mapping rules. Using these rules, we transform the OCL invariants in the policy model to invariants expressed in terms of concepts in a general context-specific aspect model and verify the required policy enforcement by establishing that transformed invariants hold in the given aspect model.

As examples of access control policies, RBAC and BLP policy requirements were represented by RBAC policy model and BLP policy model respectively and an RBAC aspect model and a BLP aspect model were formulated. As a verification example, the RBAC aspect model was verified against RBAC policies given in [27].

Chapter 5

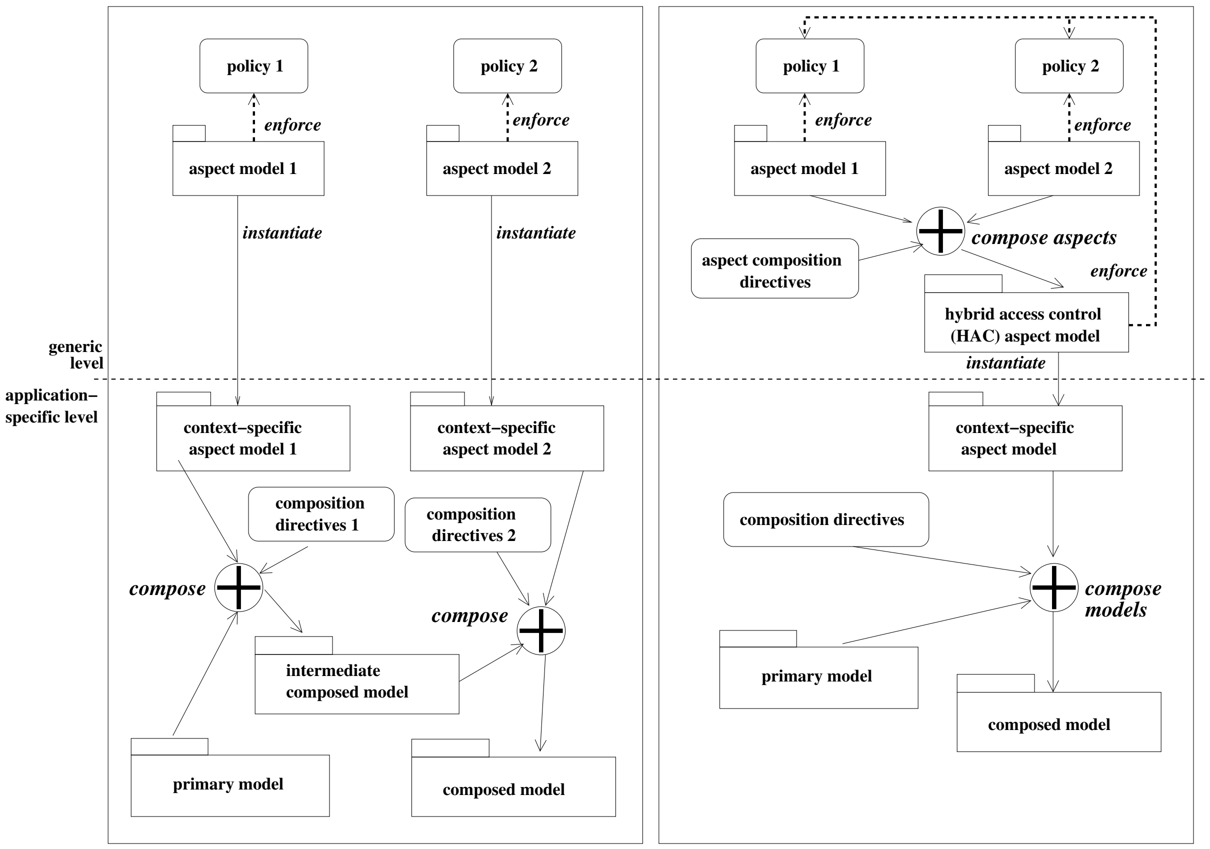
Composing Access Control Aspect Models: An Example

Complex applications typically must enforce more than on access control policies. In AOM approach, each aspect model is typically applied to a primary model sequentially, that is, one at a time. Another approach is to compose multiple aspect models with a primary model that addresses the drawbacks of the sequential approach. In the approach the aspect models are composed to produce a single hybrid aspect model. This hybrid aspect model can then be composed with a primary model to produce a design that enforces the integrated set of policies. Fig. 5.1, for example, illustrates two approaches to composing multiple aspect models with a primary model.

In the approach illustrated in Fig. 5.1(a) each aspect model is instantiated to obtain a corresponding context-specific aspect model. The context-specific models are composed one at a time with the primary model. This approach has the following drawbacks:

- It is difficult to understand and analyze the interactions between the aspects independently of the primary model.
- The approach does not allow one to leverage commonalities in how aspects are composed when the same set of aspects are composed with more than one primary model.

In the approach shown in Fig. 5.1(b) two aspect models are first composed to obtain a composed aspect model, which is a hybrid access control (HAC) aspect model that de-


policy 1
policy 2
enforce
aspect model 1
aspect model 2
instantiate
generic level
application–specific level
context–specific aspect model 1
context–specific aspect model 2
composition directives 1
composition directives 2
compose
intermediate composed model
primary model
composed model
compose aspects
aspect composition directives
hybrid access control (HAC) aspect model
context–specific aspect model
composition directives
compose models

(a) Applying aspects without aspect composition (b) Applying aspect with aspect composition

Figure 5.1: Two approaches to incorporating access control aspects into a primary model

scribes the integration of two access control features. The HAC aspect model is then instantiated, and the resulting context-specific aspect model is composed with the primary model. The approach illustrated in Fig. 5.1(b) has the following advantages over the approach shown in Fig. 5.1(a):

- The HAC aspect is specified separately from the application which makes understanding and analyzing the HAC easier.
- The HAC aspect can be reused across a number of different applications reducing the time and effort needed to perform the composition. If two aspects are to be composed with n primary models, then the approach shown in Fig. 5.1(a) requires the composition process to be applied $2 * n$ times. The approach shown in Fig. 5.1(b)

requires the composition process to be applied $n+1$ times: one for the composition of the aspect models and n for compositions of the composed aspect model and the primary models.

The aspect composition approach illustrated in Fig. 5.1(b) supports understanding and analyzing the aspects in isolation. Such analysis can help determine whether the right combination of aspects was selected before the aspects are composed with a primary model. For example, consider a situation in which a software design that addresses auditing and access control concerns is needed. There are a number of alternative features that address these concerns and a pairing of an auditing feature with an access control feature may produce undesirable emergent behavior (e.g., auditing may provide access to information that is under access control). It helps if the interactions across the access control and auditing features can be analyzed and understood before the features are composed with the primary model.

The approach also leverages commonalities to reduce the number of composition steps needed to compose the aspects with multiple primary models: Instead of carrying out multiple sequential compositions one only has to compose the composed aspect with a primary model (the composition of aspects is done only once, and the result is used across multiple primary models). The reduction in time and effort is significant when the composed aspect is used in many different applications. We illustrate our approach by composing two access control features: a feature that realizes the Bell-LaPadula (BLP) model [79] and a feature that realizes the Role-Based Access Control (RBAC) model [27]. The composed aspect can be used to develop designs that must enforce both BLP and RBAC policies (e.g., designs of applications in the military domain).

5.1 Overview

Fig. 5.2 gives an overview of our policy-based aspect composition approach. The aspect

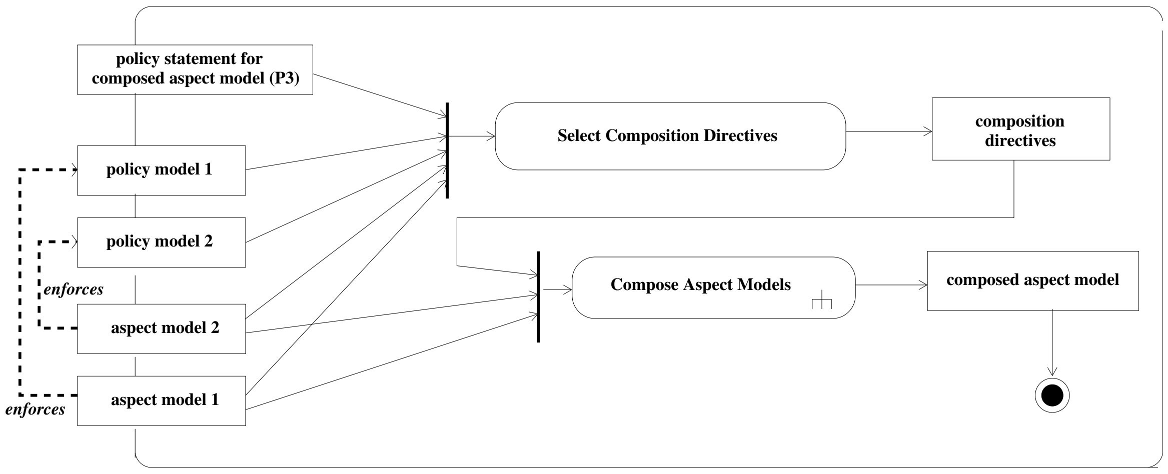


Figure 5.2: An aspect composition approach

composition approach proceeds as follows:

1. **Verify enforcement of policies in aspect models (refer to Chapter 4)** : The first step is to verify that each individual aspect models to be composed enforces the given policy. We use refinement techniques that are adapted to the UML class model for this purpose. From the verification, a modeler obtains the knowledge of the relationships between original policies and the integrated policy that the composed aspect model must enforce.

2. **Select composition directives** : The composition directives are selected by a modeler using the knowledge of how the policies enforced by the aspect models can be combined to obtain policies that must be enforced by the composed aspect model. A model specify and use composition directives to ensure that composition produces required results. We do not discuss how such knowledge is obtained (it can be based on human expertise); we show only how the knowledge is utilized to select appropriate composition directives.

3. **Compose aspect models** : The aspect models are composed using the composition directives selected in the previous step. A more detailed view of the composition pro-

cess is presented in the next section.

5.2 Aspect Compositions

Our aspect composition approach is based on the procedure that we described earlier for composing UML models [31, 72]. The composition procedure consists of a basic merging procedure that implements default rules for merging model elements and the use of composition directives. The composition directives are used for overriding the default rules and modifying models before and after basic model merging. In order to use the composition procedure we treat aspect models as UML models. This is done by syntactically treating the parameters in the aspect models as model element names. From a semantic viewpoint, the result of composing two aspect models is not a UML model but an aspect model that must be instantiated before it can be merged with a UML model.

We first present the basic merging procedure used to merge aspect models and give examples of composition directives that can be used in conjunction with the basic procedure to vary how models are composed.

5.2.1 The Basic Merging Procedure

The aspect composition approach uses a name-based matching procedure in which elements with the same name are merged in the composed model [31, 72] One model is considered to be the dominant model for the purpose of resolving problems that can arise when merging matching model elements with conflicting properties. Model elements in the dominant model are referred to as dominant model elements. In the cases where name-based merging of property values leads to conflicts the default is to have the properties of the dominant model element override the properties of the matching model element in the non-dominant model. For example, if the *isAbstract* property of a class named C in the dominant model has the value $true$ and the property has the value $false$ for a similarly named class in the non-dominant model, then the composed model will contain the class C

with the value $true$. Composition directives can be used to change the results produced by the default name-based merge procedure.

The basic merging procedure and examples of default merging rules associated with model element types are given below (aspect models are simply referred to as models in the following).

Algorithm 1 *Basic Model Merging Procedure*

Step 1 *For each model element in the dominant model, search for a model element in the non-dominant model with the same name. Elements with matching names are assumed to represent different views of the same concept and thus are intended to be merged.*

Step 2a *If a matching element is found then it is merged according to default merging rules associated with the model element type.*

Step 2b *If a matching element is not found then the dominant model element is included in the composed model.*

Step 3 *Elements in the non-dominant model that are not matched with elements in the dominant model are included in the composed model.*

In the following we outline some of the rules for merging class model elements.

Algorithm 2 *Default Rules for Merging Matching Classes*

Rule 1 *If the values associated with Class attributes (i.e., attributes of Class in the UML metamodel, for example, $isAbstract$) are different then the default rule is to use the value in the dominant model. This rule can be overridden by composition directives.*

Rule 2 *Attributes of matching classes with identical names and data types are merged. If the matching attributes are associated with OCL invariants, the conjunction of the invariants is associated with the merged attribute in the composed model. This rule can be changed using composition directives.*

Rule 3 *Operations with identical names are merged. If matching operations have different argument lists, the merged operation in the composed model will have a list of arguments formed by appending the list of arguments in the non-dominant model to the list of arguments in the dominant model. If the matching operations are associated with OCL specifications, then the specification associated with the merged operation in the composed model is formed as follows: the precondition is the disjunction of the preconditions associated with the matching operations, and the postcondition is the conjunction of the postconditions associated with the matching operations.*

Algorithm 3 *Default Rules for Merging Associations*

Rule 1 *Association ends match when they have the same role end name. Alternatively, two associations match when they have the same name (we require that associations have either association names or role names at each end - if they have both, the role names are used to determine matches).*

Rule 2 *If matching association ends have different multiplicities, then the multiplicity in the dominant model is used.*

These rules can be overridden by composition directives.

5.2.2 Composition Directives

Use of the basic name-based merging procedure is not likely to produce desired results in all cases. Composition directives can be used in conjunction with the merging procedure to ensure that desired models are produced [86]. Composition directives can be broadly classified as follows:

Pre-Merge Directives: These directives are used to modify models before they are merged by the basic merging procedure. These directives define transformations on the models that are to be composed.

Merge Override Directives: These directives are used to override the default merging rules.

Post-Merge Directives : These directives are used to modify the model that is produced by the basic merge procedure. These directives define transformations on the composed model.

Below we summarize the directives used in this work:

Table 5.1: A partial list of composition directives

Name (Directive Type)	Description	Syntactic Form
rename (Pre-Merge)	A model element is renamed to the given new name and all occurrences of the old name are changed to the new name.	**rename** *owner*::*targetElement* **to** *newName*
strengthenPreSpec (Merge Override)	The default rule for merging preconditions of operations, *disjunction of preconditions*, is overridden by *conjunction of preconditions*.	**strengthenPreSpec in merging** *Aspect*1::*Class*1::*Operation*1 **and** *Aspect*2::*Class*1::*Operation*1

CreateAssociation
(Post-Merge)
	An association is created between two existing classes. Note that each association end must be created prior to creating an association. The created association is not a member of any namespace yet.	*newAssociation* = **CreateAssociation** {
		name = "*newAssociation*",
		isDerived = true|false,
		memberEnd =
		[*newAssocEnd*1, *newAssocEnd*2]}

CreateProperty
(Post-Merge)
	An association end is created. The created association end is not a member of any namespace yet.	*newAssocEnd*1 = **CreateProperty** {
		isComposite = true|false,
		aggregation = none|composite,
		type = *aspect*1::*class*1,
		opposite = *newAssocEnd*2,
		lower = *non_negative_integer*,
		upper = *non_negative_integer*|'*'}

add (Post-Merge)	The given model element is added to the specified owner namespace.	**add** *owner*::*elem*
replacePreSpec (Post-Merge)	The precondition of operation is replaced by the given OCL expression.	**replacePreSpec** *Class*1::*Operation*1 **with** {*ocl_expression*}

Continued on next page

Table 5.1 – continued from previous page

Name	Description	Syntactic Form
replacePostSpec (Post-Merge)	The postcondition of operation is replaced by the given OCL expression.	**replacePostSpec** $Class1::Operation1$ **with** $\{ocl\_expression\}$
addPreSpec (Post-Merge)	The given OCL expression is added to the operation precondition by conjunction.	**addPreSpec** $\{ocl\_expression\}$ **to** $Class1::Operation1$
addPostSpec (Post-Merge)	The given OCL expression is added to the operation postcondition by conjunction.	**addPostSpec** $\{ocl\_expression\}$ **to** $Class1::Operation1$

In the following section, we present the HAC policies and show how they are used to determine the composition directives that are needed to ensure that composition produces a HAC model that enforces the policies.

5.3 An Example of Composing RBAC and BLP Aspect Models

To compose the RBAC model and the BLP model, one first determines the model elements that represent the same concepts across the two models. These model elements must have the same name in both models if they are to be merged using the basic merging procedure. If model elements representing the same concept have different names then the rename directive is used to rename one or both of the model elements so that they have the same name. The rename directive is also used to change the name of a model element that has the same name as another model element that represents a different concept. In the BLP and RBAC aspect models, $Subject$ and $Session$ represent the same concept [76]. The rename directive is used to rename $Subject$ to $Session$ and to change all occurrences of the name $Subject$ to $Session$ in the BLP model. Also, the $CreateSubject$ and $DeleteSubject$ operations in $User$ and the $UserSubject$ association are renamed to $CreateSession$, $DeleteSession$, and $UserSession$ respectively. The following rename directives rename identified model elements accordingly:

rename BLP::Subject **to** Session
rename BLP::User::CreateSubject **to** CreateSession
rename BLP::User::DeleteSubject **to** DeleteSession
rename BLP::UserSubject **to** UserSession

These rename directives are *pre-merge* directives and thus are applied before applying the basic merge procedure.

In the HAC aspect model the *CreateSession* precondition must be the conjunction of the preconditions associated with *CreateSession* in both the RBAC and renamed BLP models. The default rule is to form the disjunction and thus a composition directive is needed to override the rule. The *merge override* directive strengthenPreSpec is used for this purpose, as shown below:

strengthenPreSpec in merging RBAC::User::CreateSession
and BLP::User::CreateSession

The other composition directives needed to produce the required HAC aspect model are identified by examining the policies that must be enforced by the HAC aspect model. These directives are applied after the RBAC and renamed BLP aspect models are merged using the basic merging procedure (i.e., they are post-merge directives). We refer to the model produced using the basic merge procedure as the *preliminary* HAC aspect model. The *post-merge* directives add an association to the composed aspect model and replace operation specifications in the preliminary HAC aspect model with specifications that enforce the required HAC policies.

The HAC policies given below are obtained from the RBAC and BLP policies:

$\mathbf{P}_{HAC}$:

$\mathbf{P}_{RBAC-1}$: If a user *u* has permission to access a target *t* to perform an operation with an operation type op in a session s, then there exists a role r such that r has permission to access t to perform an operation with a type of op, r is an authorized role for u, and r is currently activated in s (a property enforced by *CheckAccess* of RBAC).

$\mathbf{P}_{RBAC-2}$: Roles activated in a session s must be a subset of the roles assigned to the user u of s.

$\mathbf{P}_{BLP-1-Read}$: If a user u has permission to read from a target t in a subject sb, then the type of an operation op is a read type and the security level of sb dominates the security level of t (a property enforced by *CheckAccess* of BLP).

$\mathbf{P}_{BLP-1-Write}$: If a user u has permission to write to a target t in a subject sb, then the type of an operation op is a write type and the security level of sb is equal to the security level of t (a property enforced by *CheckAccess* of BLP).

$\mathbf{P}_{BLP-2}$: If a user u has permission to access a target t to perform an operation with an operation type op in a subject sb, then sb is a subject for u and the security level of u dominates the security level of sb (a property enforced by *CheckAccess* of BLP).

$\mathbf{P}_{Role-User}$: The security level of u must dominate the security level of the role that was assigned to u (role assignment is handled by *AssignRole* of RBAC).

$\mathbf{P}_{Role-Session}$: The security level of s must be equal to the security level of the role that was activated for s (role activation is handled by *CreateSession* and *AddActiveRole* of RBAC).

The first five policies given above, $\mathbf{P}_{RBAC-1}$, $\mathbf{P}_{RBAC-2}$, $\mathbf{P}_{BLP-1-Read}$, $\mathbf{P}_{BLP-1-Write}$, and $\mathbf{P}_{BLP-2}$, are enforced by the *CheckAccess* operations in RBAC and BLP. The HAC will also have a *CheckAccess* operation that enforces its policies. This operation can be obtained by merging the *CheckAccess* operations of both RBAC and BLP such that the desired policies are enforced. Applying the default operation merge rules results in the desired argument list and thus a composition directive is not needed to modify the argument list. The default operation rules though do not provide the needed operation specification for HAC's *CheckAccess*. The postcondition of RBAC's *CheckAccess* has the form $(result = P_{RBAC-1} and P_{RBAC-2})$ and the postcondition of BLP's *CheckAccess* has the form $(result = P_{BLP-1-Read}\ and\ P_{BLP-1-Write}\ and\ P_{BLP-2})$. The HAC's *CheckAccess* must have the form $(result = P_{RBAC-1} and P_{RBAC-2}\ and\ P_{BLP-1-Read}\ and\ P_{BLP-1-Write}\ and\ P_{BLP-2})$, but the preliminary HAC model has the following postcondition instead: $(result = P_{RBAC-1} and P_{RBAC-2})\ and\ (result = P_{BLP-1-Read}\ and\ P_{BLP-1-Write}\ and\ P_{BLP-2})$. A composition directive is needed to change this specification.

The post-merge composition directive that replaces the specification of *CheckAccess* in the preliminary HAC model is given below:

```
replacePostSpec Session::CheckAccess
    with {result = self.GetAllActiveRoles().Permission → exists (p |
         p.Target → includes(tar) and p.OperationType → includes(op)))
         and
         (op.Type = Read and
            self.SecurityLevel.GetAllDominatees()→includes (t.SecurityLevel))
            or (op.Type = Write and self.SecurityLevel = t.SecurityLevel)}
```

Enforcing the policies P$_{Role-User}$ and P$_{Role-Session}$ in P$_{HAC}$ can be accomplished by defining an appropriate relationship between *Role* and *SecurityLevel*. The policies can then be expressed in terms of this relationship. The *post-merge* directives create and add shown in Table 5.1 can be used to add an association between *Role* and *SecurityLevel* in the prelimnary HAC aspect model. The association is shown as a dark line in Fig. 5.3. The following directives create an association named *RoleSecurityLevel* between *Role* and *SecurityLevel* with two associstion ends, *RoleSecLevelEnd* and *SecLevelRoleEnd*:

RoleSecLevelEnd = **createProperty** {isComposite = false, aggregation = none,
    type = HAC::Role, opposite = SecLevelRoleEnd, lower = x1, upper = x2}
SecLevelRoleEnd = **createProperty** {isComposite = false, aggregation = none,
    type = HAC::SecurityLevel, opposite = RoleSecLevelEnd, lower = 1, upper = 1}
RoleSecurityLevel = **createAssociation** {name = "RoleSecurityLevel",
    isDerived = false, memberEnd = [RoleSecLevelEnd, SecLevelRoleEnd]}

Both association ends are non-composite and non-aggregation ends. The multiplicity in *RoleSecLevelEnd* is parameterized (i.e. $x1$ .. $x2$), but the multiplicity in *SecLevelRoleEnd* is '1' because a role must be associated with only one security level. These three model elements are added to the HAC by the following directives:

add RoleSecurityLevel
add Role::RoleSecLevelEnd
add SecurityLevel::SecLevelRoleEnd

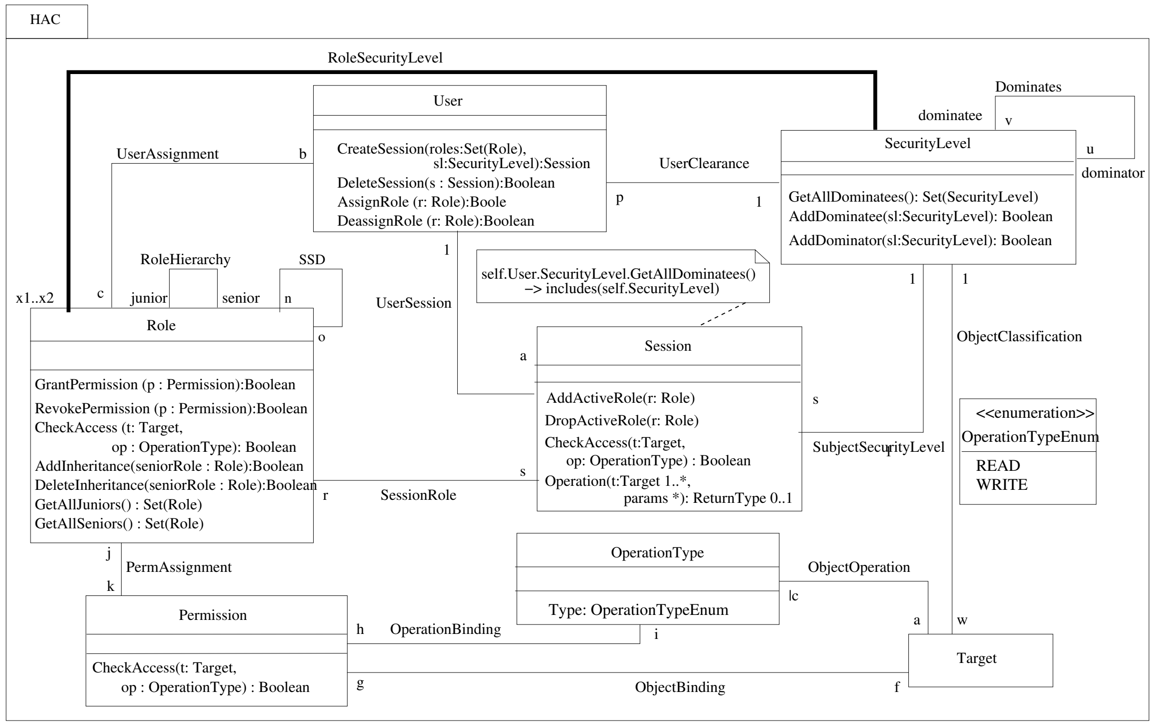

HAC
RoleSecurityLevel
User
CreateSession(roles:Set(Role), sl:SecurityLevel):Session
DeleteSession(s : Session):Boolean
AssignRole (r: Role):Boole
DeassignRole (r: Role):Boolean
UserAssignment
b
UserClearance
p
1
Dominates
dominatee
v
u
dominator
SecurityLevel
GetAllDominatees(): Set(SecurityLevel)
AddDominatee(sl:SecurityLevel): Boolean
AddDominator(sl:SecurityLevel): Boolean
1
self.User.SecurityLevel.GetAllDominatees() –> includes(self.SecurityLevel)
RoleHierarchy
SSD
x1..x2
c
junior
senior
n
o
UserSession
Role
GrantPermission (p : Permission):Boolean
RevokePermission (p : Permission):Boolean
CheckAccess (t: Target, op : OperationType): Boolean
AddInheritance(seniorRole : Role):Boolean
DeleteInheritance(seniorRole : Role):Boolean
GetAllJuniors() : Set(Role)
GetAllSeniors() : Set(Role)
a
Session
AddActiveRole(r: Role)
DropActiveRole(r: Role)
CheckAccess(t:Target, op: OperationType) : Boolean
Operation(t:Target 1..*, params *): ReturnType 0..1
s
SubjectSecurityLevel
t
1
1
ObjectClassification
<<enumeration>>
OperationTypeEnum
READ
WRITE
s
r
SessionRole
j
PermAssignment
k
OperationType
Type: OperationTypeEnum
ObjectOperation
c
a
w
Permission
CheckAccess(t: Target, op : OperationType) : Boolean
h
OperationBinding
i
Target
g
ObjectBinding
f

Figure 5.3: Class Model Template View of the Hybrid Access Control Aspect Model

The existence of the new association allows one to state the policies $P_{Role-User}$, $P_{Role-Session}$, and $P_{Role-Target}$ in P_{HAC} property as follows:

- **$P_{Role-User}$**: A user's security level must dominate the security level of the role that was assigned to that user.

  context Session

  inv: self.User.Role→forAll(r|self.User.SecurityLevel.GetAllDominatees()

  →includes(r.SecurityLevel))

- **$P_{Role-Session}$**: A session's security level must be equal to the security level of the role that was activated for that session.

  context Session

  inv: self.Role→forAll(r|self.SecurityLevel = r.SecurityLevel)

HAC operations that create or destroy links between (1) users and security levels, (2) users and roles, (3) sessions and roles, and (4) sessions and security levels must ensure that the above policies are enforced. Therefore, the specifications associated with the following HAC operations must be changed: *User::AssignRole*, *Session::AddActiveRole*, *User::CreateSession*. The property stated in $P_{Role-User}$ should be checked before an *AssignRole* operation is invoked, and $P_{Session-User}$ should be checked before *AddActiveRole* or *CreateSession* operations are invoked. The *addPreSpec* directive is used to modify the preconditions of these operations accordingly:

addPreSpec {$P_{Role-User}$} **to** User::AssignRole
addPreSpec {$P_{Role-Session}$} **to** Session::AddActiveRole
addPreSpec {$P_{Role-Session}$} **to** User::CreateSession

Applying these post-merge directives results in the following specifications:

context Session::CheckAccess(t:Target, op:OperationType):Boolean

pre: true

post: result =

self.GetAllActiveRoles().Permission → exists (p |

p.Target → includes(tar) and p.OperationType → includes(op)))

and

(op.Type = Read and

self.SecurityLevel.GetAllDominatees()→includes (t.SecurityLevel))

or (op.Type = Write and self.SecurityLevel = t.SecurityLevel)

The new specifications associated with *AssignRole*, *AddActiveRole*, and *CreateSession* operations are given below (added OCL expressions are shown in bold):

context User :: AssignRole (r : Role) : Boolean

pre: self.Role→excludes (r)

and self.GetAuthorizedRoles()→forAll(r1|r1.SSD → excludes (r))

and

self.Role→forAll(r|self.SecurityLevel.GetAllDominatees()→includes(r.SecurityLevel)

post: self.Role=self@pre.Role→including(r)

context Session :: AddActiveRole (r : Role) : Boolean

pre: self.Role→excludes (r)

and self.User.GetAuthorizedRoles()→includes(r)

and

self.Role→forAll(r|self.SecurityLevel = r.SecurityLevel

post: self.Role=self@pre.Role→including (r)

context User::CreateSession(roles:Set(Role), sl:SecurityLevel):Session

pre:

self.SecurityLevel.GetAllDominatees()→includes(sl)

and

self.GetAuthorizedRoles()→includesAll(roles)

and

self.Role→forAll(r|self.SecurityLevel = r.SecurityLevel

post: result.oclIsNew()

and self.Session=self@pre.Session→including(result)

and result.Role= roles

and

result.oclIsNew()

and self.Session=self@pre.Session→including(result)

and result.SecurityLevel = sl

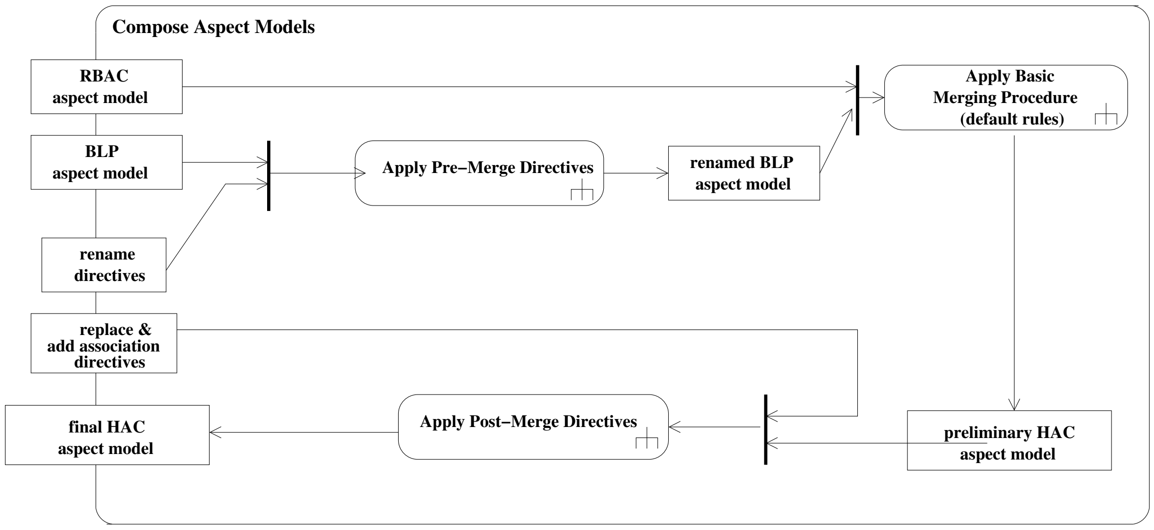

Compose Aspect Models
RBAC aspect model
BLP aspect model
rename directives
replace & add association directives
final HAC aspect model
Apply Pre−Merge Directives
renamed BLP aspect model
Apply Basic Merging Procedure (default rules)
preliminary HAC aspect model
Apply Post−Merge Directives

Figure 5.4: Overview of RBAC and BLP Composition

Fig. 5.4 is an expansion of the *Compose Aspect Models* activity shown in Fig. 5.2 that summarizes the activities involved in composing the RBAC and BLP aspect models. The pre- and post-merge composition directives and the RBAC and BLP aspect models are the inputs to the composition activity. The pre-merge directives are used to rename elements in the BLP aspect model as described in this section. The renamed BLP aspect model is then merged with the RBAC aspect model to produce a preliminary HAC model. The post-merge directives are then applied to the preliminary HAC aspect model. These directives change the operation specifications and add an association to ensure that the required policies are enforced in the composed aspect model.

5.4 Summary

In this chapter, we have demonstrated an approach to using composition directives to integrate two aspect models with an example from the military domain which requires applications that enforce integrated RBAC and BLP policies. The composed aspect describes the integration of RBAC feature and BLP features independently of the primary model. One advantage is that the interactions of the aspects can be understood in isolation without requiring the comprehension of the application. The second advantage is that it reduces the number of times the composition process must be applied – this is significant when the composed aspect is used in several applications.

Chapter 6

A Verifiable Model Composition Approach

In this chapter, we illustrate how the composition of the aspect models and a primary model can be carried out in a verifiable manner. A key issue in applying the AOM approach has been determining whether composition of aspect models and a primary model produces a composed model that has specified properties. In the previous AOM composition approach [31, 69], the composed model is analyzed after the composition of context-specific aspect models and the primary model in order to uncover emergent behaviors that result in violations to desired properties or identify undesirable interactions between aspect functionality and other behaviors described in the primary model. However the problem with this post-composition analysis is that it is difficult to identify which part of composition caused the problem because the composition is already completed.

We extend the previous model composition approach to support the generation of proof obligations that must be discharged in order to establish that a desired property holds in the composed model. In our composition approach, class diagrams are merged first, but the entire composition does not end until after the proof obligation is generated and evaluated.

6.1 Overview

Fig. 6.1 shows a high level overview of our verifiable model composition approach. The

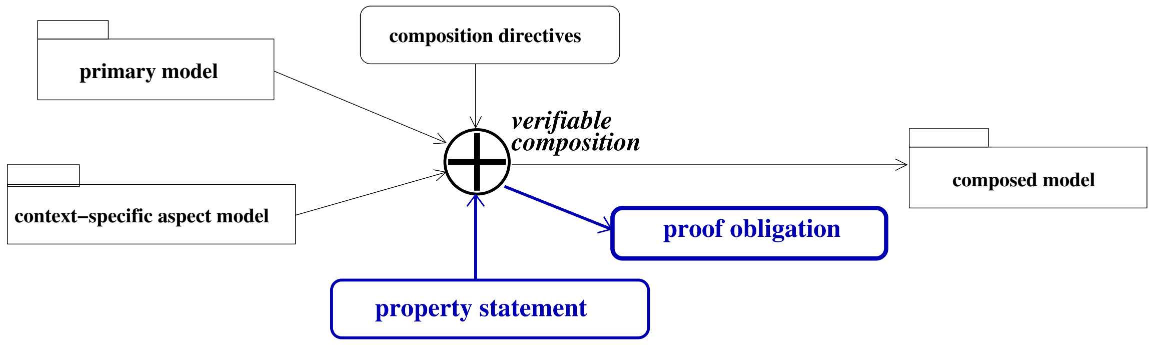


Figure 6.1: An overview of verifiable composition in the AOM approach

verifiable model composition in our approach is accomplished in the following two steps[1]:

1. **merging class diagrams** : Composition of an aspect and a primary model involves merging a context-specific aspect model[2] and a primary model. We use the name-base class model composition technique by France *et al.* [31] to match model elements and bring matched elements together to have the merged class diagram. Unmatched model elements shown in one model are added to the merged class diagram as well unless it is differently directed by any of composition directives. Other changes to default merging process can be indicated by composition directives.

2. **generating and evaluating proof obligation** : From the property to verify that is specified in OCL and the specification of an operation in the merged class diagram, the proof obligation is generated and evaluated. Discharging the proof obligation can help one identify the sources of problems when the obligations do not hold.

[1]Note that the composition in our approach is a black-box composition in that users don't see any other intermediate composition steps except the merged class model and generated proof obligation.

[2]A context-specific aspect model is obtained by instantiating aspect model using application specific values. For details on aspect instantiation, refer to Section 3.5.

The types of correctness checks that can be carried out on a model are determined by the types of formally stated properties in the model and the types of derivations that can be supported by the properties. The properties targeted by our approach are concerned with the effects of operations. Operation specifications in UML models contain preconditions[3] and postconditions[4] expressed in the Object Constraint Language (OCL) and thus can support checking of behavioral properties that can be stated in terms of the effects of operations in object states, which are specified in terms of preconditions and postconditions.

To specify that a message invocation event has taken place, the messaging expression that includes the `isSent` operator (denoted as ^)is used in postconditions. This operator takes a target object *o* and a message as *Opr* as operands. This is shown below:

context ClassName::Op()
  pre: P_{pre}
  post:
    o^Opr().hasReturned() and o^Opr().result() = true

The above postcondition specifies a message event in an interaction diagram, which states that a message event *Opr* has been sent to an object *o* and the postcondition becomes true if the message event has already finished executing and has returned a true boolean value. When all message invocations between objects are explicitly specified in the postconditions of the operation as shown above, interaction diagrams, such as sequence diagrams, can be derived from those operation specifications.

Therefore, our verifiable composition technique requires the operation specification specified in one of the following format, where *Op* is a triggering operation of a sequence

[3]A precondition is a boolean expression that must be true at the moment that the operation starts its execution [91].

[4]A postcondition is a boolean expression that must be true at the moment that the operation ends its execution [91].

diagram in the merged class model and Opr_i represents a set of sub-operations that are invoked by Op:

- For an operation Op that returns a value,

```
context ClassName::Op(p_1:T_1,...,p_n:T_n):T
  pre: P_pre
  post:
    result =
      o_i^Opr_i(p_i).hasReturned() and o_i^Opr_i(p_i).result() = Val      -- Expr_msg-i
      and
      Q                                                                   -- Expr_non-msg
```

- For an operation Op that returns no value,

```
context ClassName::Op(p_1:T_1,...,p_n:T_n)
  pre: P_pre
  post:
      o_i^Opr_i(p_i).hasReturned()                           -- Expr_msg-i
      and
      Q                                                      -- Expr_non-msg
```

In either specification format shown above, the part denoted by $Expr_{msg-i}$ states conditions under which messages are sent by the sub-operations Opr_i, and the part denoted by $Expr_{non-msg}$ represents the remainder part of expression that does not have any message invocation. $Expr_{msg-i}$ can be simply represented by "x_i.hasReturned()" if a sub-operation Opr_i has no return value.

The property to verify is written based on a triggering operation of which behavior affects the property. A sequence diagram is going to describe the sequence of messages that will be invoked by a triggering operation under our interest. Therefore, this style of writing operation pre- and postconditions makes it possible for one to generate proof obligations as a sequence diagram is obtained.

In our approach, an operation specification P_{beh} and the proof obligation PO are defined as follows:

Definition 1 *An operation specification P_{beh} is a condition that must be satisfied immediately after execution of an operation. Formally,*

$P_{beh} = P_{pre}@pre$ *and* P_{post}

where $P_{pre}@pre$ denotes the expression obtained from P_{pre} by replacing each property name p occurring in P_{pre} by p@pre.

For example, consider the following specification for the operation *LowerByOne* in an object of the class *A* that has an attribute named *attr*:

```
context A::LowerByOne()
    pre: attr > 0
    post: attr = attr@pre - 1
```

attr in the precondition is replaces by *attr@pre* and thus the operation behavior specification (P$_{beh}$) is defined as follows :

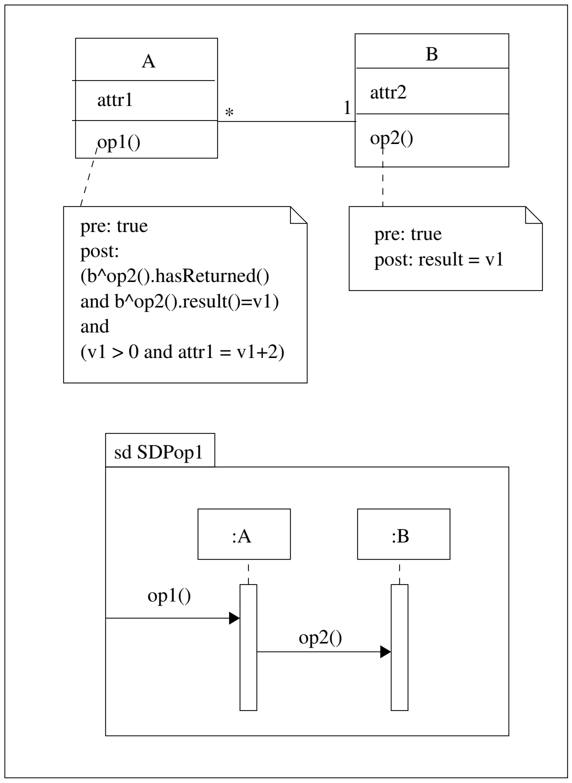
P$_{beh}$ = attr@pre >0 and attr = attr@pre - 1

Definition 2 *The proof obligation* PO *is defined as* P$_{beh}$ implies P$_{prop}$, *where P_{beh} is an operation specification in an merged class model and P_{prop} is the property to be verified.*

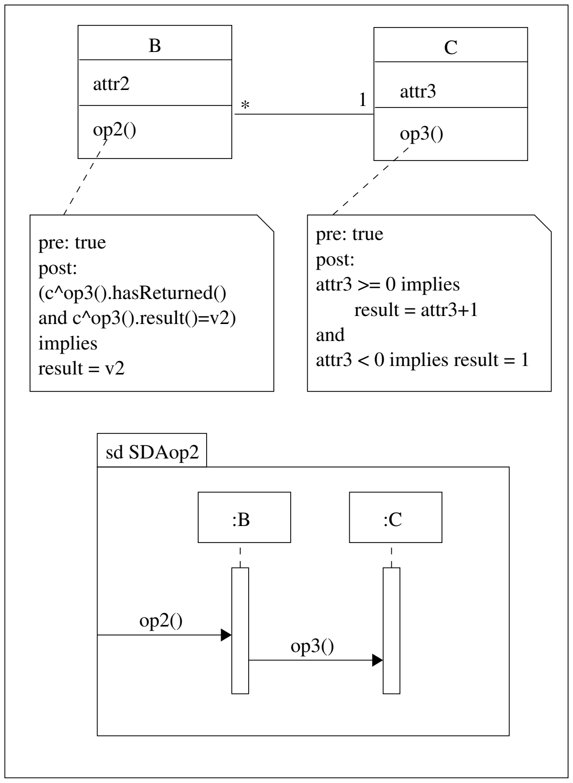
Verification of the property requires discharging this implication obtained as a *PO*.

6.2 A Composition Example

In this section we illustrate how the composition of two models can be verified using our composition approach. Consider the following example models given in Fig. 6.2. Fig. 6.2(a) shows a simple primary model that has two operations *op*1 and *op*2. The class model shown in Fig. 6.2(a) will be merged with an aspect class model given in Fig. 6.2(b) according to the class diagram composition approach proposed by Straw *et al.* [86]. A composition directive *replacePostSpec* will be applied to replace the postcondition that has



(a) Primary model



(b) Aspect model

(c) Composition directives

```
replacePostSpec B::op2 with  {(c^op3().hasReturned()
                              and c^op3().result()=v2)
                              implies
                              result = v2}
```

Figure 6.2: partial class/sequence diagrams for example models

been obtained by merging two matching operations in the class *B*. Fig. 6.3 is obtained as a merged class diagram.

Fig. 6.4 shows an overview of our verifiable composition approach. The composition of the primary model class diagram and the aspect model class diagram are accomplished through the named activity *Merge Class Diagrams* ((1) in Fig. 6.4) according to the class diagram composition algorithm proposed by [86]. In Fig. 6.4, specifying the given property statement in OCL provides the property to be verified denoted as P_{prop} (refer to the action (1) in Fig. 6.4). The operation behavior in the composed model needs to be verified against

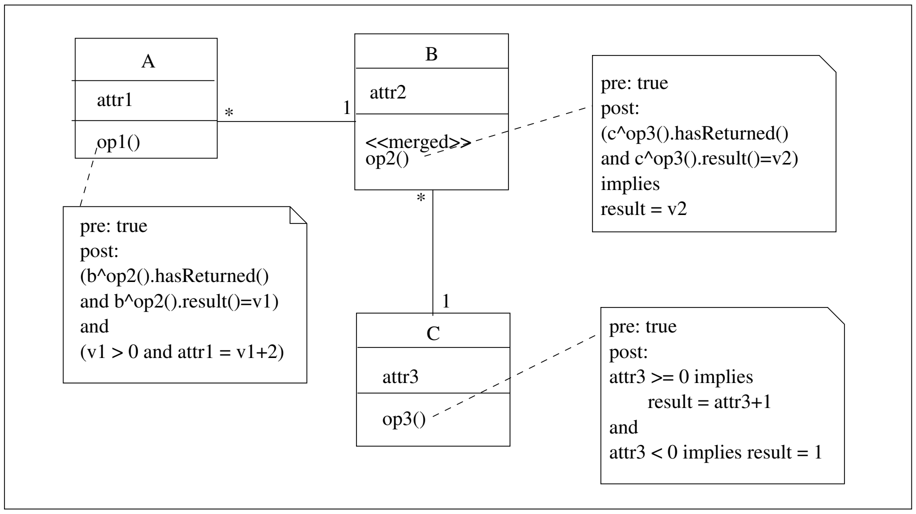


Figure 6.3: A class diagram obtained by merging the two class diagrams in Fig. 6.2(a) and Fig. 6.2(b)

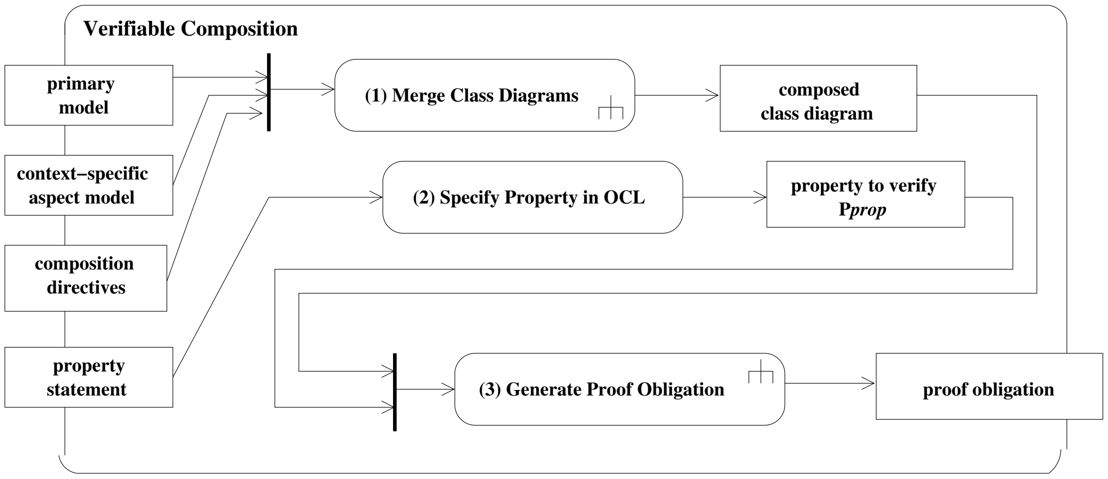
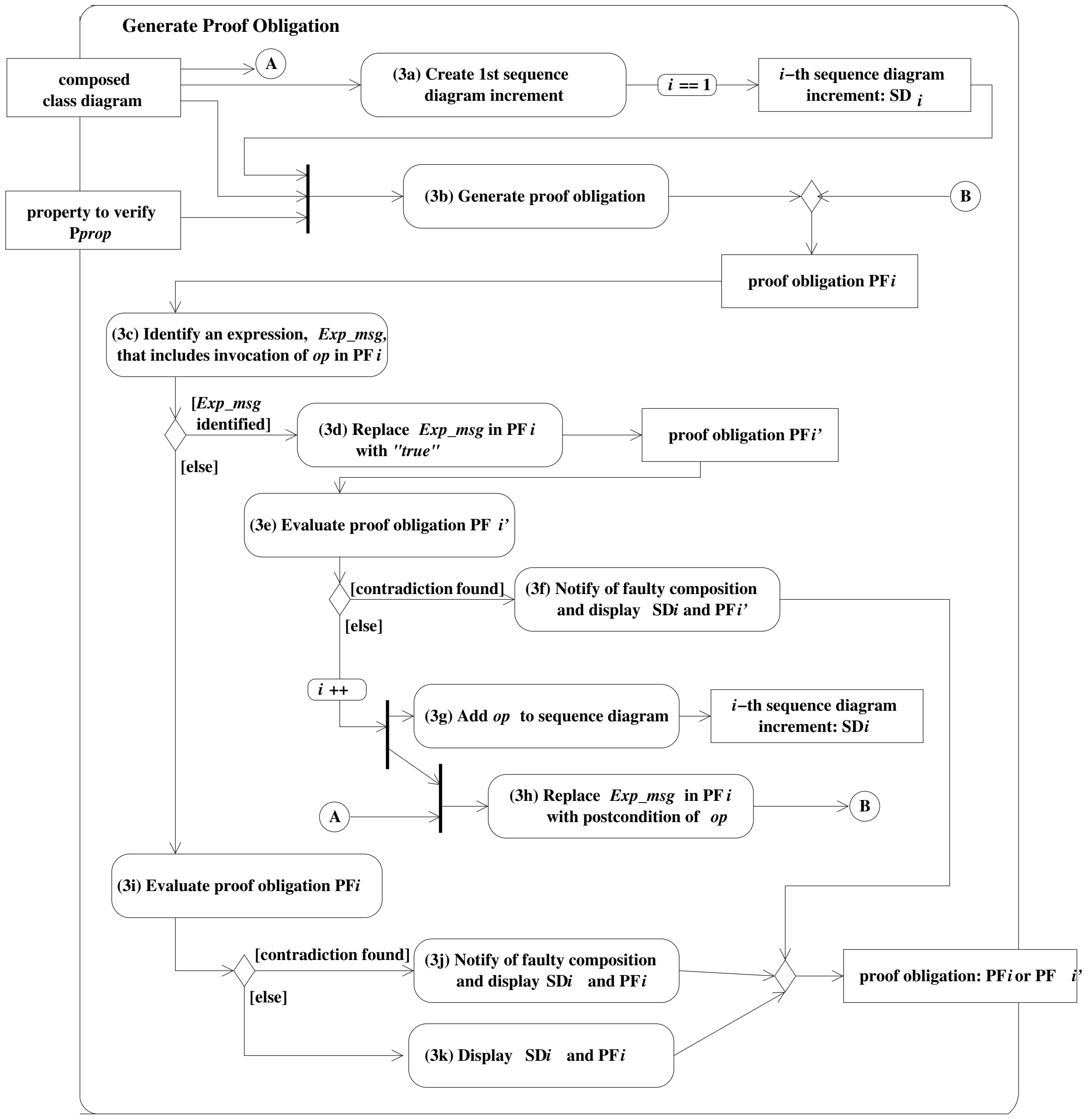


Figure 6.4: An activity diagram showing the verifiable composition approach

this property when a primary model and context-specific aspect model are composed. In our approach, all message invocations in the composed model are explicitly specified when the operation specifications of the class diagrams are composed under effects of relevant composition directives. Therefore a composed sequence diagram is derived from those operation specifications in the composed class diagram as illustrated in Fig. 6.4(refer to

(3)). While a sequence diagram is incrementally derived from the operation specifications, a proof obligation is generated, evolves, and is evaluated. If any faulty composition is notified during the evaluation, the current sequence diagram, which is partially derived at that point, and the current proof obligation may be used to determine at which part of the composition the property fails to hold. Otherwise, a sequence diagram is obtained.

Fig. 6.5 illustrates the detailed steps showing how proof obligation can be generated in our approach. The first sequence diagram increment is created, which has a triggering operation event with a lifeline of an object of the context class in which the property to be verified P_{prop} is specified (action (3a) in Fig. 6.5). By tracing this partial sequence diagram back to the relevant operation in the composed class diagram, the proof obligation, PF_i where i is 1, is obtained (action (3b)). As defined earlier, the implication P_{beh} implies P_{prop} forms the proof obligation. Discharging a proof obligation, PF_1, requires the information under which condition PF_1 holds. When an expression in the proof obligation includes any message invocations, discharging requires to know the condition under which its sub-messages hold, which may require to increment the next message events to the sequence diagram. Therefore, we propose to identify a part of expression (denoted as $Expr_{msg}$) that includes invoking sub-messages (action (3c)) and verify other parts of expression first by assuming $Expr_{msg}$ returns true (action (3d) in Fig. 6.5). The proof obligation obtained from this step is referred as PF_i' in Fig. 6.5. PF_i' is evaluated (action (3e) in Fig. 6.5). Any contradiction found from evaluating PF_i' uncovers a faulty composition and the current sequence diagram and proof obligation can be used to guide which point of the composition results in the failure (action (3f) in Fig. 6.5). On the other hand, if no contradiction is found from evaluating PF_i', now one needs to increment the current sequence diagram (action (3g) in Fig. 6.5). Selecting a further message invocation, op, is derived from the postcondition of operation in the composed class diagram. $Expr_{msg}$ of the PF_i is replaced by the postcondition of the operation to be invoked (action (3h) in Fig. 6.5). At this step, a proof obligation PF_i evolves to PF_{i+1} and the existence of $Expr_{msg}$ is checked again (action



Generate Proof Obligation
composed class diagram
A
(3a) Create 1st sequence diagram increment
i == 1
i–th sequence diagram increment: SD i
property to verify Pprop
(3b) Generate proof obligation
B
proof obligation PFi
(3c) Identify an expression, Exp_msg, that includes invocation of op in PF i
[Exp_msg identified]
(3d) Replace Exp_msg in PF i with "true"
proof obligation PFi'
[else]
(3e) Evaluate proof obligation PF i'
[contradiction found]
(3f) Notify of faulty composition and display SDi and PFi'
[else]
i ++
(3g) Add op to sequence diagram
i–th sequence diagram increment: SDi
A
(3h) Replace Exp_msg in PF i with postcondition of op
B
(3i) Evaluate proof obligation PFi
[contradiction found]
(3j) Notify of faulty composition and display SDi and PFi
[else]
proof obligation: PFi or PF i'
(3k) Display SDi and PFi

Figure 6.5: An activity diagram showing the process of generating proof obligation from the merged class diagram

(3c)). When no further message invocation ($Expr_{msg}$) is included in the proof obligation, the current proof obligation is evaluated (action (3i)). If any contradiction is found from the evaluation, the faulty composition is notified with information about current status of composition, that is, the current sequence diagram and proof obligation expression (action (3j)).

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

We demonstrate the steps that one needs to follow for the use of our verifiable composition approach to compose the two models shown in Fig. 6.2(a) and Fig. 6.2(b).

(1) Merge class diagrams. The two class diagrams shown in Fig. 6.2(a) and Fig. 6.2(b) are merged to create the model shown in Fig. 6.3.

- The class B in the primary model and the class B in the aspect model are matched when we use model element names to identify the elements that are to be merged.
  - The two attributes with the name of $attr2$ are merged to form a single attribute in the merged class model (see Fig. 6.3).
  - The operation $op2$ in the class B of the primary model is merged with the operation $op2$ in class B of the context-specific model using the composition directive *replacePostSpec*. The operation $op2$ in the composed model thus has the postcondition that came from the $op2$ in the aspect model. The stereotype <<merged>> is used to denote that the operation behavior has been changed by the merge.
- All other model elements (i.e., the class A with its attribute $attr1$ and operation $op1$, another class C with its attribute $attr3$ and operation $op3$, and two associations) are included in the composed class diagram with no modifications.

(2) Specify the property to be verified, P_{Prop}, using OCL. In this example, the property, P_{Prop}, is the condition that must be established by an operation $op1$ and is stated as follows:

If the operation op1 is completed, then $attr1$ in the object a of A must be greater than $attr3$ in the object c of C.

The above statement can be specified, using the OCL, as follows:

```
context A
    inv: attr1 > self.B.C.attr3
```

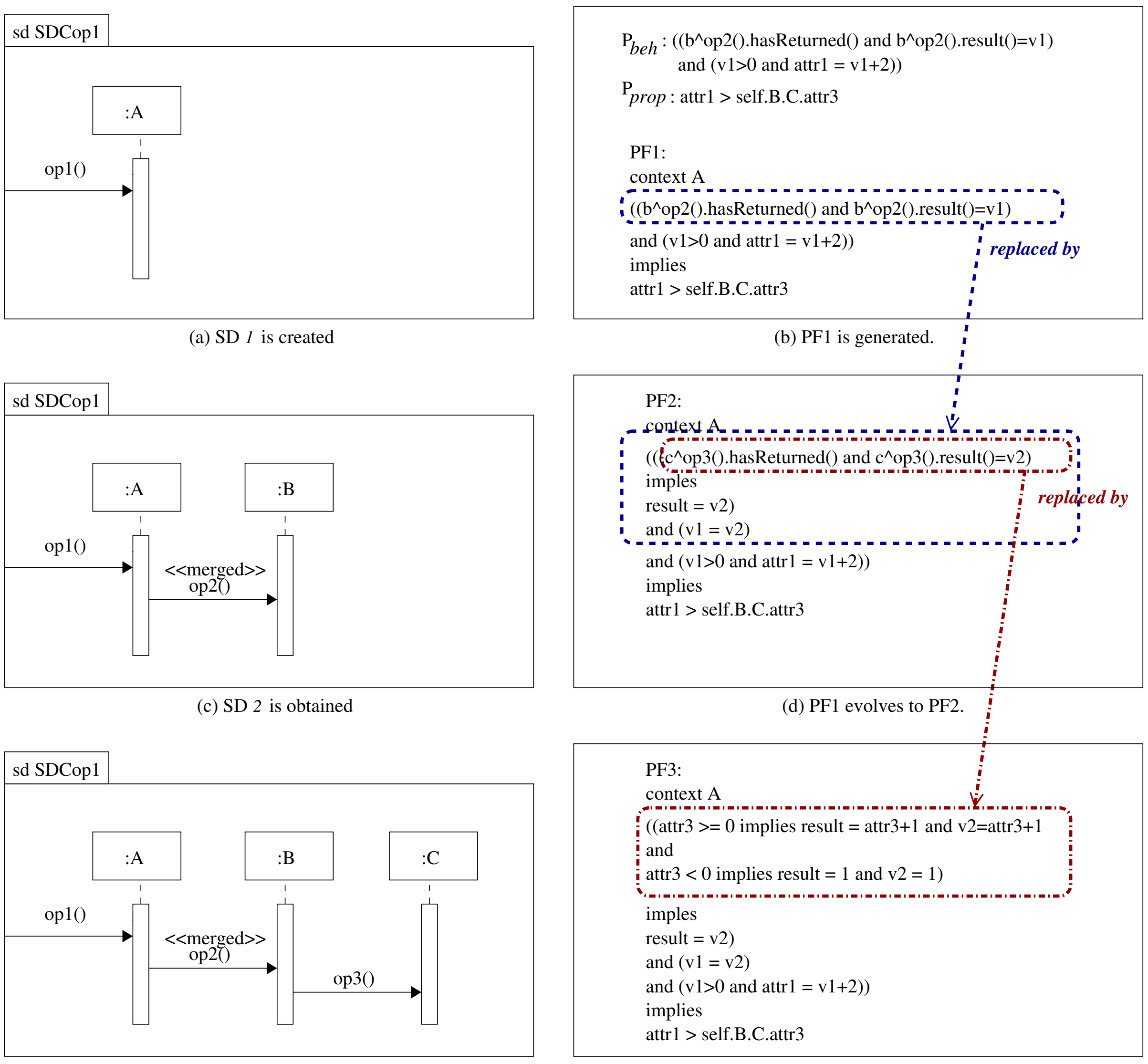
In the above, the property to be verified is specified in the context of the $op1$ operation in the class A.

(3) Generating the proof obligation. The proof obligation is generated from the operation specification P_{beh} and the property to verify P_{prop} and a partial sequence diagram will be obtained while the proof obligation is generated.

(3a) Create the first sequence diagram increment *SD*1. In this example, the composition starts with an event of operation $op1$ in an object of the class A and further operation events are identified as the composition follows. Therefore, a partial sequence diagram that shows only one operation $op1$ sent to a lifeline of an object of A is created (refer to Fig. 6.6(a)).

(3b) Generate a proof obligation *PF1*. The operation behavior specification P_{beh} for the composed sequence diagram *SDCOp1* is obtained from the operation specification of $op1$ in Fig. 6.3 and is given below:

```
P_beh
    = P_pre@pre and P_post
    = true and
      (b^op2().hasReturned() and b^op2().result() = v1)
      and
      (v1 > 0 and attr1 = v1 + 2)
    = (b^op2().hasReturned() and b^op2().result() = v1)
      and
      (v1 > 0 and attr1 = v1 + 2)
```



```
Pbeh : ((b^op2().hasReturned() and b^op2().result()=v1)
        and (v1>0 and attr1 = v1+2))
Pprop : attr1 > self.B.C.attr3

PF1:
context A
((b^op2().hasReturned() and b^op2().result()=v1)
and (v1>0 and attr1 = v1+2))
implies
attr1 > self.B.C.attr3
```

```
PF2:
context A
(((c^op3().hasReturned() and c^op3().result()=v2)
implies
result = v2)
and (v1 = v2)
and (v1>0 and attr1 = v1+2))
implies
attr1 > self.B.C.attr3
```

```
PF3:
context A
((attr3 >= 0 implies result = attr3+1 and v2=attr3+1
and
attr3 < 0 implies result = 1 and v2 = 1)
implies
result = v2)
and (v1 = v2)
and (v1>0 and attr1 = v1+2))
implies
attr1 > self.B.C.attr3
```

(a) SD *1* is created

(b) PF1 is generated.

(c) SD *2* is obtained

(d) PF1 evolves to PF2.

(e) SD *3* is obtained

(f) PF2 evolves to PF3.

Figure 6.6: Generating the proof obligation while identifying which message invocation will follow in a sequence diagram

Therefore, the proof obligation *PF1* is generated from P_{beh} and P_{prop} at the time of composition shown in Fig. 6.6(a) and is shown in Fig. 6.6(b).

(3c) Identify an expression, $Expr_{msg}$, that includes the invocation of op in $PF1$.

Identified part of expression is denoted by $Expr_{msg}$ as shown below.

```
Proof Obligation 1: PF1

context A
    ((b^op2().hasReturned() and b^op2().result() = v1)    -- (P1−1): Expr_msg
    and
    (v1 > 0 and attr1 = v1 + 2))                          -- (P1−2): Expr_non−msg
    implies
    attr1 > self.B.C.attr3                                -- (P_Prop)
```

(3d) Replace *Expr*$_{msg}$ with "*true*". Discharging a proof obligation requires the information about the conditions under which the condition labeled in either P$_{1-1}$ or P$_{1-2}$ in *PF1* holds. However, one does not have the information on invoking operation *op*2 until the operation event is actually added to the sequence diagram. Therefore, we assume P$_{1-1}$ is true for now in order to verify other parts of the expression. The following proof obligation *PF1'* is obtained at this step (not shown in Fig. 6.6):

```
Proof Obligation 1': PF1'

context A
    (true -- P1−1
    and
    (v1 > 0 and attr1 = v1 + 2)) -- (P1−2)
    implies
    attr1 > self.B.C.attr3 -- (P_Prop)
```

and it is reduced into the following:

```
Proof Obligation 1': PF1'

context A
    (v1 > 0 and attr1 = v1 + 2) -- (P1−2)
    implies
    attr1 > self.B.C.attr3 -- (P_Prop)
```

(3e) Evaluate a proof obligation PF1'. In this step, we have the expression with respect to v1, attr1, and attr3. If any contradiction is found, discharge fails and the composition must stop (refer to action (3f)). In this example, no contradiction is found based on the given information .Therefore, now we go back to our first assumption that P_{1-1} is true and it requires to add the next increment to the current sequence diagram (refer to the increment of an index i in Fig. 6.6.

(3g) Add the next message event $op2$ to the sequence diagram. To evaluate P_{1-1}, one needs to know about the conditions under which the $op2$ operation sent to an object b returns true in the composed model. Therefore the message event *op2* is added to the current sequence diagram (refer to Fig. 6.6(c)).

(3h) Replace *Expr*$_{msg}$ in *PF1* by the post condition of the $op2$ operation. This observation leads that the condition labeled in P_{1-1} in *PF1* is replaced by the the postcondition that determines when the $op2$ operation in b object of the composed model returns true. Note that *v1 = v2* is added into the replaced part as well to ensure that those two return values, $v1$ and $v2$, are same.

The postcondition of $op2$ operation of b in the composed model (see Fig. 6.3) is given below:

```
context B::op2()
    (c^op3().hasReturned() and c^op3().result() = v2)
    implies result = v2
```

Therefore, the resulting proof obligation *PF2* is as follows (also shown in Fig. 6.6(d)):

```
Proof Obligation 2: PF2

context A
    (((c^op3().hasReturned() and c^op3().result() = v2) −− (P2−1)
    implies
    result = v2 ) −− (P2−2)
    and
    v1 = v2 −− (P2−3)
    and
    (v1 > 0 and attr1 = v1 + 2) −−) (P1−2)
    implies
    attr1 > self.B.C.attr3 −− (PProp)
```

The current proof obligation $PF2$ still includes an expression that requires the result of invoked sub-operation $op3$ (refer to P$_{2-1}$).

(3d) Replace *Expr*$_{msg}$ with "*true*". Assuming that " *(P$_{2-1}$)* implies *(P$_{2-2}$)*" is true for now gives us the following:

```
Proof Obligation 1': PF2'

context A
    (v1 = v2 −− (P2−3)
    and
    (v1 > 0 and attr1 = v1 + 2)) −− (P1−2)
    implies
    attr1 > self.B.C.attr3 −− (PProp)
```

By v1 = v2 given in (P$_{2-3}$, all occurrences of v1 in *(P$_{1-2}$)* are replaced with v2 and the changed *(P$_{1-2}$)* is now labeled with *(P$_{2-4}$)*.

```
context A
    (v2 > 0 and attr1 = v2 + 2) −− (P2−4)
    implies
    attr1 > self.B.C.attr3 −− (PProp)
```

(3e) Evaluate a proof obligation PF2'. Again, no contradiction is found based on the given information. Therefore, discharging proof obligation PF2 requires determining the conditions under which the implication of *(P_{2-1})* and *(P_{2-2})* is true. The next message event needs to be added to the current sequence diagram (refer to the increment of an index *i* in Fig. 6.6.

(3g) Add the next message event $op3$ to the sequence diagram. The message event *op3* is added to the current sequence diagram (refer to Fig. 6.6(e)).

(3h) Replace *Expr_{msg}* in *PF2* by the post condition of the $op3$ operation. This step requires us to evaluate whether the implication with *(P_{2-1})* and *(P_{2-2})* holds or not. Again, we must know about the conditions under which the $op3$ operation in the *c* object returns true. This observation leads to a modified proof obligation in which the condition labeled in *(P_{2-1})* is replaced by the postcondition that determines when the $op3$ operation in *c* object returns true. That is, *(P_{2-1})* is replaced by the postcondition of $op3$ operation in *c*.

The postcondition of $op3$ operation in *c* is repeated below:

```
context C::op3()
    attr3 ≥ 0 implies result = attr3 + 1
    and
    attr3 < 0 implies result = 1
```

Therefore, the resulting proof obligation *PF3* (also shown in Fig. 6.6(f)) is as follows:

Proof Obligation 3: PF3

context A
((attr3 $\geq$ 0 implies result = attr3 + 1 and v2 = attr3 + 1 $--$ *(P_{3-1})*
and
attr3 $<$ 0 implies result = 1 and v2 = 1) $--$ *(P_{3-2})*
and
result = v2) $--$ *(P_{2-2})*
and
v1 = v2 $--$ *(P_{2-3})*
and
(v1 $>$ 0 and attr1 = v1 + 2) $--$ *(P_{1-2})*
implies
attr1 $>$ self.B.C.attr3 $--$ *(Prop)*

Note that v2 = attr3 + 1 and v2 = 1 are added to *(P_{3-1})* and *(P_{3-2})* respectively as done in the previous step.

At this point, no further message invocation is included in the current proof obligation *PF*3 shown above. Therefore it can be evaluated. Any failure to discharge it may reveal the problem in the composition.

(3i) Evaluate a proof obligation PF3. In this example, no contradiction is found based on the given information.

(3k) Display the composed sequence diagram *SDi* and the generated proof obligation.

6.3 A Pilot Study: Composing an RBAC Aspect with a Banking Application

In this section, we present a context-specific RBAC aspect model for banking applications and outline how the RBAC aspect model is composed with a banking application primary model using our verifiable composition technique described in previous sections.

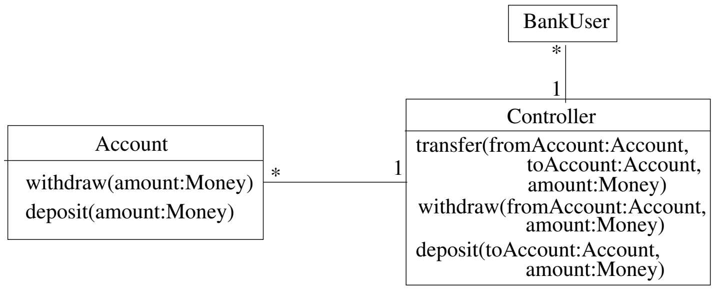

BankUser
*
1
Controller
transfer(fromAccount:Account, toAccount:Account, amount:Money)
withdraw(fromAccount:Account, amount:Money)
deposit(toAccount:Account, amount:Money)
Account
withdraw(amount:Money)
deposit(amount:Money)
*
1

Figure 6.7: A partial class diagram for a banking application (a primary model)

6.3.1 A Banking Application Primary Model

Fig. 6.7 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).

The dynamic view of the primary model consists of a set of interaction diagrams. In this section we show the sequence diagram and OCL specification for the $transfer$ operation only. As shown in Fig. 6.8, a $transfer$ operation results in the invocation of two other operations ($withdraw$ and $deposit$). Operation specifications for $transfer$, $withdraw$ and $deposit$ operations are given below:

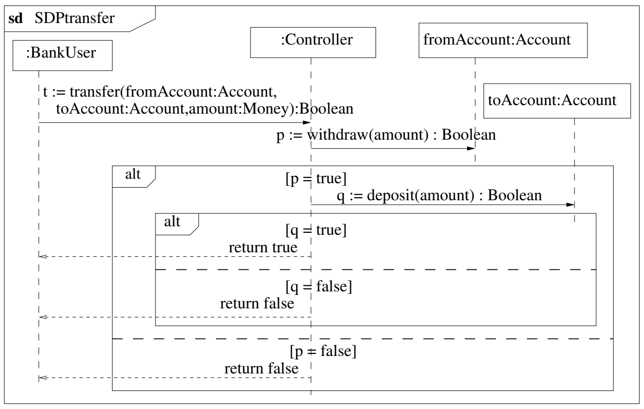


Figure 6.8: Sequence diagram for the *transfer* operation in a banking application (a primary model)

P-SPEC-1

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
result =
(fromAccountˆwithdraw(amount).hasReturned() and
fromAccountˆwithdraw(amount).result() = true) and
(toAccountˆdeposit(amount).hasReturned() and
toAccountˆdeposit(amount).result() = true)

P-SPEC-2

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

P-SPEC-3

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

6.3.2 The RBAC Aspect Model

As described in the previous chapters, 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 chapter we focus only on constraints in the *hierarchical SSD* RBAC aspect model that was described in Section 4.2. The class diagram template of the hierarchical SSD RBAC is repeated in Fig. 6.9 for the readability. Annotated operation specification templates for *Operation*

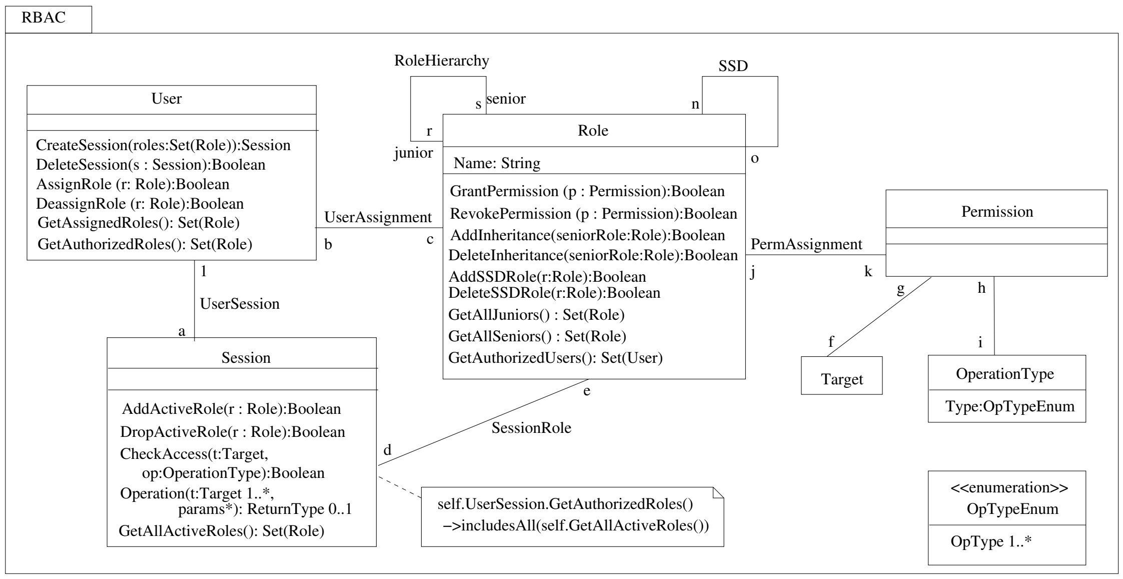

RBAC
RoleHierarchy
SSD
User
CreateSession(roles:Set(Role)):Session
DeleteSession(s : Session):Boolean
AssignRole (r: Role):Boolean
DeassignRole (r: Role):Boolean
GetAssignedRoles(): Set(Role)
GetAuthorizedRoles(): Set(Role)
s senior
r
junior
n
o
Role
Name: String
GrantPermission (p : Permission):Boolean
RevokePermission (p : Permission):Boolean
AddInheritance(seniorRole:Role):Boolean
DeleteInheritance(seniorRole:Role):Boolean
AddSSDRole(r:Role):Boolean
DeleteSSDRole(r:Role):Boolean
GetAllJuniors() : Set(Role)
GetAllSeniors() : Set(Role)
GetAuthorizedUsers(): Set(User)
UserAssignment
b
c
PermAssignment
j
k
Permission
g
h
f
i
Target
OperationType
Type:OpTypeEnum
l
UserSession
a
Session
AddActiveRole(r : Role):Boolean
DropActiveRole(r : Role):Boolean
CheckAccess(t:Target, op:OperationType):Boolean
Operation(t:Target 1..*, params*): ReturnType 0..1
GetAllActiveRoles(): Set(Role)
e
SessionRole
d
self.UserSession.GetAuthorizedRoles()
->includesAll(self.GetAllActiveRoles())
<<enumeration>>
OpTypeEnum
OpType 1..*

Figure 6.9: The class model template view of the hierarchical SSD RBAC aspect model

and *CheckAccess* in the *Session* template are given below. As shown in the operation specification templates for *Operation*, 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.
  −− Start of constraint in postcondition:
    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.
      (self ˆ CheckAccess(tar-i: Target, op: OperationType).hasReturned() and
       self ˆ CheckAccess(tar-i: Target, op: OperationType).result() = true) and }
      −− 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
      (? ˆ DoOperation(tar:Target *, params *).hasReturned() and
       ? ˆ DoOperation(tar:Target *, params *).result() = true)
      −− represents the sending of the DoOperation message to an
      −− unknown object (the object is provided when the template is instantiated
      −− and an instantiated template is incorporated into a primary model).
  −− End of Operation specification
```

```
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().Permission
      →exists(p|p.Target → includes(tar) and p.OperationType → includes(op))
  −− End of CheckAccess specification
```

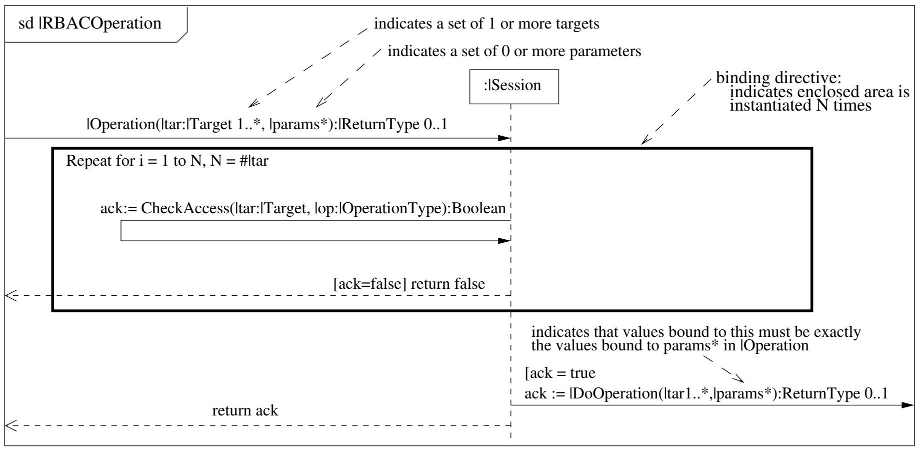

sd |RBACOperation
indicates a set of 1 or more targets
indicates a set of 0 or more parameters
:|Session
binding directive: indicates enclosed area is instantiated N times
|Operation(|tar:|Target 1..*, |params*):|ReturnType 0..1
Repeat for i = 1 to N, N = #|tar
ack:= CheckAccess(|tar:|Target, |op:|OperationType):Boolean
[ack=false] return false
indicates that values bound to this must be exactly the values bound to params* in |Operation
[ack = true
ack := |DoOperation(|tar1..*,|params*):ReturnType 0..1
return ack

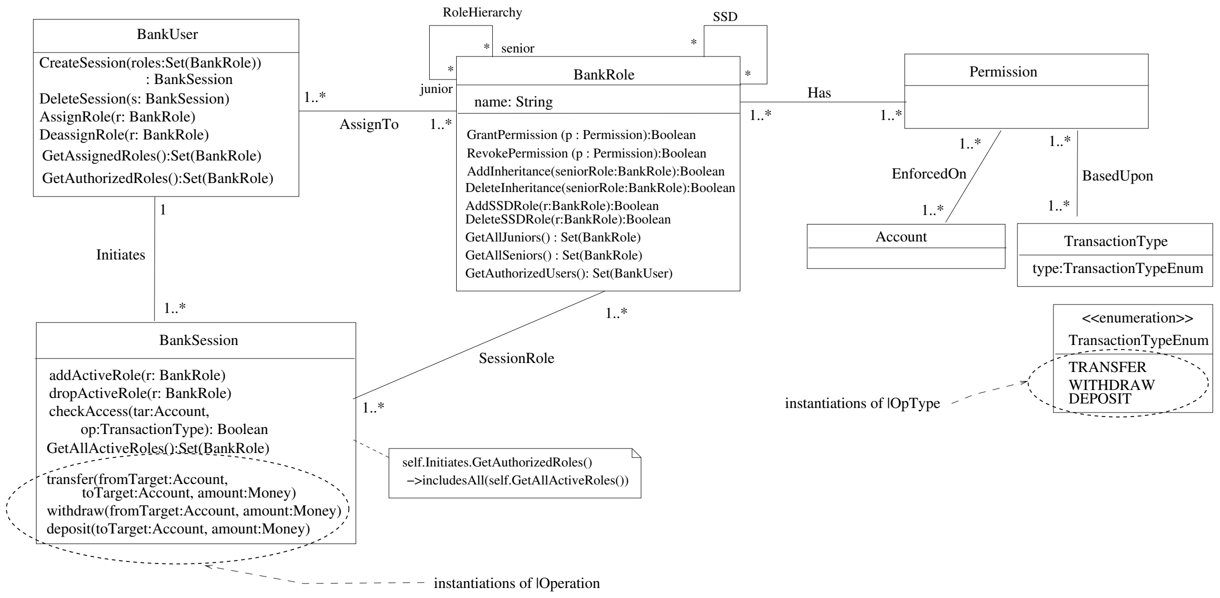
Figure 6.10: *SDAOperation* sequence diagram template

The sequence diagram template *SDAOperation* shown in Fig. 6.10 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. 6.10 (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.



BankUser
CreateSession(roles:Set(BankRole)) : BankSession
DeleteSession(s: BankSession)
AssignRole(r: BankRole)
DeassignRole(r: BankRole)
GetAssignedRoles():Set(BankRole)
GetAuthorizedRoles():Set(BankRole)
RoleHierarchy
senior
junior
SSD
BankRole
name: String
GrantPermission (p : Permission):Boolean
RevokePermission (p : Permission):Boolean
AddInheritance(seniorRole:BankRole):Boolean
DeleteInheritance(seniorRole:BankRole):Boolean
AddSSDRole(r:BankRole):Boolean
DeleteSSDRole(r:BankRole):Boolean
GetAllJuniors() : Set(BankRole)
GetAllSeniors() : Set(BankRole)
GetAuthorizedUsers(): Set(BankUser)
AssignTo
Has
Permission
EnforcedOn
BasedUpon
Account
TransactionType
type:TransactionTypeEnum
<<enumeration>>
TransactionTypeEnum
TRANSFER
WITHDRAW
DEPOSIT
instantiations of |OpType
Initiates
SessionRole
BankSession
addActiveRole(r: BankRole)
dropActiveRole(r: BankRole)
checkAccess(tar:Account, op:TransactionType): Boolean
GetAllActiveRoles():Set(BankRole)
transfer(fromTarget:Account, toTarget:Account, amount:Money)
withdraw(fromTarget:Account, amount:Money)
deposit(toTarget:Account, amount:Money)
self.Initiates.GetAuthorizedRoles() ->includesAll(self.GetAllActiveRoles())
instantiations of |Operation

Figure 6.11: A context-specific core RBAC class diagram

6.3.3 Instantiating the RBAC Aspect Model for a Banking Application

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. 6.11. 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 *Account* 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*.

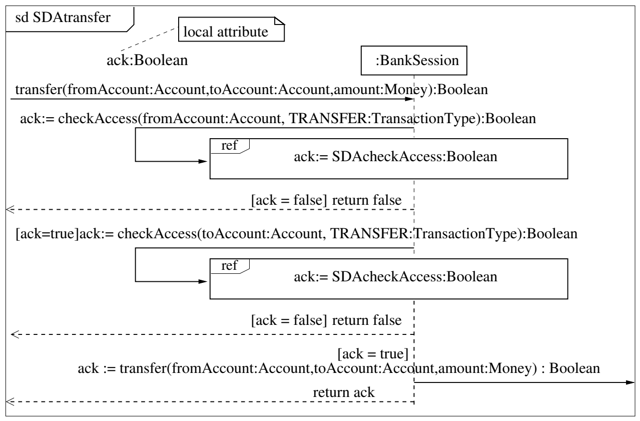
The enumeration values in *TransactionTypeEnum* (*TRANSFER*, *WITHDRAW*, and *DEPOSIT*) are instantiations of an attribute template *OpType*. The operation specification template associated with the *Operation* template is also instantiated for each of these operations. For example, the *transfer* operation in *BankSession* class of the aspect model is associated with the following instantiation of the *Operation* specification template:

```
context BankSession::transfer(fromAccount:Account,
      toAccount:Account,amount:Money):Boolean
pre: true
post:
result =
−− Statement in Repeat block of template is instantiated
−− twice because there are two targets
−− in the argument: fromAccount and toAccount.
  (self^checkAccess(fromAccount, TRANSFER).hasReturned()
  and
  self^checkAccess(fromAccount, TRANSFER).result()=true)
  and
  (self^checkAccess(toAccount, TRANSFER).hasReturned()
  and
  self^checkAccess(toAccount, TRANSFER).result()=true)
  and
  (?^transfer(fromAccount, ...).hasReturned()
  and
  ?^transfer(fromAccount, ...).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.GetAllActiveRoles().Permission
    →exists(p|p.Account→includes(tar)
    and p.TransactionType→includes(op))
```

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* (see Fig. 6.12, Fig. 6.13, and Fig. 6.14).

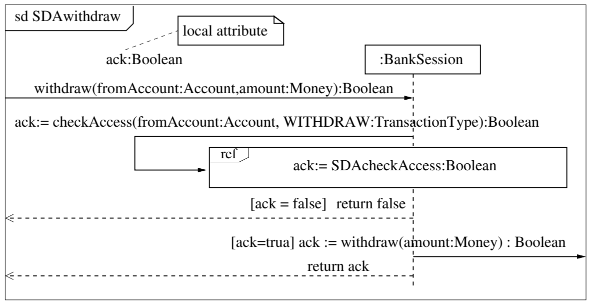


sd SDAtransfer
local attribute
ack:Boolean
:BankSession
transfer(fromAccount:Account,toAccount:Account,amount:Money):Boolean
ack:= checkAccess(fromAccount:Account, TRANSFER:TransactionType):Boolean
ref
ack:= SDAcheckAccess:Boolean
[ack = false] return false
[ack=true]ack:= checkAccess(toAccount:Account, 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 6.12: SDAtransfer: Context-specific sequence diagram for the *transfer* operation in *BankSession*

6.3.4 Merging a Context-specific RBAC Aspect Model with a Primary Model

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 (For a detailed description of class diagram composition, refer to more complex examples given



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

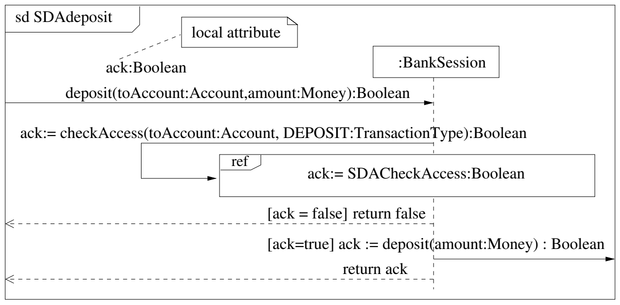

sd SDAdeposit
local attribute
ack:Boolean
:BankSession
deposit(toAccount:Account,amount:Money):Boolean
ack:= checkAccess(toAccount:Account, DEPOSIT:TransactionType):Boolean
ref
ack:= SDACheckAccess:Boolean
[ack = false] return false
[ack=true] ack := deposit(amount:Money) : Boolean
return ack

Figure 6.14: SDAdeposit: Context-specific sequence diagram for the *deposit* operation in *BankSession*

in France *et al.* [31] and Reddy *et al.* [73]. The result of merging the aspect model's class diagram and the primary model's class diagram is shown in Fig. 6.15. [31].

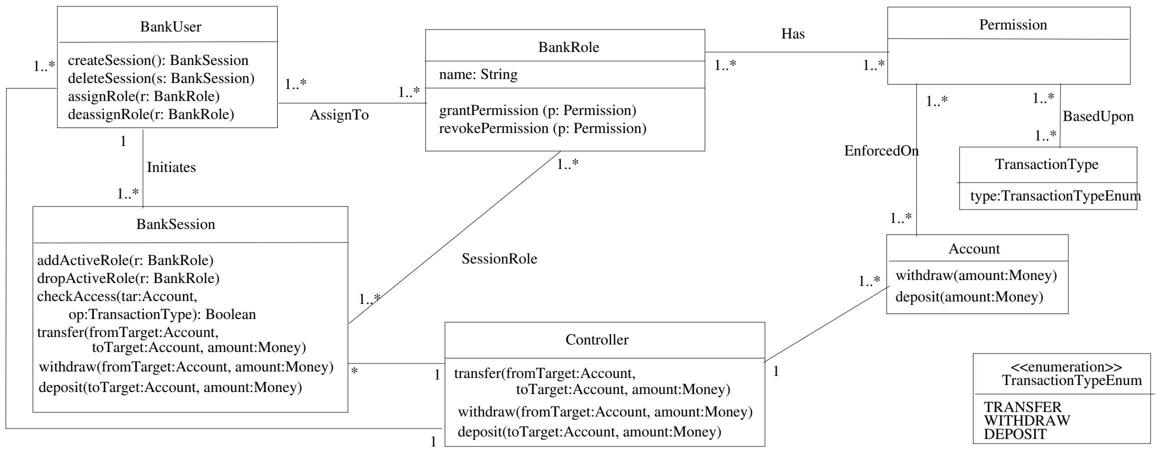

BankUser
createSession(): BankSession
deleteSession(s: BankSession)
assignRole(r: BankRole)
deassignRole(r: BankRole)
BankRole
name: String
grantPermission (p: Permission)
revokePermission (p: Permission)
Permission
Has
AssignTo
Initiates
SessionRole
EnforcedOn
BasedUpon
TransactionType
type:TransactionTypeEnum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

Figure 6.15: Class diagram of the composed model

Incorporating the access control behavior into the banking application requires that the *transfer* operation specification in the primary model's *Controller* class (see *P-SPEC-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 a composition directive, *replacePostSpec*, that replaces 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:
  result =
    (bankSession^withdraw(fromAccount,amount).hasReturned() and
    bankSession^withdraw(fromAccount,amount).result()=true and
    bankSession^deposit(toAccount,amount).hasReturned() and
    bankSession^deposit(toAccount,amount).result()=true)
```

The unknown object (represented by "?") shown in the $transfer$ operation in *BankSession* class of the composed model is now associated with the $Controller$ and the following shows its completed operation specification:

```
  context BankSession::transfer(fromAccount:Account,
        toAccount:Account,amount:Money):Boolean
  pre: true
  post:
  result =
    (self^checkAccess(fromAccount, TRANSFER).hasReturned()
    and
    self^checkAccess(fromAccount, TRANSFER).result()=true)
    and
    (self^checkAccess(toAccount, TRANSFER).hasReturned()
    and
    self^checkAccess(toAccount, TRANSFER).result()=true)
    and
    (Controller^transfer(fromAccount, toAccount, amount).hasReturned()
    and
    Controller^transfer(fromAccount, toAccount, amount).result() = true)
```

6.3.5 Specifying the Property to Verify

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. 6.8) is composed with the *SDAtransfer*, *SDAwithdraw*, *SDAdeposit* sequence diagrams in the context-specific aspect model (Fig. 6.12, Fig. 6.13, Fig. 6.14). 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 property to verify 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
  verify TransferProp:
  let
    successful-transfer =
      (if fromAccount.balance@pre >= amount
      then (fromAccount.balance =
          fromAccount.balance@pre-amount
        and toAccount.balance =
          toAccount.balance@pre + amount)))
  in
    if (self^checkAccess(fromAccount, TRANSFER).hasReturned()
      and self^checkAccess(fromAccount, TRANSFER).result()=true
      and self^checkAccess(toAccount, TRANSFER).hasReturned()
      and self^checkAccess(toAccount, TRANSFER).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*.

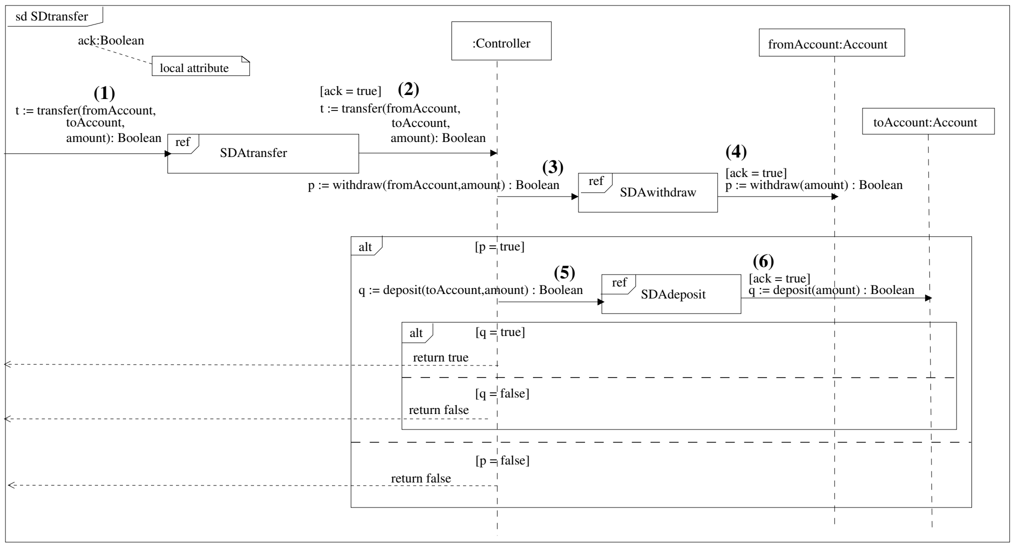


Figure 6.16: Overview of generating and evaluating the proof obligation

6.3.6 Generating the Proof Obligation

We use Fig. 6.16 to illustrate how a proof obligation evolves while the sequence diagram describing the *transfer* operation in the composed model is derived from the class diagram composition. 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 performs the following steps (the numbers shown in Fig. 6.16 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 in-

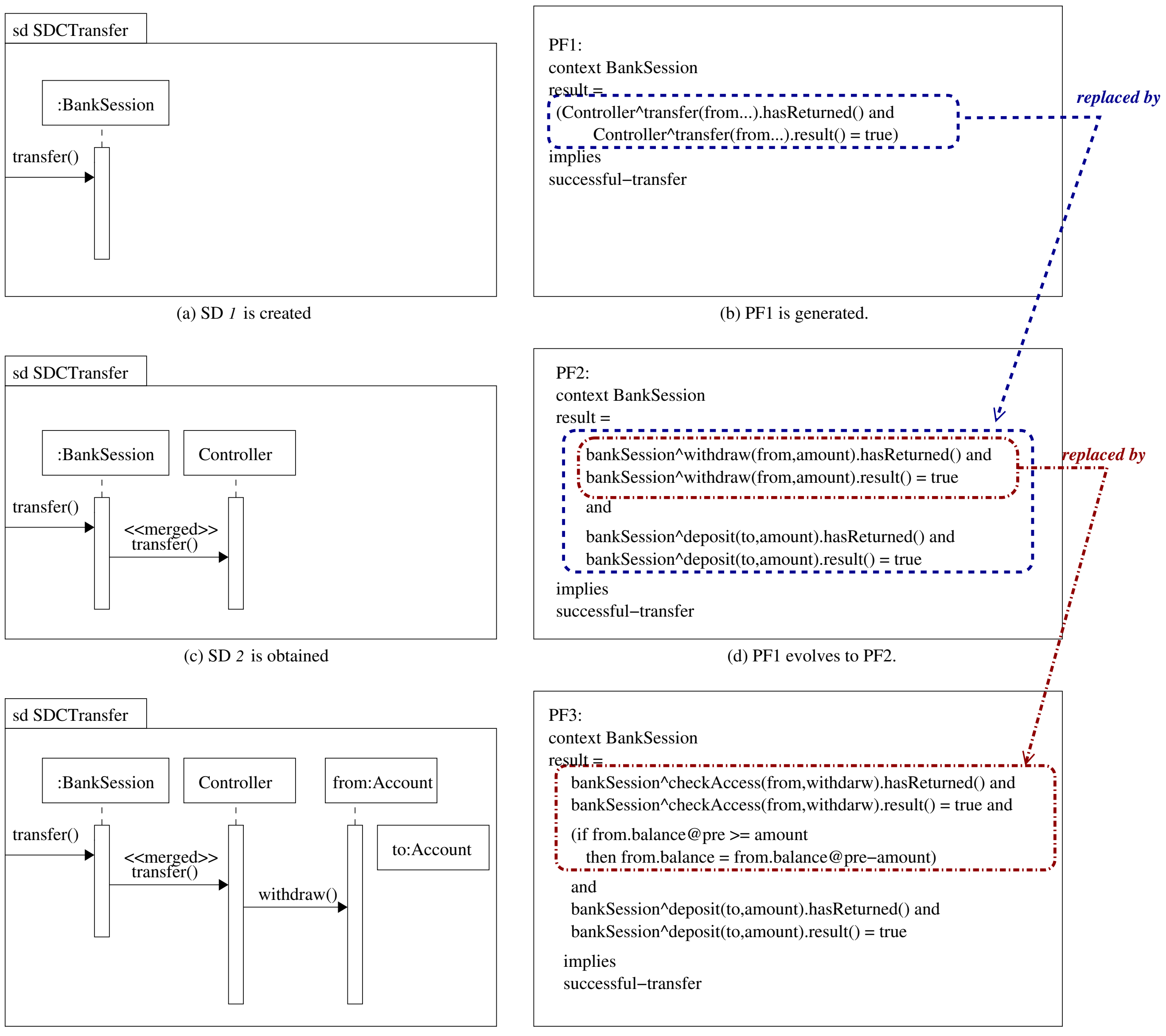
tercepted by *SDAtransfer* and passed on to the *Controller* object only if access is granted.
(3) The call to the *withdraw* operation made by the *Controller* during the invocation of the *transfer* operation is intercepted by the *SDAwithdraw* sequence diagram.
(4) If access is granted then a *withdraw* 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*.
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.

The proof obligation and a sequence diagram is obtained as illustrated in Fig. 6.17. In steps (1) and (2) of the sequence diagram composition described earlier with Fig. 6.16, the *SDAtransfer* sequence diagram is incorporated into the primary model's *transfer* sequence diagram. The initial sequence diagram is derived from the current composition (refer to Fig. 6.17(a)). At this point, the proof obligation can be expressed as an implication *P1 implies TransferProp*, where *P*1 specifies the condition under which the *transfer* operation in the *BankSession* object returns true. The postcondition for *transfer* is repeated below:

Proof Obligation 1: PF1

```
context BankSession
result =
  ((self^checkAccess(fromAccount, TRANSFER).hasReturned() and
    self^checkAccess(fromAccount, TRANSFER).result()=true)
    and (self^checkAccess(toAccount, TRANSFER).hasReturned() and
    self^checkAccess(toAccount, TRANSFER).result()=true)
    and (Controller^transfer(fromAccount, ...).hasReturned() and
    Controller^transfer(fromAccount, ...).result() = true))
```

The following proof obligation (obtained after simplification) is generated as shown



sd SDCTransfer
:BankSession
transfer()
(a) SD 1 is created
PF1:
context BankSession
result =
(Controller^transfer(from...).hasReturned() and
Controller^transfer(from...).result() = true)
implies
successful−transfer
(b) PF1 is generated.
replaced by
sd SDCTransfer
:BankSession
Controller
transfer()
<<merged>>
transfer()
(c) SD 2 is obtained
PF2:
context BankSession
result =
bankSession^withdraw(from,amount).hasReturned() and
bankSession^withdraw(from,amount).result() = true
and
bankSession^deposit(to,amount).hasReturned() and
bankSession^deposit(to,amount).result() = true
implies
successful−transfer
(d) PF1 evolves to PF2.
replaced by
sd SDCTransfer
:BankSession
Controller
from:Account
to:Account
transfer()
<<merged>>
transfer()
withdraw()
(e) SD 3 is obtained
PF3:
context BankSession
result =
bankSession^checkAccess(from,withdarw).hasReturned() and
bankSession^checkAccess(from,withdarw).result() = true and
(if from.balance@pre >= amount
then from.balance = from.balance@pre−amount)
and
bankSession^deposit(to,amount).hasReturned() and
bankSession^deposit(to,amount).result() = true
implies
successful−transfer
(f) PF2 evolves to PF3.

Figure 6.17: Generating the proof obligation while identifying which message invocation will follow in a sequence diagram

in Fig. 6.17(b):

```
Proof Obligation 1: PF1

context BankSession
result =
  (Controller^transfer(fromAccount, ...).hasReturned() and
    Controller^transfer(fromAccount, ...).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 leads to adding the invocation of the *transfer* operation in the *Controller* object to the sequence diagram (refer to Fig. 6.17(c)) and modifying the 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 (refer to Fig. 6.17(d)). The resulting proof obligation is given below:

```
Proof Obligation 2: PF2

context BankSession
  (bankSession^withdraw(fromAccount,amount).hasReturned() and
    bankSession^withdraw(fromAccount,amount).result()=true and −− (WD)
    bankSession^deposit(toAccount,amount).hasReturned() and
    bankSession^deposit(toAccount,amount).result()=true) −− (DP)
  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, invoking the *withdraw* operation in the *Controller* is added to the current sequence diagram (refer to Fig. 6.17(e)) and the proof obligation evolves again by replacing (WD) by the relevant part of the *withdraw* postcondition (refer to Fig. 6.17(f)). 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. The proof obligation is given below:

```
Proof Obligation 3: PF3

context BankSession
  self^checkAccess(fromAccount, withdraw).hasReturned() and
    self^checkAccess(fromAccount, withdraw).result()= true and
    (if fromAccount.balance@pre >= amount
    then fromAccount.balance =
          fromAccount.balance@pre-amount) and
    bankSession^deposit(toAccount,amount).hasReturned() and
    bankSession^deposit(toAccount,amount).result()=true -- (DP)
  implies successful-transfer
```

At this point an inspection of PF3 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.

6.4 Summary

In this chapter, we present 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. Given an initial proof obligation, its evolution during the process of establishing sequence diagrams can be mechanized since it essentially involves replacing specified parts of the proof obligations with parts of operation specifications.

Chapter 7

Conclusion and Future Work

In this dissertation, we described an AOM approach to isolating features that enforce access control policies as *aspects*. To illustrate the approach, RBAC and BLP were modeled as aspects and they were individually verified against given policy requirements represented by *policy models*.

There are cases in which the applications in an organization must enforce more than one access control policy. In such cases, one needs to integrate multiple access control aspects and the resulting aspect must enforce the integrated policies. This dissertation shows how aspects can be composed to produce integrated access control features. One advantage of integrating aspects instead of composing the aspects one at a time with a primary model, is that one gets a view of the relationships between concepts in the aspect models that is not cluttered with concepts from a primary model. The second advantage is that it reduces the number of times the composition process must be applied – this is significant when the composed aspect is used in several applications or in multiple places in an application.

It is often necessary to establish that composition yields a model that is correct with respect to specified properties. In this dissertation we illustrated an approach that supports generation of proof obligations during the composition of aspects and a primary model. The approach is used to generate proof obligations that must be discharged in order to verify properties that constrain the effects of sequences of operations on system states. A proof obligation can be discharged manually or with the help of automated tools. If it is determined that the proof obligation obtained at a point during the composition does not hold, then the composition can be stopped at that point. The information that is available when the composition is stopped can be used by a developer to determine what needs to be done to correct the situation. Composition directives can be used when it is determined that the problems can be solved by refactoring the models or by overriding default composition rules. In order to facilitate automation we restrict the form of properties. Given an initial proof obligation, its modification during the composition essentially involves replacing specified parts of the proof obligations with parts of operation specifications.

7.1 Lessons Learned

In our approach, access control features are modeled as aspect models using the RBML template notation. Access control policies are expressed in terms of UML class diagrams with OCL constraints. Verifying access control aspect models against given policy requirements required us to define realization mapping pairs to show the relationship of concepts in two models. These mapping rules were used to transform the OCL invariants in the policy model into invariants expressed in terms of aspect model concepts. A difficulty arose when relating generic and domain-specific concepts: An aspect model describes generic concepts while a policy model expresses domain-specific concepts. To tackle this problem we obtained a prototypical UML model from the generic aspect model. The prototypical model is one of the most-general context-specific models that can be instantiated from the aspect model. It is obtained by simply replacing parameters with the parameter names. The multiplicity parameters are replaced with the weakest form of multiplicities allowed by the aspect model. This prototypical model is used to establish that an aspect model enforces access control policies expressed as UML class models.

One can claim that directives can be determined only after two models are composed once and problems are discovered by evaluating the composed result. However we have discovered that there are cases that can be anticipated even before the composition. For example, in case of a postcondition that describes what should be the result of the operation using *result* keyword, the default merge rules produce the conjunction of two OCL expressions that has two *result* keywords. This is not the desired result (refer to an example composition for *CheckAccess* operation shown in Section 5.3). Knowledge of the default merge rules can also be used to predict composition problems. It is true that selecting the right composition directives is heavily dependent upon human knowledge. More work needs to be done on providing support for selecting composition directives.

Our verifiable composition approach allows the policy aspect to be methodically integrated with the application, and the process can also be used to incorporate multiple policies when it is applied to compose the integrated aspect model with the application. Key challenges were (1) to determine what types of property to verify and (2) to make the process of generating and evaluating the proof obligation automatable. In the current approach, we target properties that constrain the effects of sequences of operations on system states. Operation specifications are written based on a triggering operation of which behavior affects the property. The verifiable composition in our work is accomplished in two steps. First, two class diagrams are composed using composition directives to fix any known problems during the composition. Second, the proof obligation is generated from the property to verify and the operation specifications in the composed class model.

7.2 Future Work

Selecting composition directives in our aspect composition approach is currently based on a modeler's knowledge of the policies. We plan to extend the current approach by treating the composition directives as transformations and develop a more systematic approach to selecting composition directives based on pre- and postconditions associated with the transformation.

We plan to develop tool support for composition that will include support for generating proof obligations and also investigating ways of integrating existing proof tools that can be used to assist in generating and discharging proof obligations during composition of aspect and a primary model.

Security policies including access control policies may change during a mission and evolving systems must keep pace with those changes. Our approach can be extended to evolving security policies. By encapsulating security concerns (in aspects), changes can be made in the aspect, and the effects can be incorporated into the models through composition. We will investigate how aspects can be extracted from the composed model, how they can be modified, and the modified aspect once again woven with the application.

We also plan to explore an approach to the composition with traceable properties that do not require a post-composition verification. An initial idea is to make properties in aspects *traceable* throughout the composition so that the composition may be accomplished in a manner that the result must have the desired properties. Some kind of treatment of the traces obtained in the verification when the property is not satisfied would seem to be of great interest. More precisely, that the representation of the traces should be visual by using some animated representation of the UML diagrams which could help the user to locate the error source very quickly. Chapter 5 in this dissertation shows preliminary work in this direction.

We are also currently investigating techniques for transforming aspect models to programming aspects and considering using either the mapping from Themes to implementation suggested by Clarke *et. al* [5, 19] or the aspect-based model transformation techniques suggested by Simmonds *et al.* [83] as a framework for our extension.

Appendix A

An RBAC Aspect Model

A.1 Operation specifications for generic RBAC aspect model

context |User::|CreateSession(roles:Set(|Role)):|Session

pre: self.|AssignedRoles→includesAll(roles)

post: result.oclIsNew()

and self.|Session=self@pre.|Session→including(result)

and result.|Role= roles

context |User :: |AssignRole (r : |Role) : Boolean

pre: self.|Role → excludes (r)

post: self.|Role=self@pre.|Role→including(r)

context |Session :: |AddActiveRole (r : |Role) : Boolean

pre: self.|Role→excludes (r)

and self.|User.|AssignedRoles()→includes(r)

post: self.|Role=self@pre.|Role→including (r)

context |Role::|GrantPemission(p:|Permission)

```
  pre: self.|Permission→excludes(p)
  post: self.|Permission=self@pre.|Permission→including(p)

 context |User :: |DeassignRole (r : |Role) : Boolean
  pre: self.|Role → includes (r)
  post: self.|Role=self@pre.|Role→excluding(r)
      and self.|Session.|Role→excludes (r))

 context |Session :: |DropActiveRole (r : |Role) : Boolean
  pre: self.|Role→includes (r)
  post: self.|Role=self@pre.|Role→excluding (r)

 context |User :: |DeleteSession (s : |Session) : Boolean
  pre: self.|Session→includes (s)
  post: self.|Session=self@pre.|Session→excluding (s)


 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
```

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.|Permission

→exists(p|p.|Target → includes(tar)

and p.|OperationType → includes(op)))

−− End of CheckAccess specification

A.2 Operation specifications for Context-specific RBAC aspect model

context BankUser::CreateSession(roles:Set(BankRole)):BankSession

pre: self.BankRole→includesAll(roles)

post: result.oclIsNew()

and self.BankSession=self@pre.BankSession→including(result)

and result.BankRole= roles

context BankUser :: AssignRole (r : BankRole) : Boolean

pre: self.BankRole → excludes (r)

post: self.BankRole=self@pre.BankRole→including(r)

```
  context BankSession :: AddActiveRole (r : BankRole) : Boolean
    pre: self.BankRole→excludes(r)
         and self.User.BankRole→includes(r)
    post: self.BankRole=self@pre.BankRole→including(r)

  context BankSession::CheckAccess(a:Account, op:TransactionType):Boolean
    pre: true
    post: result =
         self.BankRole.Permission → exists (p |
            p.Account→includes(a) and p.TransactionType→exists(tr|tr.Type=op.Type))

  context Account::withdraw(amount:Money):Boolean
pre: balance >= amount
post: (balance = balance@pre-amount and result = true)
        or result = false

  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)
        or result = false

  context BankSession::withdraw(a:Account, amount:Integer):Boolean
pre: true
```

```
post:
let
   chkAccMsg:OclMessage =
      self^checkAccess(a, WITHDRAW),
   doOpMsg:OclMessage =
      ?^withdraw(a, amount)
in
   if chkAccMsg.hasReturned()
   then if chkAccMsg.result()=true
      then if doOpMsg.hasReturned()
         then result = doOpMsg.result()
         else result = false
         endif
      else result = false
      endif
   else result = false
   endif

  context BankSession::deposit(a:Account, amount:Integer):Boolean
pre: true
post:
let
   chkAccMsg:OclMessage =
      self^checkAccess(a, DEPOSIT),
   doOpMsg:OclMessage =
      ?^deposit(a, amount)
in
```

```
  if chkAccMsg.hasReturned()
  then if chkAccMsg.result()=true
    then if doOpMsg.hasReturned()
      then result = doOpMsg.result()
      else result = false
      endif
    else result = false
    endif
  else result = false
  endif
```

```
 context BankSession::transfer(a1:Account, a2, Account, amount:Integer):Boolean
pre: true
post:
let
  chkAccMsg1:OclMessage =
    self^checkAccess(a1, TRANSFER),
  chkAccMsg2:OclMessage =
    self^checkAccess(a2, TRANSFER),
  doOpMsg:OclMessage =
    ?^transfer(a1, a2, amount)
in
  if chkAccMsg1.hasReturned() and chkAccMsg2.hasReturned()
  then if chkAccMsg1.result()=true and chkAccMsg2.result()=true
    then if doOpMsg.hasReturned()
      then result = doOpMsg.result()
    else result = false
    endif
  else result = false
  endif
else result = false
endif
```

Appendix B

A BLP Aspect Model

B.1 Operation/Invariant templates for BLP

context SecurityLevel::GetAllDominatees(): Set(SecurityLevel)

body: self.dominatee→union(self.dominatee → collect(sl | sl.dominatee)→asSet())

→ union(self)

context SecurityLevel::GetAllDominators(): Set(SecurityLevel)

body: self.dominator→union(self.dominator → collect(sl | sl.dominator)→asSet())

→ union(self)

context SecurityLevel::GetAllDominatees(): Set(SecurityLevel)

body: self.dominatee→union(self.dominatee → collect(sl | sl.dominatee)→asSet())

→ union(self)

context SecurityLevel::AddDominatee (sl:SecurityLevel):Boolean

pre: self<>sl

and self.dominatee→excludes(sl)

and self.GetAllDominators()→excludes(sl)

post: self.dominatee=self@pre.dominatee→including(sl)

```
context User::CreateSubject(sl:SecurityLevel):Subject
  pre: self.SecurityLevel.GetAllDominatees()→includes(sl)
  post: result.oclIsNew()
        and self.Subject=self@pre.Subject→including(result)
        and result.SecurityLevel=sl

context User :: DeleteSubject (sb : Subject) : Boolean
  pre: self.Subject→includes (sb)
  post: self.Subject=self@pre.Subject→excluding (sb)

context Subject::CheckAccess(t:Target, op:OperationType):Boolean
  pre: true
  post: result =
        t.OperationType→includes (op)
        and
        ((op.Type = OperationType::READ and
            self.SecurityLevel.GetAllDominatees()→includes (t.SecurityLevel))
            or
        (op.Type = OperationType::WRITE and
            self.SecurityLevel = t.SecurityLevel))
```

REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 4(15):706–734, 1993.

[2] G. J. Ahn and R. Sandhu. The RSL99 Language for Role-Based Separation of Duty. In *Proceedings of the 4th ACM Workshop on Role-Based Access Control*, pages 43–54, Fairfax, VA, 1999.

[3] G. J. Ahn and M. E. Shin. Role-based authorization constraints specification using object constraint language. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 157–162, Washington, DC, USA, 2001. IEEE Computer Society.

[4] R. J. Anderson. A Security Policy Model for Clinical Information Systems. In *IEEE Symposium on Security and Privacy*, pages 30–43, Oakland, CA, May 1996.

[5] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*, pages 158–167, 2004.

[6] S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.

[7] S. Barker and A. Rosenthal. Flexible Security Policies in SQL. In *Proceedings of the 15th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Niagara-on-the-Lake, Canada, 2001.

[8] J. F. Barkley, K. Beznosov, and J. Uppal. Supporting Relationships in Access Control Using Role Based Access Control. In *Proceedings of the 4th ACM Workshop on Role-Based Access Control*, pages 55–65, Fairfax, VA, October 1999.

[9] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

[10] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. *Communications of the ACM*, 44(10):51–57, Oct 2001.

[11] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[12] E. Bertino, P. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-Based Access Control Model. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 21–30, Berlin, Germany, 2000.

[13] K.J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977.

[14] M. Jazayeri C. Ghezzi and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[15] 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.

[16] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[17] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1987.

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

[19] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[20] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.

[21] S. Clarke and J. Murphy. Developing a tool to support the application of aspect-oriented programming principles to the design phase. In *Proceedings of the International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 1998.

[22] N. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, University of London, London, U.K., 2002.

[23] N. Damianou and N. Dulay. The Ponder Policy Specification Language. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.

[24] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and T. Tonouchi. Tools for Domain-based Policy Management of Distributed Systems. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, Florence, Italy, April 2002.

[25] G. Engels, J. M. Küster, R. Heckel, and M. Lohmann. Model-based verification and validation of properties. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[26] D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House computer security series. 2003.

[27] D.F. Ferraiolo, R. Sandhu, S. Gavrila, and D. R. Kuhn an d R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, August 2001.

[28] J. L. Fiadeiro and A. Lopes. Algebraic Semantics of Coordination or What Is in a Signature. In A. Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 293–307, Amazonia, Brasil, January 1998. Springer-Verlag.

[29] R. B. France and G. Georg. Modeling fault tolerant concerns using aspects. Technical Report 02-102, Computer Science Department, Colorado State University, 2002.

[30] R. B. France, D.K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004.

[31] 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):173–185, August 2004.

[32] G. Georg, R. B. 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.

[33] G. Georg, R. B. France, and I. Ray. Designing High Integrity Systems using Aspects. In *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*, Bonn, Germany, November 2002.

[34] G. Georg, I. Ray, and R. B. France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, pages 117–126, Greenbelt, MD, December 2002. ACM Press.

[35] V. Gligor. Characteristics of Role Based Access Control. In *Proceedings of the 1st ACM/NIST on Role-Based Access Control Workshop*, Gaithersburg, MD, November 1995.

[36] D. Gluch and J. Brockway. An introduction to software engineering practices using model-based verification, 1999.

[37] D. Gluch and C. Weinstock. Model-based verification: A technology for dependable system upgrade. Technical Report CMU/SEI-98-TR-009, ADA354756, Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, 1998.

[38] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, October 2002.

[39] R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.

[40] Q. He. *Requirements-Based Access Control Analysis and Policy Specification.* PhD thesis, North Carolina State University, Raleigh, NC, 2004.

[41] M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.

[42] J. A. Hoagland, R. Pandey, and K. N. Levitt. Security Policy Specification Using a Graphical Approach. Technical Report CSE-98-3, Computer Science Department, University of California Davis, July 1998.

[43] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[44] 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.

[45] 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, October 2001.

[46] K. Kieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, October 2001.

[47] D. K. Kim. *A Meta-Modeling Approach to Specifying Patterns*. PhD thesis, Colorado State University, Fort Collins, CO, 2004.

[48] D. K. Kim, I. Ray, R. B. France, and N. Li. Modeling role-based access control using parameterized uml models. In *FASE*, pages 180–193, 2004.

[49] 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.

[50] 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*, pages 426–441, 2002.

[51] Formal Systems (Europe) Ltd. Failures-divergence refinement: Fdr2 user manual, 1997.

[52] E. Lupu and M. Sloman. Conflict Analysis for Management Policies. In *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management*, pages 430–443, San Diego, California, May 1997. Chapman & Hall.

[53] J. Michael. *A Formal Process for Testing Consistency of Composed Security Policy*. PhD thesis, George Mason University, Fairfax, Virginia, 1993.

[54] N. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building Reconfiguration Primitives into the Law of a System. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 89–97, Annapolis, MD, May 1996.

[55] J. D. Moffett. Control Principles and Role Hierarchies. In *Proceedings of the 3rd ACM/NIST on Role-Based Access Control Workshop*, Fairfax, VA, October 1998.

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

[57] OASIS. XACML Language Proposal, Version 0.8. Technical report, Organization for the Advancement of Structured Information Standards, January 2002. Available electronically from http://www.oasis-open.org/committees/xacml.

[58] OMG Adopted Specification ptc/03-10-04. The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, http://www.omg.org.

[59] R. Ortalo. A Flexible Method for Information Systems Security Policy Specification. In *Proceedings of the 5th European Symposium on Research in Computer Security*, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

[60] 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.

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

[62] J. A. D. Pace and M. R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, Oct. 2001.

[63] C. Pfleeger and S. Pfleeger. *Security in Computing, Third Edition*. Prentice Hall, 2003.

[64] J. Ramachandran. *Designing Security Architecture Solutions*. John Wiley & Sons, 2002.

[65] A. Rashid. A Hybrid Approach to Separation of Concerns: The Story of SADES. In *3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), Springer-Verlag Lecture Notes in Computer Science 2192*, pages 231–249, Kyoto, Japan, September 25–28 2001.

[66] A. Rashid and R. Chitchyan. Persistence as an Aspect. In *2nd International Conference on Aspect-Oriented Software Development, ACM*, pages 120–129, Boston, March 2003.

[67] A. Rashid, A. Moreira, and J. Araujo. Modularization and Composition of Aspectual Requirements. In *2nd International Conference on Aspect-Oriented Software Development, ACM*, pages 11–20, Boston, March 2003.

[68] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *IEEE Joint International Conference on Requirements Engineering, IEEE Computer Society Press*, pages 199–202, Essen, Germany, September 9–13 2002.

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

[70] I. Ray, N. Li, R. B. 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)*, pages 31–40, 2004.

[71] I. Ray, N. Li, D. K. Kim, and R. France. Using parameterized UML to specify and compose access control models. In *Proceedings of Sixth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2003)*, 2003.

[72] R. Reddy, R. B. France, S. Ghosh, F. Fleury, and B. Baudry. Model composition - a signature based approach. In *Proceedings Aspect Oriented Modeling workshop held with MODELS/UML 2005*, Montego Bay, Jamaica, October 2005.

[73] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *in The Transactions on Aspect-Oriented Software Development*, 2006.

[74] C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[75] P. Samarati and S. Vimercati. Access Control: Policies, Models and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196, September 2000.

[76] R. Sandhu. Role-Based Access Control. In *Advances in Computers*, volume 46. Academic Press, 1998.

[77] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[78] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 47–61, Berlin, Germany, July 2000.

[79] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications*, 32(9):40–48, September 1994.

[80] E. Sibley. Experiments in Organizational Policy Representation: Results to Date. In *Proceedings of the IEEE International Conference on Systems Man and Cybernetics*, pages 337–342, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[81] E. Sibley, J. Michael, and R. Wexelblat. Use of an Experimental Policy Workbench: Description and Preliminary Results. In C. Landwehr and S. Jajodia, editors, *Database Security V: Status and Prospects*, pages 47–76. Elsevier Science Publishers, 1992.

[82] A. R. Silva. Separation and composition of overlapping and interacting concerns. In *OOPSLA '99 First Workshop on Multi-Dimensional separation of Concerns in Object-Oriented Systems*, Denver, Colorado, November 1999.

[83] D. Simmonds. *transforming Aspect-Oriented Design Models*. PhD thesis, Colorado State University, Fort Collins, CO, 2006.

[84] E. Song, Y. R. Reddy, R. B. France, I. Ray, G. Georg, and Roger Alexander. Verifiable composition of access control and application features. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 120–129, New York, NY, USA, 2005. ACM Press.

[85] M. W. A. Steen and J. Derrick. Formalizing ODP Enterprise Policies. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference*, Mannheim, Germany, September 1999.

[86] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman. Model composition directives. In *Proceedings of the International Conference on the UML, October 2004*, pages 84–97. Springer, 2004.

[87] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, October 2001.

[88] J. Suzuki and Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.

[89] The Object Management Group. The Common Warehouse Metamodel (CWM) Specification. Version 1.0, OMG, http://www.omg.org, 2001.

[90] The Object Management Group. UML 2.0: Superstructure Specification. Version 2.0, OMG, formal/05-07-04, 2005.

[91] The Object Management Group. OCL Specification, v2.0. Version 2.0, OMG, formal/06-05-01, May 2006.

[92] 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.

[93] I. Traore and D. B. Aredo. Enhancing structured review with model-based verification. *IEEE Trans. Softw. Eng.*, 30(11):736–753, 2004.

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