Chapter 1

INTRODUCTION

1.1 Problem Statement

Today, most security requirements, such as access and flow control policies, are considered only after the completion of functional requirements because security requirements are considered non-functional requirements, which are difficult to express, to analyze, and to test, and because languages used to specify access and flow control policies (such as FAF [JSSS01], PERMIS [CO02], Author-X [BBC+00] and PDL [LBN99]) are separate from the languages used to model functional requirements (such as UML) during the software development life cycle. Consequently, security considerations may not be properly engineered during the software development life cycle, and less secure systems may result.

1.2 Thesis Statement

Devanbu and Stubblebine [DS00] challenged the academic community to adopt and extend standard modeling languages such as UML to include security-related features. I accepted this challenge by showing that:

- It is possible to incorporate access and flow control policies with other functional requirements during the early phases of the software development life cycle by extending the Unified Modeling Language (UML) to include security features as first class citizens.
- It is possible to develop tools that help software analysts and designers to verify the compliance of the access and flow control requirements with with policy before proceeding to other phases of the software development process.

I substantiated my claim by:

1. Extending the metamodel of UML to incorporate access and flow control policies in the design.
2. Enhancing and extending the Use Case model by providing a unified specification of access and flow control policies using object constraint language (OCL).
3. Designing a formal framework to detect inconsistency, incompleteness, and application-definable conflict among access control policies.
4. Designing a formal framework that verifies the compliance of information flow requirements with information flow control policies.
5. Integrating both frameworks to analyze both access and flow control policies at the same time.

1.3 Significance of Contributions

Access and flow control policies in software security have not been well integrated with functional specifications during the requirements engineering and modeling phases of the software development life cycle. Security is considered to be a non-functional requirement (NFR) [CNY+00]. Such requirements are difficult to express, analyze, and test; therefore, they are usually evaluated subjectively. Because NFRs tend to be properties of a system as a whole [CNY+00, NE00],, most security requirements are considered after the analysis of the functional requirements [DS00]. The consequences of ignoring NFR are low-quality and inconsistent software, unsatisfied stakeholders, and more time and cost in re-engineering [CNY+00]. Therefore, integrating security into the software life cycle throughout all its phases adds value to the outcome of the process.

It is important to specify access control policies precisely and in sufficient detail, because ambiguities in requirements specifications can result in erroneous software [GW89]. In addition, careful consideration of requirements – including NFRs – will result in reducing project cost and time, because errors that are not detected early can propagate into the other phases of the software development life cycle, where the cost of detection and removal is high [DS00] [Boe81]. By analyzing large projects in IBM, GTE, and TRW, Boehm [Boe81] computed the cost of removing errors in general made during the various phases of the development life cycle, as shown in table 1.

Table 1: Relative Cost to Correct an Error

Phase where the error is found	Cost ratio
Requirements	1
Design	3-6
Code	10
Development test	15-35
Acceptance test	40-75
Operation	30-1000

In UML-based software design methodologies, requirements are specified using Use Cases at the beginning of the life cycle. Use Cases specify actors and their intended usage of the envisioned system. Nevertheless, a Use Case is written in natural language, which lacks the precision and specification of security [DS00]. Therefore, there is a need to provide a unified language for representing security features, such as access and flow control policies [DS00, CNY+00], in the early phases of the software development life cycle. This language must allow software developers to model access control policies in a unified way and it must be compatible with other requirements modeling languages.

In addition, there is a need to verify the compliances of security requirements with the security policies before proceeding to other phases of the software development life cycle [NE00, Pfl98, Rus01]. I used Logic as the underlying language because it is potentially amenable to automated reasoning [NE00, Rus01].

My dissertation partially fulfills Devanbu and Stubblebine's challenge [DS00], because totally satisfying their requirement has to consider all aspects of security in all phases of the software development life cycles. My contributions meet the challenge in the requirements, analysis and design phases only by specifying and verifying access and flow control policies there.

1.4 Summary of Contributions

My dissertation introduced several contributions that assist software developers to specify and analyze access and flow control policies during the first three phases of the software development process, requirement specification, analysis, and design phases. The following summarize my contributions:

6. I extended the UML Metamodel in a way that allows systems designers to model dynamic and static access control policies as well as flow control policies in a unified way. The extension provides a better way to integrate and impose authorization policies on commercial off-the-shelf (COTS) and mobile code. I showed how this extension allows non-security experts to represent access control models, such as Role-Based Access Control (RBAC) and workflow policies, in an uncomplicated manner.
7. I extended the Use Case model to specify access control policies precisely and unambiguously with sufficient details in the UML's Use Case. I added to Use Cases by using something analogous to operation schemas [SS00], which I called *access control policy schemas*. The extension employs the Object Constraint Language (OCL) [OCL01], which is more formal than the existing Use Case language (natural language) for specifying access and flow control policies.

8. I developed a framework called AuthUML that formally verifies the compliance of access control requirements with the access control policies during the requirement specification and analysis phases using Prolog style stratified logic programming.
9. I developed a framework called FlowUML to verify the proper enforcement of information flow control policies on the requirements.
10. I incorporated the analysis of both access and flow control requirements by integrating both AuthUML and FlowUML. The incorporation of both frameworks improves the analysis and detection of improper access and flow control requirement.

Based on my work in this dissertation I published several papers.

1.5 Organization of the Dissertation

Chapter 2 summarizes the literature that is related to my work, it also analyzes and compares the work with what I presented in this dissertation. Chapter 3 summarizes background works that are used as bases for my extensions, such as the UML, FAF and Operation Schemas. Chapter 4 presents the extension of the UML Metamodel to design access and flow control policies, and it shows the application of the extension on different existing access control models. Chapter 5 presents the extension of the Use Case to formally specify access and flow control requirements, and it shows the extension of the Use Case diagram and how to analyze the access control requirements visually. Chapter 6 introduces AuthUML, a framework to verify and detect improper access control requirements. Chapter 7 presents the FlowUML, a framework that analyzes information flow control requirement and detects violation of information flow control policies. Chapter 8 incorporates the analysis of both AuthUML and FlowUML and produces a coherent framework to verify both access and flow control requirements. Finally, summary of my contributions and discussion of future research are presented in chapter 9.

Chapter 2

LITERATURE REVIEW

Several new papers have been published in this area; those works concentrate on different aspects of security features and software development phases. However, there are some drawbacks in those works that need to be improved; further, some additional issues need to be addressed.

There are several aspects of security that need to be integrated into the software development process such as access control policies, flow control policies, authentication, integrity, and encryptions. Likewise, there are different phases of software development such as requirements specification, analysis, design, implementation, and testing, that require security to be integrated with them for better secure software systems.

In this dissertation, I have focused on five aspects of integrating access and flow control policies during requirement engineering. First, I extended the UML metamodel to allow the proper specification of access and flow control policies. Second, I extended the Use Case model to formally specify access and flow control policies. Third, I developed a framework to verify the access control requirements. Fourth, I developed a framework to verify the flow control requirements. Both frameworks detect improper access and flow control requirements as early as possible during the software process. The following sections summarize the literatures related to each aspect.

2.1 Extending the UML Metamodel and Use Case Model

Lodderstedt *et al.* [LBD02] proposed a methodology to model access control policies and integrate them into a model-driven software development process. The work was based on RBAC as a security model. My work is differs from [LBD02] by concentrating on specifying dynamic access control policies (e.g. dynamic separation of duty) and workflow as well as static access control policies. Furthermore, I focused on dynamic design modeling while Lodderstedt's focus was on static design model. Also, my prospective view of enforcing constraints is from the flow view not from the static view.
There are several issues missing from the work of Lodderstedt *et al.* First, history-related constraints cannot be modeled with Lodderstedt's method. Second, the metamodel is not flexible enough to model all access control policies, because it is based on RBAC only. Third, the metamodel cannot restrict people in senior roles from performing certain junior operations and it cannot specify conflict among users, operations, or roles.
Fernandez-Medina *et al.* [FPS01] introduced a language called Object Security constraint Language (OSCL). OSCL extends the Object Constraint Language (OCL) [WK99] to specify security constraints to represent multi-level security systems. Also, Fernandez-Medina *et al.* in [FMM+02] proposed an extension to the Use Case and Class models of the UML. The extensions of Use Case diagram which they introduced were stereotypes: <<safe-UC>> and <<accredited -actor>> as an indication of a secure Use Case and

authorized actor. Their work is focused on database security and shows how to model multilevel security on the static diagram such as Class diagram by introducing tagged values to classes, attributes, operations, and association ends where those tagged values indicate the security level of the element. However, this extension did not represent dynamic authorization and workflow policies. Also, the extension was limited to multilevel security model. Finally, the extension did not address the type of authorization that is granted to the accredited actor, nor the integrity constraints associated with such authorizations.

Brose *et al.* [BKL02] extended the UML to support the automatic generation of the access control policies to configure a CORBA-based infrastructure for view-based access control. It stated permissions and prohibitions of accessing system's objects (read, write, execute...etc) explicitly by writing notes that are attached to actors in the Use Case diagrams. However, their work was based on static specification of access policies but it could not model dynamic access control policies such as *Dynamic Separation of Duty* nor it could enforce some flow requirement such as the order of operations in a specific workflow systems. Although, that work covered most parts of the software development life cycle, it did not integrate access control policies in the interaction diagrams such as the Sequence diagram, and that what I presented in this dissertation. In addition, the specification language of that work was natural language which is imprecise. Therefore, I used the OCL to specify the constraints more precisely. Finally, that work considered role hierarchies, but no propagation or conflict resolution policies have been addressed for the inherited authorizations.

Jurjens's work in [Jur01] extended the UML to integrate standards concepts from formal methods regarding multi-level secure system and security protocols. His work was based on Mandatory Access Control, as in [FPS01] [LBD02], limiting the number of access control policies that can be specified using that extension.

Koch and Parisi-Presicce proposed in [KP00] how to integrate access control policies using existed UML diagrams (Class and Object diagrams). Both their work and mine share the importance of specifying access control constraints by using the OCL. However, my extension focuses more on the constraints that is why I introduced a constraint repository called the *Security Policy Constraint* (SPC). Also, my work focus more on specifying the dynamic access control policies by introducing the necessary repository for *Conflict* sets, *History logs*, *Business Task*. In addition, their work is based on static UML diagram (Class and Object diagrams) while my work relies on interaction diagram (Sequence diagram). They proposed to verify the coherent of access control specification by using the graph-based formal semantics while I used logic-based rules.

Shin and Ahn work in [SA00] modeled RBAC in the UML notations from three views: static, functional and dynamic view. Their work focal point was representing RBAC in the UML but not how to incorporate authorization models as RBAC in a real secure application design. Although, the work models constraints that are imposed on user, role, session and permission, the work did not provide more fine-grained specification of constraints needed to design dynamic authorization policies.

Ahn and Shin specified in [AS01] role-based authorization constraints using OCL. The work discussed several authorization constraints then presented it by OCL. However, the work did not show how to represent role-based authorization constraints using OCL and how to model those constraints using the UML diagrams that helps developers to integrate those constraints in the design phase.

Fernandez and Hawkins proposed in [FH97] an extension to the Use Cases. The extension was by means of a stereotype that states the access constraints. In addition, they proposed an approach to generate rights for roles. That work did not address complications arising out of hierarchies and how to resolve access control conflicts.

Ray et al. [RLK+03] proposed a technique to model and compose RBAC and MAC using the UML. Also, [RLK+03] showed how to identify conflicts arising due to such compositions. However, they do not address writing and enforcing access control constraints in detail as done in [AgW03]. That work of [RLK+03] focused on representing access control model but not representing access control constraints. Therefore [RLK+03] could be classified as a step towards integrating RBAC and MAC in the UML rather than addressing integrating general access control policies using the UML.

Operation schemas introduced by Sendall and Strohmeier [SS00] enriched Use Cases by introducing conceptual operations and specifying their properties using OCL syntax. Operation schema specifies operations that apply to the whole system to be taken as one entity. One of the advantages of operation schemas is that they can be directly mapped to collaboration diagrams that are used later in the analysis and design phases of the software development life cycle. I extended that work to express access and flow control policies in the requirement phase.

2.2 Analyzing Access Control Requirements

Ahn and Sandhu purposed in [AS00] a formal language called RCL2000 for specifying role-based authorization constraints. It identified useful role-based authorization constraints such as prohibition and obligation constraints using RCL2000 but its main users are security policy designers and security researchers who need to understand the organization objectives. RCL2000 did not address state or time nor it specified history based separation of duty. Also, RCL2000 is not designed to be used in the software requirements and analysis. In contrast, I developed a formal language that is useful during the software development to detect inconsistency, incompleteness and conflicts among access control requirements.

In the area of access control enforcement language, Brose [Bro00] presented an access control language that allows security administrators to specify access control policies in CORBA. However, the language did not detect inconsistency or conflict of access control policies.

There are several access control mechanisms such as FAF [JSSS01], PERMIS [CO02] and Author-X [BBC+00]. However, the FAF is different from others because it addresses three important issues: 1) propagation of authorization, 2) managing conflicts between positive and negative, 3) providing a final and unique access control decision (positive or negative) for each request. Moreover, FAF does not bind each issue to a specific policy but leaves the choice of policy open - resulting in a more flexible access control model. It provides the system security officer with rules that can enforce authorizations, derived authorizations, and conflict resolution and integrity constraints checking.

FAF is not designed to be used during the software development, rather, it is designed to be used in real-time checking of system calls. I presented a framework called AuthUML that advances the application of FAF to the requirements specification phase of the software development life cycle. Therefore, AuthUML is a customized version of FAF that is to be used in requirements engineering. Therefore, AuthUML uses similar components of FAF with some modification in the language and the process to suit the Use Case model used in UML. Because FAF specifies authorization modules in computing systems, FAF is invoked per each authorization request. Contrastingly, AuthUML is to be used by requirements engineers to avoid conflicts and incompleteness of accesses. Therefore, while FAF is used frequently to process each access control request during execution, AuthUML is to be used less frequently during the requirements engineering phase to analyze the access control requirements.

Koch and Parisi-Presicce presented in [KP03] an approach of model-driven to specify access control policies in the analysis phase of the software development life cycle. They showed how to derive access control requirements models from functional models. The access control model is represented by graph-based formal semantic which allow the verification of security constraints. However, my work is based on formal logic instead of graph-based which it is potentially amenable to automated reasoning that is useful in analysis [Rus01] [NE00].

2.3 Analyzing Flow Control Requirements

Although, information flow has a rich publication history, most papers concentrated on designing newer and richer flow control models. For example, based on the discretionary access control model, Samarati et al. [SBC+97] described a model that prevents information leakage by Trojans. Also, Bertino et al. [BA99] presented a logic programming based specification framework to enforce workflows constraints. Although important, these papers did not address flow policies and verifications techniques that go hand in glove with software design life cycle models that use UML.

At the other end, Myers [Mye99] presented JFlow, an extension to Java that adds statically checkable flow constraints. However, JFlow can be used during the implementation phase while FlowUML which is the framework I presented is to be used during the requirements, design and analysis phases of the software development life cycle. JFlow concerns more on controlling the information flow between the variables and objects in the programming language.

FlexFlow of Chen et al. [CWJ03] is logic based flexible flow control framework to specify data-flow, workflow and transaction systems. Although FlowUML and FlexFlow analyze and prevent unauthorized flows, FlowUML is different from FlexFlow in many aspects: 1) FlexFlow is meant to be used at the execution time, while FlowUML is intended to be executed earlier in the development process to prevent unsafe information flow from getting implemented, 2) FlowUML looks at validating information flow control from the software engineering perspective view rather than the system perspective view, 3) FlowUML add the time of flow and the initiator of flow as factors to analyze information flow that are absent in FlexFlow.

2.4 Summary

In the area of integrating access control policies into the design phase of the software development life cycle, most of the current works – I know- are built to specify static policies but not dynamic. Also, they are designed to fit with a specific access control model such as RBAC or MAC, but not general enough to express other access control policies or other access control model. In the area of analyzing access and flow control policies during the analysis phase, to the best of my knowledge, I am not aware of work that combines the analysis of both access and flow control policy during the early software development life cycle.

Chapter 3

BACKGROUND

This chapter summarizes the background used during the dissertation, e.g., the Unified Modeling Language (UML), Flexible Authorization Framework (FAF) and Access control models.

3.1 Unified Modeling Language

Modeling is the blueprint of software applications. It is used to guarantee that all business requirements are considered before starting coding. Also, modeling is used to analyze system's requirements and their consequences. In the 1980s and 1990s, several object-oriented analysis and design methods introduced with different notations. Therefore, there was a need for standardizing modeling notations. The outcome was the Unified Modeling Language (UML) [OMG01].

The following subsection will draw a brief background of the UML and some of its components that are related to my work.

3.1.1 Overview of the UML

The UML is defined according to Booch et al. [BRJ99] as follow: "The UML is a language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system."

The UML was first developed by Grady Booch, Jim Rumbaugh and Ivar Jacobson. In 1997, the UML has been submitted to and approved by the Object Management Group (OMG) [OMG01] as the OMG standard for object-oriented modeling. Since then, many versions of the UML have been developed, and the current version is 1.5. OMG's Revision Task Force (RTF) is working on version 2.0.

An important fact is that the UML is not a method, but a modeling language. Most methods consist of a modeling language and a process. The process consists of steps while the modeling language is a collection of notions that the method uses to convey the requirement to a design [Boe81].

The UML has several advantages:

- The UML is an expressive language with a rich set of notations that lets designer model any type of application that runs on any kind of hardware, programming languages or operating systems. The UML has multiple views to represent requirements and specifications.
- The UML is beneficial for communication. It is a unified language that allows one to communicate conceptual modeling with others clearly. It is between natural language, which is too imprecise, and code that is too detailed.

- The UML is extensible. The well-defined extension mechanisms that the UML has presented provide the possibility of extending the UML to facilitate new domains such as security and performance. The extension constructs are *stereotypes, tagged values* and *constraints*. *Stereotypes* are used to define new types of model elements or building blocks. *Tagged values* extend the metamodel types by adding new attributes. *Constraints* extend the semantics of the model element by adding or modifying rules.

There are twelve kinds of diagrams in the UML that are categorized as follow:

- **Structural Diagrams**: These diagrams represent the static part of the model that is conceptual or physical. The structural diagrams include the Class Diagram, Object Diagram, Components Diagram and Deployment Diagram.
- **Behavioral Diagrams**: These diagrams represent the dynamic part of the model over time and space. The behavioral diagrams contain: the Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram and Statechart Diagram.
- **Model Management Diagrams**: These are the organizational parts of the UML. They include Packages, Subsystems and Models.

3.1.2 Use Case

Use case is a behavioral diagram used to model requirements at the beginning of the software development life cycle. According to Booch *et al.* [BRJ99] *"A use case is a* *description of a set of sequence of actions, including variants that a system performs to yield an observable result of value to an actor"*

Use cases specify actors and their intended usage of the envisioned system. Such usage - usually, but not always - is specified in terms of the interactions between the actors and the system, thereby specifying the behavioral requirements of the proposed software. The scenario contains a normal scenario, and alternatives. The actor represents a coherent set of roles that the users of a use case exercise during the interaction. An actor may be a human, a hardware device or an external system that needs to interact with the system under design.

Use cases are written in an informal natural language that is easy for modelers and stakeholders to use and to understand. There is no standard on how to develop the use case or to what degree a modeler should go. Thus, different people may write varying degrees of details for the same use case. A use case may carry a high level goal such as the whole system goal, or a low level operation. The modeler is free on which degree of decomposing. A use case is a textual description that may include – but is not limited to: actors, preconditions, post conditions, normal scenarios and abnormal or exceptional scenarios. Figure 3.1 shows an example of a use case description:

In contrast, a use case diagram visualizes actors and their relationships with use cases. A use case diagram is essential for visualizing, specifying, and documenting the behavior of an element. It provides a perspective view of the system interaction. See Figure 3.2 for an example of a use case diagram.

Use Case Name: Sign a check

Actor: Manager

Precondition: The check must be issued before and not by the same user.

Normal Scenario: Manager searches for checks to be signed, and makes sure that the check's amount is available at the Bank and verify the check's amount with the invoice total amount. If all conditions are correct then the Manager sign the check.

Abnormal Scenario: if the check's amount is not available at the company's bank account, then the manager postpone the signature of that check.

If the check's amount is different from the invoice's total amount then Manager send it to the Clerk for correction.

Postcondition: check is signed and amount is detected form the company's bank account.

Figure 3.1. Use Case Description

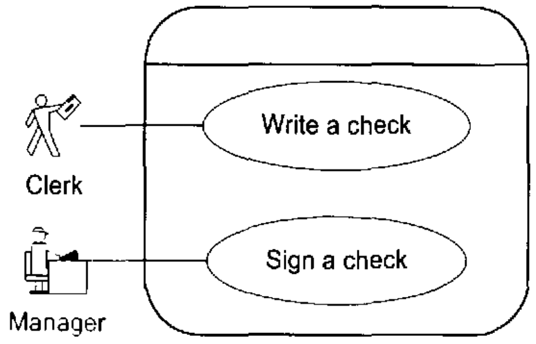


Figure 3.2: Use Case Diagram

The structure of the use case can vary. There are three relationships that can be used: an *include* relationship can be used to reduce the duplications of similar behavior across more than one use case. An *extend* relationship can be used when describing a variation on normal behavior in another use case, i.e., splitting the normal scenario from the variation scenarios. A *generalization* relationship can be used when two use cases do the same work, but one of them does more (*i.e.* inheritance); it is just like the generalization among classes.

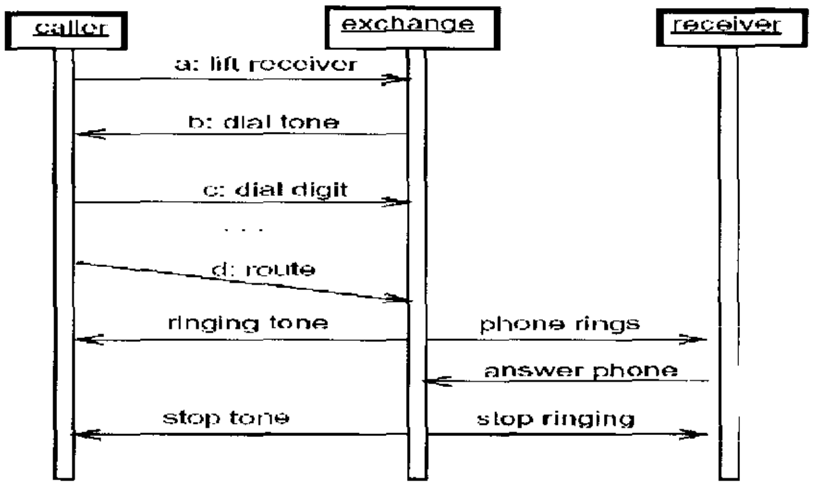


Figure 3.3. An Example of Sequence Diagram

3.1.3 Sequence Diagram

A sequence diagram is a behavioral UML diagram. The interaction diagrams describe how groups of objects collaborate in a particular behavior. For each use case, modelers need to specify how objects that participate in the use case interact with each other to achieve the goal of the use case. Sequence diagram shows the sequence of messages –

ordered with regard to time – between objects. See Figure 3.3 for an example of a sequence diagram taken from [OMG01].

3.1.4 Object Constraint Language

A writing constraint is necessary because not all constraints can be drawn in the UML. The constraint must be relatively easy for a software developer to write and read. Also, it must be precise enough to prevent ambiguities and imprecision. A natural language is easy to read and write but it is imprecise. On the other hand, formal languages are precise, but they require a strong mathematical background. OCL is based on basic set theory and logic and it is intended to be used by the software developers. OCL can be used to specify invariants, preconditions, postconditions and other kinds of constraints in the UML [WK99].

OCL employs the *design by contract* principle, where modelers can use OCL expressions to specify the pre- and postconditions of operations on all classes, interfaces, and types. OCL is an expression language where it has no side effect where it returns a value, but cannot change any values. OCL is a typed language. To be well formed, OCL expressions must obey the type conformance rules of the language. OCL provides elementary types, e.g., Boolean, Integer, Set, Bag, and Sequence, etc. According to the OMG's OCL specification of OCL [OCL01], OCL can be used to:

- Specify invariants on classes and types in the class model.
- Specify type invariant for Stereotypes.
- Describe pre- and postconditions on Operations and Methods.

- Specify constraints on operations.
- Describe Guards.
- Function as a navigation language.

3.2 Flexible Authorization Framework

The Flexible Authorization Framework (FAF) of Jajodia *et al.* [JSSS01] is a logic-based framework to specify authorizations in the form of logical programming rules. It uses a Prolog style rule base to specify access control policies that are used to derive permissions. It is based on four stages that are applied in a sequence, as shown in Figure 3.4. In the first stage of the sequence, some basic facts, such as authorization subject and object hierarchies (for example, directory structures) and a set of authorizations along with rules to derive additional authorizations, are given. The intent of this stage is to use structural properties to derive permissions. Hence, they are called *propagation policies.* Although propagation policies are flexible and expressive, they may result in *over specification* (*i.e.*, rules could be used to derive both negative and positive authorizations that may be contradictory). To avoid conflicting authorizations, the framework uses *conflict resolution policies* to resolve conflicts, which comprises the second stage. At the third stage, *decision policies* are applied to ensure the completeness of authorizations, where a decision will be made to either grant or deny every access request. This is necessary, as the framework makes no assumptions with respect to underivable authorizations, such as the closed policy. The last stage consists of checking for integrity constraints, where all authorizations that violate integrity constraints will be denied. In addition, FAF ensures that every access request is either honored or rejected, thereby providing a built-in completeness property.

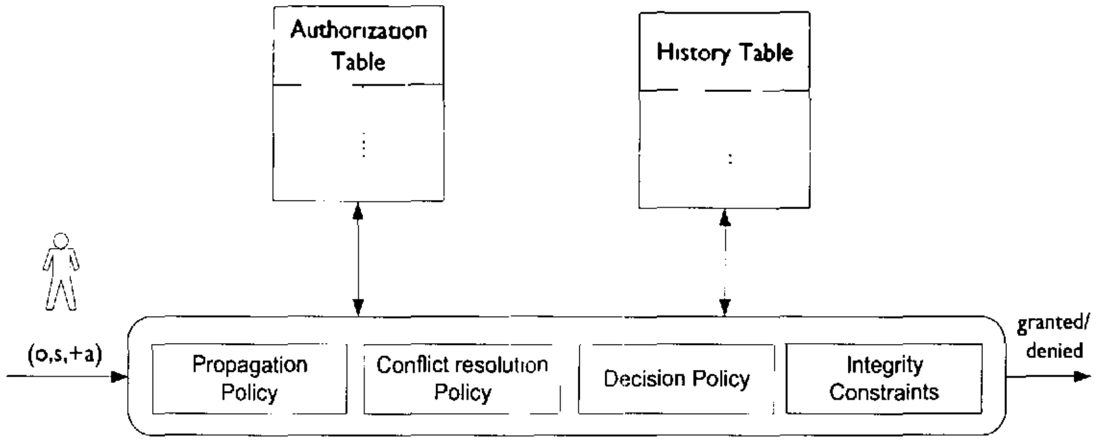


Figure 3.4. FAF System Architecture

3.3 Access control policies

Organizations have a set of policies to maintain their goals. One important policy is the access control policies. Access control policies bind the actions or activities that a legitimate user of a computer system can execute [SS94]. They protect information from unauthorized access.

3.3.1 Discretionary Access Control

Discretionary access control (DAC) restricts the access of subject to object based on the subject's identity and authorization. The object's owner at his/her discretion allows or disallows other subject to access the object.

It is a flexible model that has been adapted widely by commercial and industrial systems. However, DACs do not control the usage of information after it has been legitimately accessed. That may lead to low assurance of flow of information. For example, a subject that is authorized to read the data of an object can write that object's data to another object that allows subjects who are not authorized to read the first object to read the second object's data.

3.3.2 Mandatory Access Control

Mandatory Access Control (MAC) restricts the access of subject to object on the basis of classification of subjects and objects in the system. All objects are labeled with levels of sensitivity and all users have clearances that allow them to access objects according to the level of the objects. Flow of information is controlled in MAC-based systems by preventing information read from a high-level object to flow to a low-level object. MAC is widely used in military and government systems.

3.3.3 Role-based Access Control

In RBAC, the central issue is roles, which are absent in other two access control models. Role-based access control (RBAC) [SCFY96] governs the access of subject to object based on the role that the subject assumes during execution.

Figure 3.5 depicts the RBAC model. A user can be a human being, a process or a device. A role is a job function or title within the organization that describes the authority and responsibility conferred on a user assigned to the role. Permission is an approval of a particular action on an object. Roles are structured in a partial order relationship or hierarchy. A senior role inherits all permissions of its junior role. Each role has a set of permissions that allow the role to complete its job or function. A user must assume a role by invoking a session to perform the role's job. A user can be a member of more than one role and a role can include more than one user.

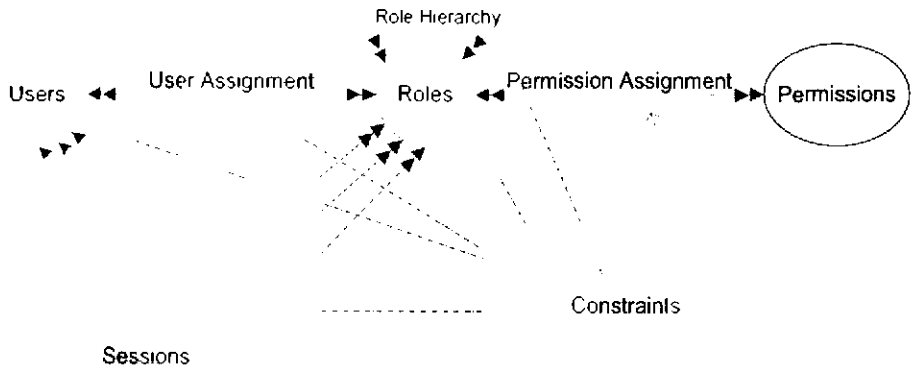


Figure 3.5. Role-based Access Control Model

RBAC has several benefits [SCFY96]:

- **Authorization management**: RBAC breaks the authorization into users, roles and permissions. This division eases the management of authorization, i.e., invoking and revoking a user from a job is a straightforward step of modifying the user assignment relationship.
- **Hierarchal roles**: another simplification of authorization management is the hierarchal relationship among roles. It is a relation of generalization/ specifications where a senior role inherits all the permissions of its junior role.
- **Least privilege**: only the permissions required for the task performed by the user in the role are assigned to the role. Least privilege principle reduces the danger of damage that may be result from errors or intruders masquerading as legitimate users.
- **Separation of duties**: completing a critical task needs the invocation of mutually exclusive roles. It prevents errors and frauds [CW87]. An example of mutual exclusive roles is the account payable manager and the purchasing manager; a user must not assume both roles. There are two major types of separation of duties: static and dynamic.

Chapter 4

EXTENDING THE UML METAMODEL

As the UML becomes the de-facto language for software design, it is important to have a sufficiently rich linguistic structure to model security specifications accurately. This chapter presents such an extension to the UML to specify dynamic access control policies in the design phase by extending the UML metamodel with a security policy specification and enforcement module, a log of method call histories and business tasks. This chapter shows that it is possible to specify access and flow control policies using the elements of the UML.

The extension of the metamodel is not a new access control model, but rather a set of related elements necessary to model existing access control models in the UML. Therefore, such elements can be incorporated differently for each access control model. An advantage of this work is the ability to enforce dynamic access control and flow control policies with the UML. These aspects of security cannot be expressed in static UML diagrams. Thus, I model them as constraints over interaction diagrams. In addition, I also focus on flow constraints using the industry standard, OCL.

Most initiatives proposed by researchers toward integrating security into the design of systems lack the representation of the dynamic access control and workflow policies. Their approaches are focused on static access control policies that can be modeled using static diagram. Dynamic access control policies rely on system states and other histories to authorize systems' users. For example, *Dynamic Separation of Duty* principles rely not on user's privileges, but also on execution history of operations.

The reminder of this chapter is organized as follows. Section 4.1 presents an example to be used during the chapter to show the features of the new Metamodel. Section 4.2 presents the new Metamodel. Section 4.3 shows how to apply the new metamodel extension to the running example. Section 4.4 presents three case studies of existed access control model and describes how to represent those models using the new Metamodel.

4.1 Running Example

This example demonstrates the application of the extension. It is based on RBAC, and it is about a typical purchasing process. As shown in Figure 3.1, it consists of three use cases: *Record invoice arrival, verify invoice validity* and *authorize payment* that is ordered as shown in Figure 4.2.

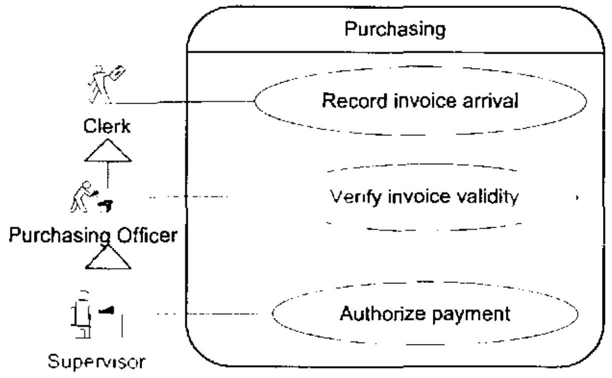


Figure 4.1. Use Cases for the Purchasing Process

The example requires different types of authorization policies:

1. **Required Sequence of operations** where each use case (except the first) has a prerequisite action, as shown in Figure 4.2. For example, *Authorize payment* cannot start until verify invoice validity is complete.
2. **Role Restrictions** where each operation can only be executed by a particular role. For example, the *Verify invoice validity* can be invoked only by the *Purchasing Officer* role. However, the *Purchasing Officer* role is a part of a role hierarchy where lower roles inherit higherrole permissions [SCFY96]. Therefore, the *Supervisor* role is allowed to execute the *Verify invoice validity*, because (s)he inherits the permission of the *Purchasing Officer* role. Nevertheless, this is not always preferred in real world scenarios. Therefore, there should be a way to explicitly restrict certain roles.

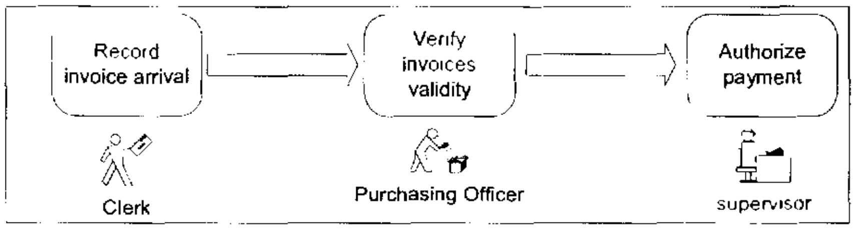


Figure 4.2. Sequencing of Purchasing Process Workflow

3. **Dynamic Separation of Duty policy** states that no user is allowed to perform all – or a specified subset – of the three operations on the same invoice, in order to avoid fraud. For example, any actor that is allowed to record the arrival of invoices must not be allowed to verify the validity of the same invoice. In my model, it is possible to enforce finer dynamic access control policies by restricting not just the roles, but the users as well.
4. **Conflict avoidance and/or resolution policy** can be enforced to avoid any conflict, either between simultaneously active roles or operations permitted for such roles.

4.2 Extension to the Metamodel of UML

This section presents my extension of the UML metamodel that allows system developers to design dynamic authorization and workflow policies in the design phase using the UML.

My metamodel extends the UML's core package [OMG01] that defines basic types required to model objects such as classes, attributes, operation and associations. I do so by using *stereotypes* for new types and *tagged values* for new attributes on existing types. Explicitly, I added three new metamodel types: 1) *security policy constraints (SPC)* that hold and impose constraints necessary to model security, it plays the role of monitor, 2) a *history log* that records all actions executed by the system and 3) *business tasks* that act as a reference architecture for encapsulating related tasks within a single entity. The new metamodel extensions are sufficiently flexible to accommodate not only RBAC, but also other access control models and workflow policies. Following subsections explain the metamodel in detail.

4.2.1 Security Policy Constraints

SPC is a specialized type of constraint element in the Core package of the UML's metamodel [OMG01]. It is shown with a broken line in Figure 4.3 with the UML core metamodel. Therefore, SPC can be related to any specialized elements from *ModelElement*. For instance, SPC can be related to *Class, Operation, Attribute* or *Association*. SPC has two associations, one with itself and the other with the *Constraint* element. These associations are used to refer to other constraints. Therefore, SPC is a set of explicit or implicit constraints that are imposed in association with other SPCs or constraint elements.

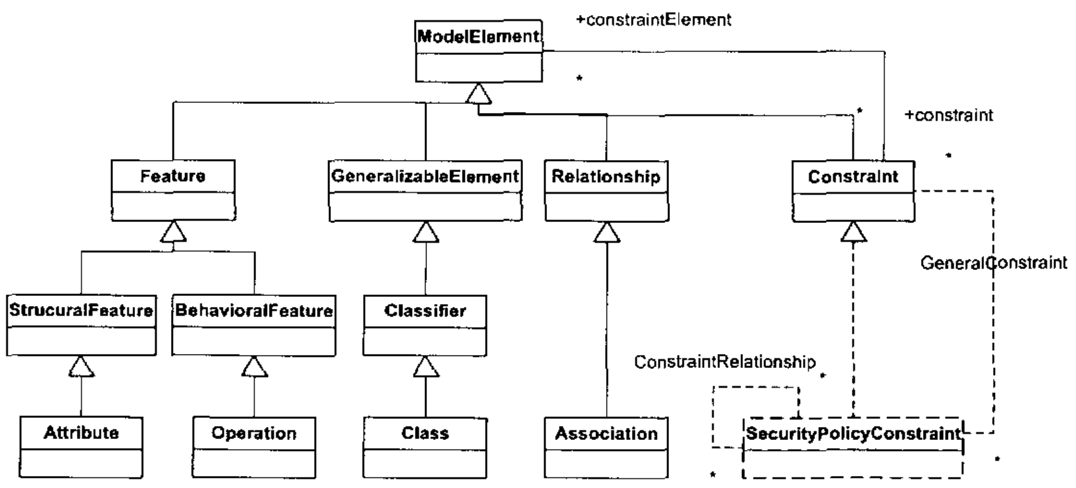


Figure 4.3. Security Policy Constraints' Relationship in the Core Package

Table 4.1 shows an example of an SPC from the *authorize payment* operation in the running example, the SPC consists of three constraints written in OCL. The first constraint limits role access to operations. The second constraint makes sure that the current user is not the same one that performed the prerequisite task. The third constraint specifies operation prerequisites. These constraints are just an example of the types of constraints in the SPC. They are written in a generic way that can be applied to other SPC's of other objects. However, constraints can also be written in a specific way (e.g., by explicitly naming the object, role, user and operation in the constraint), but that will limit their re-use.

Table 4.1. An Example of the SPC of the Authorize Payment Operation

Security Policy constraint (SPC)		
Operation	Policy	Constraint(s)
Authorize_ payment	Role Restriction	(User->select(user=CurrentUser)).role→ intersection (CurrentOperation.Allowedroles)→ size>0 AND (User→select(user=CurrentUser)).role→ intersection (CuurentOperation.Deniedroles)→ size=0
Authorize_ payment	Avoiding of Conflicting User	((ConflictingUsers->select(UserName=CurrentUser)) → collect(users)→asSet())→ intersection ((History_Log→ select(Action= (BusinessTask→ select(Task="Purchasing").Operation→ Prior (Operation=CurrentOperation))AND Object= CurrentObject))→ collect(ActionUser))→ isEmpty
Authorize_ payment	Operation Sequence (Workflow)	Histroty_Log→ select(Action=(BusinessTask→ select(Task="Purchasing").Operation→ Prior (Operation=CurrentOperation)) AND Object=CurrentObject)→ notEmpty

One of the main issues to be addressed by security constraints is their placement in the UML models and the metamodel. If all constraints are imposed on one object, then that object encompasses all access control specifications. While centralizing the design at the Meta level, this option has the advantage of having a clean separation of security policy from the rest of the design. This facilitates applying different security policies to the same design.

Conversely, each object or combination of objects can be decorated with corresponding security policy constraints. While not giving rise to difficulties in binding objects to their constraints and encapsulating policies within objects, this approach presents a security nightmare. That is, it scatters the security policies governing a piece of software throughout its design diagrams. There is also the danger of confusing the difference between security constraints from other types of constraints such as performance and design integrity. However, separating each object with its security policy constraints in different objects, such as SPCs, can solve the latter problem of mixing security policies with other types of policies, but it does not solve the problem of scattering security policies all over the design.

4.2.2 History Logs

In order to enforce history based and dynamic access control policies I introduce the history log. It maintains method and task execution histories. It was referred to as *history based controls* by Simon and Zurko [SZ97]. The history log also facilitates auditing – an important aspect in enforcing security specifications according to Sandhu et al. [SS94]. The history log can also be maintained centrally or distributed. Maintaining centralized history logs can detect inter-object conflicts, although it may become a performance bottleneck for larger designs.

4.2.3 Business Tasks

In the business world task may consist of more than one operation. For example, the purchasing task in Figure 4.2 consists of three operations: *record invoice arrival, verify invoice validity* and *authorize payment.* Because many duties, in the sense of *separation of duty principles,* are formulated in terms of business tasks, I model business tasks as first class objects. Specifying operations as a part of a business task means that no user can perform all the operations of a single business task on a single object that leads to fraud and errors.

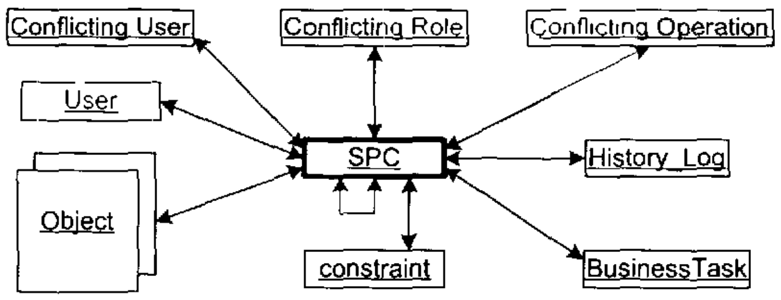


Figure 4.4. SPC Interactions with Other Elements

4.2.4 Conflicts

Some operations conflict with others. For example, a single user should not perform *the writing a check* operation and the *signing a check* operation. In addition, conflicts may occur between roles, e.g., the *purchasing officer* role and the *accounts payable manager* role. Conflict may also take place between users, e.g., relatives should not execute complementary critical tasks. For that reason, I introduce three conflict associations: *conflicting user*, *conflicting roles* and *conflicting operations*.

4.2.5 Interactions between SPCs, History Logs and Business Tasks

Figure 4.4 demonstrates how the SPC interacts with other elements of the metamodel in order to validate access control policies. The SPC is the corner stone of my model that intercepts each method's call and validates the permission of the caller. The SPC decides according to the set of all authorization constraints related to the called operation. These constraints rely on some data contained in other objects. Objects can be one of the following: First, *History_Log* contains a log of all users' actions. Second, *Business Task* contains all required operations for a particular task. This information can be used to prevent a user from executing all operations of a single task that may lead to fraud and errors. Third, conflicting sets of (roles, users, and operations) are used as a knowledge base by the SPC to prevent conflicts. Fourth, users and objects can be consulted to provide identity and other attribute values that are helpful to validate constraints.

4.2.6 Enforcing Access Control Constraints

In order to enforce security constraints, I assume that every system designed with this metamodel has a reference monitor. The reference monitor ensures that its execution does not violate security related constraints, and filters every method call. In order for this to be effective, I assume that all attributes are encapsulated inside their classes and cannot be accessed other than by calling those operations. This approach is similar to the reference monitor Sandhu et al. [SS94]. A reference monitor may not necessarily have to be implemented as a centralized entity as such a design may introduce vulnerabilities. However, it can be modeled and implemented in a decentralized manner to suit new decentralized computing paradigms. Section 4.4.3 provides an example of how to model the extension in a decentralized way.

4.3 Applying the New Extension to the Running Example

This section shows how the new metamodel can be applied to the running example described in Section 4.1. Here, I demonstrate that *Use Cases* result in flow constraints. The following are some sample security related requirements and their translations as OCL constraints:

1. **Required sequence of operations (Workflow policies):** Enforcing this policy requires writing constraints in the SPC and consulting the *History_Log* and the *Business_Task* for previous actions. The business task is consulted to get the sequence of operations in a single task, and the *History_Log* is used to validate the occurrence of previous operations. For example, this policy is enforced on the *Authorize Payment* use case by the following preconditions written in OCL:

```
Context Invoice:: Authorize_Payment():Void                    (1)
Pre: History_Log→ select(Action=(Business_Task→
            select(Task="Purchasing").Operation→
            Prior  (Operation=CurrentOperation))  AND
            Object=CurrentObject)→ notEmpty
```

2. **Role constraints:** Allow only permissible roles and restrict unauthorized roles. Suppose that the *supervisor* role is prohibited from executing the *verify invoice validity* operation. This constraint can be directly specified in OCL as follows:

```
verify_invoice_validity.AllowedRoles={PurchasingOfficer}  (2)
verify_invoice_validity.DeniedRoles={Supervisor}
```

However, to express it as a precondition in the SPC, I write it as follows:

```
Context Invoice::verify_invoice_validity (...):Void                (3)
    Pre:(User→select(user=CurrentUser)).role→
    intersection(CurrentOperation.Allowedroles)→size>0
    AND(User→select(user=CurrentUser)).role→
    intersection(CuurentOperation.Deniedroles)→size=0
```

These are written as general constraints that can enforce the *Role Restriction* policies. They consist of two expressions. The first expression intersects two sets: the current user' roles and the current operation's allowed roles (i.e., the *Verify invoice validity* operation). If the result set is not empty then the user is allowed to perform the operation and otherwise disallowed. The second expression ensures that none of the user's roles are in the current operation's denied roles. For example, a user playing the *Supervisor* role cannot perform the *verify invoice validity* operation, because it is in the operation's denied roles.

3. **Dynamic separation of duty:** According to the original specification, recording the arrival of the invoice and the authorization of the payment has to be performed by two different subjects. This DSOD requirement can be satisfied by enforcing the following precondition on the *authorize payment* method.

```
Context Invoice:: Authorize_Payment():Void                (4)
    Pre: History_Log→ select(ActionUser= CurrentUser
    AND action="record_invoice_arrival" AND Object=
    CurrentObject)→isEmpty
```

This constraint ensures that there should not be any record in the *History_Log* that shows that the current user has executed the *Record invoice arrival* operation on the current invoice. Note that there is no similar constraint on the *Authorize payment* operation because the *Supervisor* role is not allowed to perform the *Verify invoice validity* operation. Also, note that I am using users and not roles to specify this constraint, because users allow me to impose finer constraints than roles in this context where a role consists of a set of users.

4. **Conflict avoiding policies**. The conflict association in each *User, Operation,* or *Role* is used to enforce this kind of policy. For example, the following constraints ensure that the current user is not in conflict with the user who performed the previous operation of the task:

```
Context Invoice:: Authorize_Payment():Void                          (5)
  Pre:((ConflictingUsers→select(UserName=CurrentUser))→
  collect(users)→asSet())→intersection((History_Log→ select
  (                    Action=                    (Business_Task→
  select(Task="Purchasing").Operation→
  Prior(Operation=CurrentOperation))AND
  Object= CurrentObject))→ collect(ActionUser))→ isEmpty
```

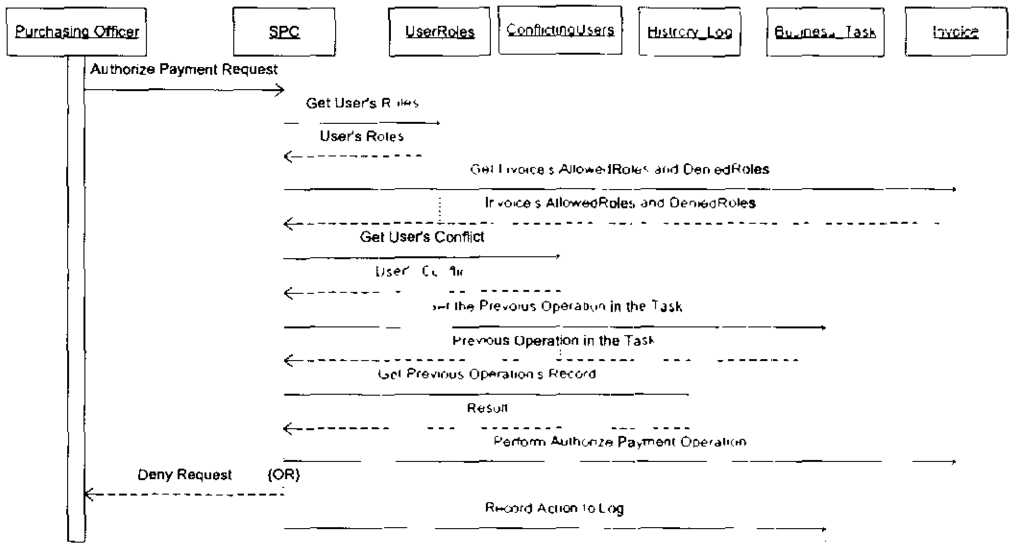

Purchasing Officer
SPC
UserRoles
ConflictingUsers
History_Log
Business_Task
Invoice
Authorize Payment Request
Get User's Roles
User's Roles
Get Invoice's AllowedRoles and DeniedRoles
Invoice's AllowedRoles and DeniedRoles
Get User's Conflict
User's Conflict
Get the Previous Operation in the Task
Previous Operation in the Task
Get Previous Operation's Record
Result
Perform Authorize Payment Operation
Deny Request
(OR)
Record Action to Log

Figure 4.5. Sequence Diagram for the Authorize Payment Use Case

Figure 4.5 shows the sequence diagram of the authorize payment use case. It is an example of how the SPC interacts with other objects to validate the call to *Authorize Payment* operation. As stated, every method call in Figure 4.5 is redirected to the reference monitor - the SPC. That shows that all security requirements are enforced and are based on the fact that we can translate all security requirements to constraints on method calls and, therefore, they can be enforced by filtering method calls at the SPC. This argument assumes that SPC can be designed to enforce all requirements so translated.

4.4 Case Study of Existing Security Models

In this section, I show how to enforce three examples of security policies taken from the security literature. The first is RBAC the second is flow control polices by Ferrari et al. [FSBJ97],; and the third is the distributed authorization processor by Kraft [Kra02]

4.4.1 Role-based Access Control Metamodel

This section shows how to enforce RBAC in a real, secure, application design. In order to enforce RBAC models, I extend the core of RBAC as suggested in the new metamodel and shown in Figure 4.6. The dotted lines represent the extension of the new metamodel to model RBAC polices in the UML, while the rest of the diagram represents my extension of the UML. The following section shows RBAC policies and how to model them using the new metamodel:

- **Dynamic separation of duty (DSOD):** The *business task* element holds all related operations of a single business task and the *history log* records all previous actions of all users on all objects. Those two types of information suffice to enforce DSOD.
- **Static separation of duty (SSOD):** The constraint associations between the SPC and other elements such as *User, Role* and *Operation* are the required information for this policy.

- **Flow control and workflow:** The SPC gets the sequence of operations in a *Business Task* and query the *history log* to check whether the previous operation to the current one is already completed.
- **Conflicts in *User*, *Role* and *Operation*:** To avoid such conflicts I specify three distinct associations that are related to *Role, User,* and *Operation.*
- **Cardinality in *Roles* and *User* elements**: I have extended *Role* element by adding tagged values: *MaxAllowedUsers* and *MinAllowedUsers*, that are used to limit the number of users in each role. For example, only one user should play the CEO role and, hence, the *MaxAllowedUsers* is 1. Note that the same is true with Users.

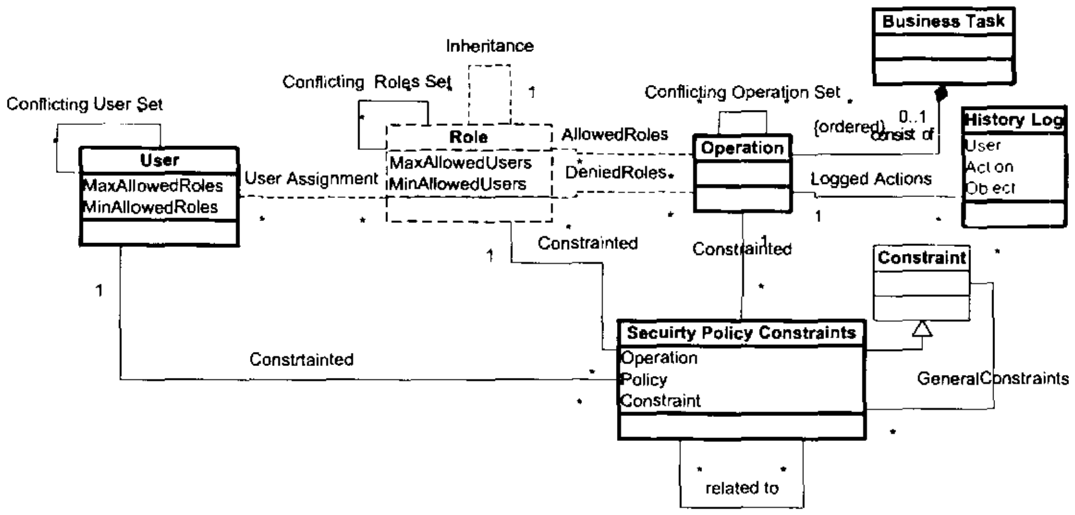


Figure 4.6. RBAC Metamodel

The extension I did for RBAC introduces two associations: *AllowedRoles* and *DeniedRoles* where the former represents all roles that are allowed to execute an operation and the later represents all denied roles. Moreover, an operation with no *AllowedRoles* association is considered open or closed to all roles according to the access control meta-policies such as open-policy or closed-policy respectively, unless some roles are denied in the *DeniedRoles* association.

4.4.2 Workflow Policies

The extension can be used to specify and enforce workflow and information flow control policies in object-oriented systems such as the flexible flow control model of Ferrari et al. [FSBJ97].

The strict policy of [FSBJ97] allows writing an object O' that depends upon the reading of object O only if all readers of O' are a subset or equal to the readers of O. But the approach they presented adds flexibility to this strict policy by introducing two types of exceptions to methods:

1. Exceptions on releasing information based on return parameters of a method called reply-waivers.
2. Exceptions for passing information by parameters of a method that are called invoke-waivers.

The model uses three components when deciding to permit an information flow. They are:

1. Access Control Lists (ACL) for each object.

2. Reply waivers for each operation where each waiver contains the objects and a set of users who are waived to receive the operation's returned value.
3. Invoke waivers for each operation where each waiver contains the objects and a set of users who are waived to invoke the operation.

The example in Figure 4.7 shows a transaction execution tree of method Op1. Each operation either reads from object or writes to object. Each object has a set of users who are allowed to access the object. Op1 calls Op2 which reads Obj1 and Obj2, and after the completion of Op2, Op1 calls Op3 which writes the information read from Op2 to Obj3. According to strict policy, Op3 cannot write to Obj3 because the information that is needed to be written in Obj3 came from more restrictive Objects than Obj3. For example, the Obj3's users set is not a subset or equal to the users of Obj1 and Obj2. Likewise, Op4 will not execute, because it intends to write information that is read from more protective object (i.e., Obj1). However, [FSBJ97] provides more flexibility than the strict policy. If I attach a reply waiver: {({Obj1},{B,C}), ({Obj2},{C})} to Op2 then Op3 can write the information because this is a safe flow according to the safe flow theorem [FSBJ97]. Also, if I attach a revoke waiver: ({{Obj1},{B}}) to Op4 then Op4 can write to Obj4 freely because now the users who can access Obj4 is a subset or equal to the ACL of Obj1 and Obj2, plus the invoke-waivers of Op4, e.g., the safe flow theorem.

One important element of the model is the message filter. It is a trusted system component that intercepts every message exchanged among objects in a transaction to prevent unsafe flows. The message filter relies on a concept called folder to check each message exchange.

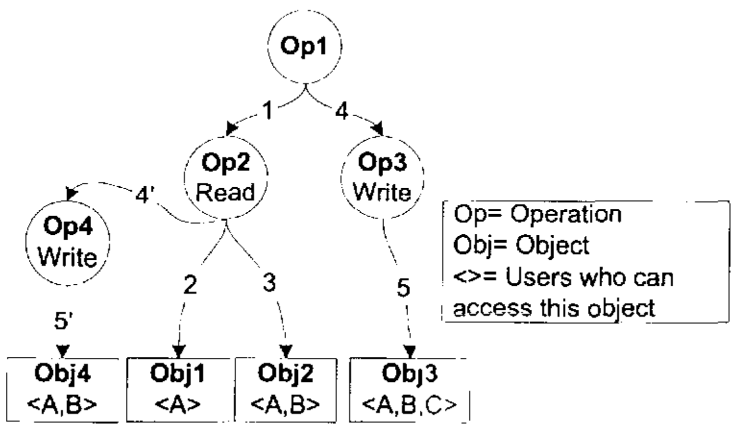


Figure 4.7. Transaction Execution Tree

A many-to-many association between user and operation can represent ACLs. For example, each object's operation may have one or many users who are allowed to access it, and each user may be allowed to access one or more of the same object's operation, as shown in Figure 4.8. Waivers are new elements attached to each object's operation in one end and to the User on the other end.

A folder is a set of records where each record holds the waivers applicable to the information flowing. To represent it in my model, I use the history log as a representation of the folder to store the required information. The history log in this model is associated at one end with operations because each folder's record should be related to one operation at one end (i.e., the first execution of the transaction) and to the user at the other end (i.e., the allowed users of each folder). The folder records consist of a transaction id, the type of folder (backward or forward), the execution source (i.e., an operation), the execution destination (i.e., an operation), the object's identification, and the object's operation, and each record is associated with a set of allowed users. The message filter intercepts each message to prevent unsafe flows by checking the ACL, the reply-waivers and the invoke-waivers, or to construct folders.

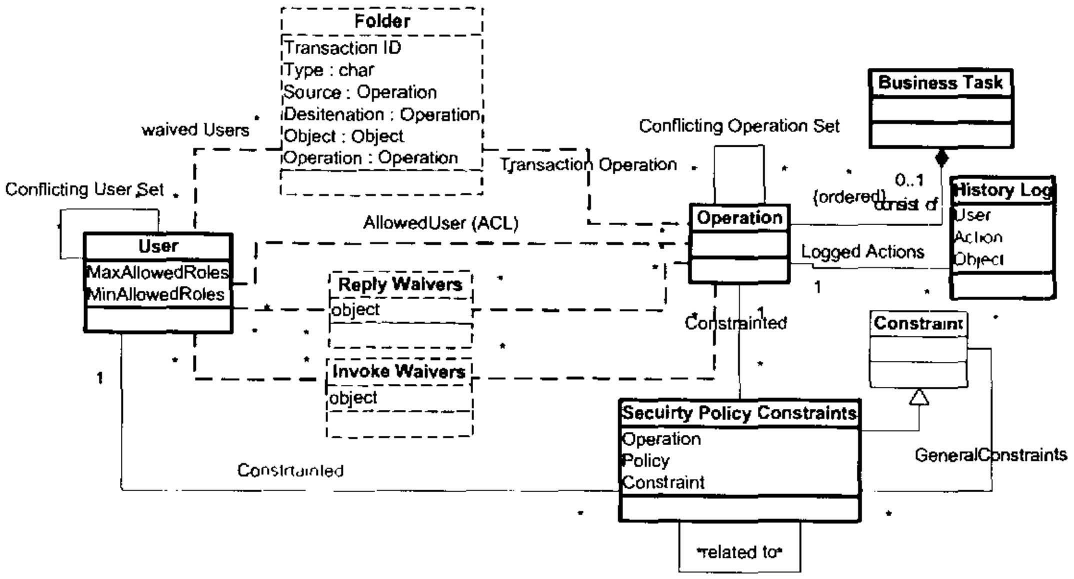


Figure 4.8. Metamodel for the Flexible Information Flow Control Model

4.4.3 Representing Distributed Authorization Model

In this section, I show how the new metamodel can be used to model a new proposal of a Web service's authorization by Kraft [Kra02]. Web services provide an easy development, deployment, maintainability and accessibility through the Internet. However, security must be imposed on Web services to succeed. There are security standards and proposals to achieve better security on Web services and one of them is a Kraft proposal [Kra02] on the Web service's authorization.

Kraft [Kra02] introduces a distributed authorization processor architecture that incorporates basic Web service objects, plus aggregation, composition, operations and specialization on Web services. The model designed as a SOAP (Simple Object Access Protocol) filter gateway that operates as an authorization service for Web services. The distributed authorization processor is based on two components: a gatekeeper and an authorization processor. Authorization Processor is a web services that makes authorization decisions for a Web services component, whereas, a gatekeeper is an authorization processor that has to make the final decision on granting or denying requests. Each Web service component may have one or more authorization processor while it may have at most one gatekeeper. Also, a gatekeeper has the function of authenticating principles of incoming requests. Another issue is that Web services may belong to a web services collection; therefore, in order to access a Web service that is a member of a collection, the gatekeeper needs to check the Web service's authorization processor and the collection's authorization processor to make the authorization decision.

A simple scenario is shown in Figure 4.9 (taken from Kraft [Kra02]). The scenario starts when a client request a Web service object 3, then the gatekeeper (#1) of the requested Web service (#3) intercepts the request to determine if the client is allowed to access the required Web service or not. Thus, first, the gatekeeper authenticates the client (I will ignore authenticating clients to focus on access control only). Second, the gatekeeper checks every access control processor that is related to the requested Web service (#3,

#6) to find out whether the client is allowed. Because the Web service object 3 is a member of the Web service object 6, the gatekeeper must also check the access control policy (ACP) (#2) that controls the access to the Web service object 6. If all access control processors accepted the request, the gatekeeper routes the request to the requested Web service otherwise, it rejects the request.

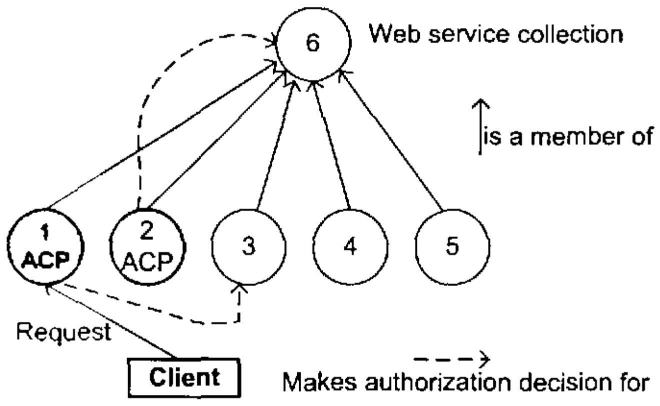


Figure 4.9. Distributed Access Control Processor Architecture Scenario

The representing Kraft model [Kra02] is straightforward using the new metamodel. Before I show how to model it, I assume that, when a client accesses a Web service, he/she is invoking an operation on that Web service. The authorization processor is a set of authorization constraints that are related to a specific Web service's operation. Therefore, the authorization processor is modeled as an SPC. Furthermore, because gatekeeper is an authorization processor, the gatekeeper is also modeled as an SPC. The

SPC is flexible to accommodate any constraints that belong either to the authorization processor or the gatekeeper.

Kraft [Kra02] introduces the Web service collection, which contains a number of Web services. The access rights of any member are based on the union of both the Web service's access rights and the collection root's access rights for the Web service. Therefore, there should be some sort of representation of the relation between a member and its root. The new metamodel provide this representation by the association "related to" that associate Web service's SPC to its root's SPC.

4.5 Conclusions

Security needs to be integrated into the software development life cycle, and propagated throughout its various phases [DS00]. Therefore, it is beneficial to have secure development integrated with industry standard methodologies and notations such as Rational Unified Process (RUP) [RUP04], Concurrent Object Modeling and Architectural Design with the UML (COMET) [Gom00] and the UML.

I extended the UML metamodel to specify and enforce access and flow control policies. I added SPC, business tasks and a history log. Then I showed how security requirements could be specified and enforced by using new extensions. These requirements are in the access control, flow control and workflow specifications. Based on an implementation of the SPC as a reference monitor, I show how to enforce security requirements specified at the requirements specification stage of the life cycle.

Chapter 5

EXTENDING THE USE CASE MODEL

In this chapter, I show how to extend the use case model in a three ways: 1) I present the *access control schemas* that unify the specification of the access control policies in the use case, 2) I present the *access control table* that visualizes the access control policies and helps in applying inconsistency and conflict resolution for small scale software systems, 3) I extend the use case diagram to show accurate access control policies.

5.1 Introduction

In the UML, requirements are specified with *use cases* at the beginning of the life cycle. *Use cases* specify actors and their intended usage of the envisioned system. Such usage - usually, but not always - is specified in terms of the interactions between the actors and the system, thereby specifying the behavioral requirements of the proposed software. Fowler and Scott say that *a use case is a set of scenarios tied together by a common user goal* [FS99]. Use cases are written in an informal natural language. Thus, different people may write varying degrees of details for the same use case. Currently, a use case is a textual description with: 1) actors and/or their roles; 2) preconditions and post conditions, 3) *normal* scenarios with sequence of actions by the actors and/or the system; 4) *abnormal* or exceptional scenarios. In contrast, a use case diagram visualizes actors and their relationships with scenarios [JCJO92, BRJ99]. As I shall demonstrate during the course of this chapter, use cases are not sufficient to model the details of access control policies. Consequently, I enhanced the use cases model by adding something analogous to (soon to be discussed) operation schemas.

Operation schemas, introduced by Sendall and Strohmeier [SS00], enrich use cases by introducing conceptual operations and specifying their properties using OCL syntax [WK99]. The operation schema can be directly mapped to collaboration diagrams that are used later in the analysis and design phases.

Although operation schemas are precise, they do not specify system security. Therefore, I extended the operation schemas to cover access control, and I refer to the extended schemas as the *access control schema*. Introducing an *access control schema* as a separate syntactic entity has several advantages. First, it isolates access control policies from other functional requirements that are usually elaborated in operation schemas. Second, this separation facilitates several access control policies to one use case, thereby modularizing the design.

There is a need for negative authorization as there is a need for positive authorization. In particular, with the presence of subject hierarchy, the need for explicit negative authorization is greater because subjects do not have explicit authorizations only, but also may have implicit authorizations from the inheritance of the junior subject's permissions. Therefore, negative authorizations are used to block some positive authorizations that have been granted to subject. With the introduction of a negative authorization, there is also a need to manage any conflict between authorizations (positive and negative).

Sometimes use cases over-specify or under-specify authorizations, leading to inconsistency and incompleteness, respectively. In order to avoid both extremes, security literatures use *conflict resolution* and *decision* meta-policies. I applied the two policies to the use cases. In addition, adhering to the visual specification tradition of the UML, I attached *access control tables* to visualize the process of applying meta-policies in conflict resolution.

Based on resolved complete and consistent policies, I constructed refined use case diagrams that illustrate access control policies visually. I developed a methodology with the following steps: 1) writing the access control policy schema, 2) developing an access control table and applying propagation, conflict resolution, and decision policies on all use cases, 3) propagating authorizations to operations, 4) resolving conflicts in the access control table for operations; and 5) drawing the refined use case diagram.

This chapter does not show how to model or implement the access control policies as FAF [JSSS01], Author-X [BBC+00] or PERMIS [CO02] does, but it rather addresses the representation and management of access control policies at the early phases of the software development life cycle - thereby focusing on how to represent and produce conflict-free and complete authorizations. The output of this work can be used later to feed other access control mechanisms such as FAF, Author-X or PERMIS.

The remainder of this chapter is organized as follows. Section 5.2 explains an example that will be used throughout the chapter. Section 5.3 describes the steps of specifying the access control policies with the use case.

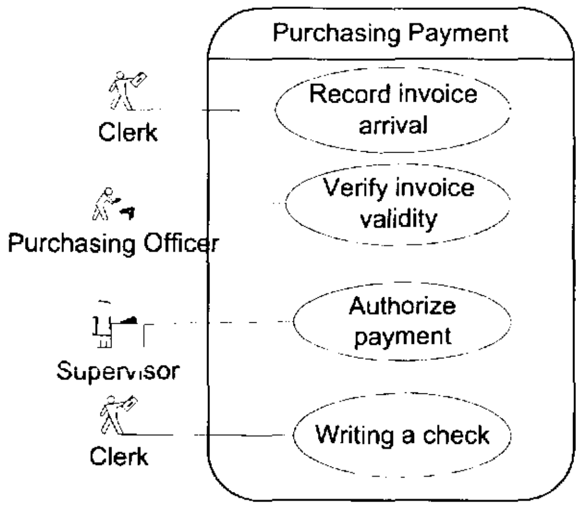


Figure 5.1. The Use Case Diagram

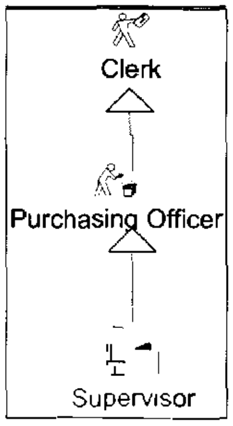


Figure 5.2. The Role Hierarchy

5.2 Running Example

The running example describes a purchasing process where a set of tasks assigned to authorized roles as shown in Figure 5.1. Role-Based Access Control (RBAC) [SCFY96] is the access control model for this example. The set of access control policies applicable to this example are as follows:

1. Use cases such as record invoice arrival, verify invoice validity, authorize payment and write a check are to be applied in the specified order.
2. Each use case should be executed by an actor playing an authorized role(s) as shown in Figure 5.1. For example, the write a check use case should be invoked by (authorized to) clerk role. In addition, the role hierarchy implicitly authorizes a specialized role to inherit permissions. For example, according to Figure 5.2, the supervisor role inherits the purchasing officer's permissions and the purchasing officer inherits the clerk's permissions.
3. Supervisor cannot execute the write a check use case.
4. No user should perform more than one use case on each object. This is a one type of Dynamic Separation of Duty (DSOD) policy. For example, a user should not record and verify the same invoice. This policy is claimed to prevent fraud and errors [CW87].
5. If the invoice's total amount exceeds one million, then two different supervisors must authorize the invoice.

5.3 Formal Steps for Specifying Access Control Policies

5.3.1 Writing Access Control Schema

Operation schemas do not cover access control policies. Therefore, I introduce the *access control schema* to specify them.

Figure 5.3 and Figure 5.4 show the standard format and an example of the *access control schema*, respectively. As shown in Figure 5.3, an *access control schema* has: 1) Use case name, 2) Object, 3) Description, 4) Declarations, 5) Users and roles that are either authorized or denied to invoke a use case, and 6) Pre and post conditions of the schema. Figure 5.4 refers to the *authorize a payment* use case of Figure 5.1. The pre-condition of the schema in Figure 5.4, has four constraints: 1) the invoice is already verified; 2) if the invoice's total amount is less or equal to one million, then the invoice must not be authorized yet, 3) if the invoice's total amount exceeds one million, then either the invoice is not yet partially authorized or partially, but not fully, authorized, and 4) the current user did not participate in any prerequisite operation on the same invoice. Conversely, the postcondition ensures the correctness of operations with respect to the access control constraints.

Use Case: the use case name.

Object: the object of the use case.

Description: short textual description of the action.

Declares: constants, variables, objects and data types used in the pre and post conditions.

Authorized (user, group, and role): a list of users, groups or roles that are authorized to access this operation on this object.

Denied (user, group, and role): a list of users, groups or roles that are denied to access to this operation on this object.

Integrity Constraints (Pre): specify all integrity constraints that must be satisfied before executing the operation written in OCL.

Integrity Constraints (Post): specify all integrity constraints that must be satisfied after the operation is executed. It is written in OCL.

Figure 5.3. Format of Access Control Schemas

Use Case: Authorize Payment

Object: Invoice

Description: Actor authorizes the payment after it has been verified. If the amount exceeds one million dollar then the authorization is partial until a different supervisor completes it.

Declares:

```
  UserWhoDidPreviousOperations: Set(History_Log) ::= History_Log→
  select (User= CurrentUser AND
  (Operation="Record_Invoice_Arrival" OR
  Operation="Verify_Invoice_Validity")AND Object= CurrentObject); -
  -it will return a record or  more if the current user has done one of the previous use case.
```

Authorized (User, Group, Role): Supervisor--*Role*

Denied (User, Group, Role): none

Integrity Constraints (Pre):

```
  Invoice.verified="true";
  Invoice.TotalAmount<=1000000 implies Invoice.authorized=
        "false";
  Invoice.TotalAmount>1000000 implies
        (Invoice.partialAuthorized= "false" OR Invoice.authorized=
        "false")
  UserWhoDidPreviousOperations → isEmpty; – The current user did not do
  other operation on the current invoice(Dynamic Separation Of Duty)
```

Integrity Constraints (Post):

```
   If (invoice.TotalAmount>1000000 AND
       invoice.partialAuthorized@pre="false") then  –the invoice has not
       been partially authorized by different Supervisor before.
       Invoice.partialAuthorized="true";
   else
       invoice.authorized= "true";
    Endif;
```

Figure 5.4. The Access Control Schema for the *Authorize Payment* Use Case

5.3.1.1 Constraints

Authorizations in the form of authorized or denied clauses in the *access control schema* do not capture all access control constraints. Therefore, there is a need to properly express application constraints such as dynamic separation of duty. Next, I will provide some access control constraints in commercial systems, and I will consider several known versions of separation of duty (SOD) policies. I show how to write SOD policies as an OCL constraint in the integrity constraint clause of the *access control schema.* Figure 5.5 illustrates the relationship between objects that are used to specify integrity constraints.

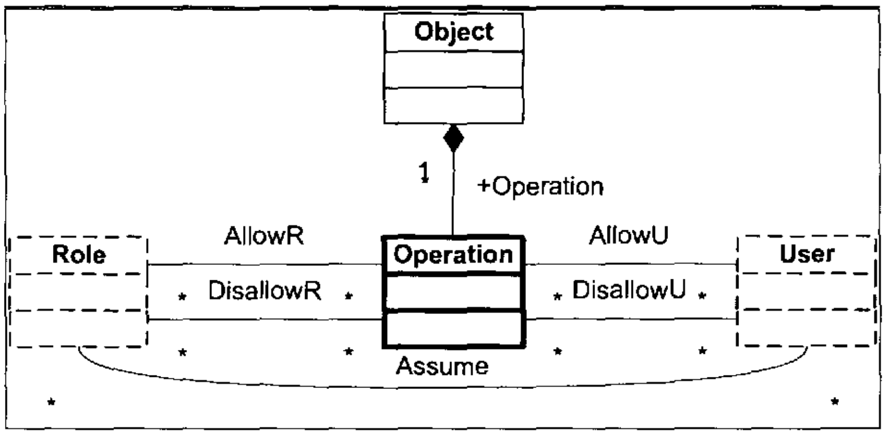


Figure 5.5. The Access Control Model

5.3.1.2 Static Separation of Duty Principles

Static SOD principles prevent subjects (role or user) from gaining permissions to execute conflicting operations. There are many kinds of static SOD policies and they are listed below:

Mutually exclusive roles: A user shall not assume two conflicting roles. For example, a user must not assume both the *Purchasing Officer* and the *Accounts Payable Manager* roles. This policy can be ensured if no user is enrolled in two mutually exclusive roles, say Role_A and Role_B and can be specified in OCL as follows:

```
(Role→ select(name= "RoleA")).user→
               intersection(Role→select(name="RoleB").user)→size=0
```

Business Task: A user must not execute a specified business task that comprises a set of operations. For example, user *U* must not be authorized to perform the *Record, Verify and Authorize* use cases on the same object and this can be specified as follows:

```
User.AllowU→
 select(Operation=Operation1 OR Operation=Operationn) → size<n
```

Where *n* is the number of operations to perform a critical task.

Mutually exclusive operations: Mutually exclusive operations must not be included in one role, i.e., writing and signing a check must not be allowed to the *Manager* role.

```
OperationA.AllowR→intersection(OperationB.AllowR)→size=0
```

5.3.1.3 Dynamic Separation of Duty Principles

Dynamic separation of duty (DSOD) allows user to assume two conflicting roles, but not to use permissions assigned to both roles on the same object. There are several types of this policy discussed in [SZ97], of which I will show some. One DSOD constraint is to restrict the user from performing the *Record, Verify and Authorize* use cases on the same object. In order to specify this policy, a history of already granted authorizations must exist. For this purpose, I added a formal syntactic object *History_Log* to maintain a Table of (user, role, operation, object and time).

Dynamic Separation of Duty: This version says that a user cannot perform more than n operation on the same object, stated as a precondition of an operation:

History_Log→ select (User= CurrentUser AND
  (Operation=Operation$_1$ OR Operation=Operation$_2$ OR
  Operation=Operation$_{n-1}$) AND Object= CurrentObject)→ size<n-1

5.3.1.4 Other Access Control Constraints

Role prerequisites: A user must be enrolled in a particular role before assuming another role. This can be stated as a postcondition of the role assignment where *Role$_B$* is the prerequisite role as follows:

User.Role→ includes(Role$_A$) implies User.Role→ includes(Role$_B$)

Permission Prerequisites: A role must be authorized to execute a particular operation before granting that role with another operation. This constraint can be specified as a postcondition of permission assignment where *Operation$_B$* is the prerequisite permission. For example, the *Supervisor* role cannot assume the *authorizes a payment* role unless this role already has a permission to read the invoice's data.

Role.Operation→ includes(Operation$_A$) implies Role.Operation→
  includes(Operation$_B$)

Cardinality Constraints: This constraint specifies a maximum and/or a minimum number of operations that can be executed by a user or a role. This policy may be applied to the number of users for each role or to the number of permissions for a specific role. For example, the *Supervisor* role must have at most one user. This constraint can be specified as follows:

```
(Role→select(name=RoleName)).User→ size <sign> n
```
where <sign> is one of the following (<,>,<=,>=,<>,=) and n is the limit.
```
(Role→select (name=RoleName)).Operation→ size <sign> n
```
where <sign> is one of the following (<,>,<=,>=,<>,=) and n is the limit.

5.3.2 Developing an Access Control Table and Applying Propagation, Conflict Resolution and Decision Policies on All Use Cases

Use cases and their *access control schemas* may over or under specify authorizations, thereby resulting in inconsistency or incompleteness. To analyze access control policies in the early phases of small-scale software systems, I present the following steps. First, present the access control of the use cases using the *access control table.* Second, apply the propagation policy. Third, apply the conflict resolution policy. Fourth, apply the decision meta-policy on the *access control tables* in order to resolve inconsistencies and incompleteness.

Access control tables show static access control policies rather than dynamic access control policies because the dynamic policies can be validated only during the execution time; thus, all that I can do during the analysis phases regarding the dynamic policies is to write the policies in unified constraints as I discussed them in Section 5.3.1.

The access control table is a matrix where the *(i,j)*th entry in the Table is (✓) /(×) symbolizing if role *i* is permitted/prohibited in invoking use cases *j*. Table 5.1 shows the access control table for the use case of the running example.

Table 5.1. Access Control Table of The Running Example.

Role\Use case	Record Invoice Arrival	Verify Invoice Validity	Authorize Payment	Write a Check
Clerk	✓			✓
Purchasing Officer	✓	✓		
Supervisor			✓	×

Next, I will show how propagation, conflict resolution decision policies can be applied to make an access control table complete.

5.3.2.1 Propagation Policies

Most systems use some kind of hierarchy, e.g., roles, subjects or objects. Hierarchies induce inheritance and, if the intended application has some form of applicable inheritance principles, they can be used to complete the access control table. These inheritance principles are referred to as propagation principles in the access control literature [JSSS01]. Several examples of propagation policies from [JSSS01] are as follows:

- **No propagation:** Permission shall not propagate throughout the hierarchy.

- **No overriding:** Permissions propagate through the hierarchy and other contradicting authorizations. Therefore, if an entity's authorization is positive and its ancestor's authorization is negative, then both authorizations apply to the entity.
- **Most specific overrides:** If an entity has an explicitly granted permission, then that overrides any inherited permission. However, if the entity does not have an explicitly authorization, then its immediate ancestor's authorization will apply to the entity.
- **Path overrides:** An entity's authorization overrides any inherited conflicting permissions for that node and for all unauthorized sub-nodes on the same path only.

It is up to the requirement engineer to choose the policy. I refer the reader to [JSSS01] for an example of how each policy is applied. Table 5.2 shows the access control table of the running example after applying *the most specific overrides* policy where ✓✓ and ×× denotes derived positive and negative permissions, respectively. For example, because the *Purchasing Officer* is a specialized role of *Clerk*, all permissions of the *Clerk* role should be propagated to the *Purchasing Officer*, such as the permission to *write a check*. However, if there is an opposite explicit permission for the role that the authorization will propagate to - such as, the *Supervisor* role for the *write a check* use case - then the propagation policy enforces the most specific permission.

Table 5.2. Access Control Table After Applying Propagation and Decision Policies.

Role \ Use case	Record Invoice Arrival	Verify Invoice Validity	Authorize Payment	Write a Check
Clerk	✓			✓
Purchasing Officer	✓	✓		✓✓
Supervisor	✓✓	✓✓	✓	x

5.3.2.2 Conflict Resolution Policies

Propagation may generate access control conflicts. Thus, conflict resolution policy resolves conflicting permissions, of which I will show some variants [JSSS01] as follows:

- **Denials take precedence:** When positive and negative permissions are granted, negatives override positives.
- **Permissions take precedence:** When positive and negative permissions are granted, positives override the negatives.
- **Nothing takes precedence:** When positive and negative authorizations apply for an object, neither positive nor negative apply, leaving the specification incomplete.

Table 5.3. Access Control Table After Applying Propagation and Decision Policies.

Role \ Use case	Record Invoice Arrival	Verify Invoice Validity	Authorize Payment	Write a Check
Clerk	✓	×××	×××	✓
Purchasing Officer	✓	✓	×××	✓✓
Supervisor	✓✓	✓✓	✓	×

5.3.2.3 Decision Policies

Decision policies complete incomplete authorization. For example, the *authorize a payment* use case in Table 5.2 does not have an authorization (positive or negative) for the *Clerk* role. Does this mean that the *Clerk* can or cannot execute the use case? The following are some decision policies that can be used to answer this question:

- **Closed Policy**: Every access is denied, unless there is a positive permission.
- **Open Policy**: Every access is granted, unless there is a negative permission.

The result of applying the closed policies for the running example is shown in Table 5.3 where all undecided permission is marked ××× indicating prohibitions.

5.3.3 Propagating Authorizations to Operations

Previous sections showed how complete and consistent permissions could be assigned for actors to invoke use cases. This section shows how they can be propagated to abstract operations used in the *access control schemas*. Thus, permission (for subject, operation) needs to be derived from those assigned to (actor, use case) pairs. One of the issues is that an abstract operation may be part of more than one use case and, therefore, could inherit multiple permissions per user. Therefore, although incompleteness does not arise, inconsistency may arise due to multiple inheritances of permissions. I illustrate this issue with the running example. Table 5.4 shows the operations of each use case.

Based on identified operations, the *access control table* specified for actors to use cases can be used to derive permissions for abstract operations. I refer to this Table as the *access control table for operations*. Thus, the access control table for operations consists of permissions of each operation for every actor. Table 5.5, shows the access control table of all operations of the use cases in the running example. Inconsistency may happen in this stage where an operation may belong to two use cases and an actor may have inconsistent authorization for the same operation due to authorization inheritance. For example, the *Read* operations authorization for the *Supervisor* role inherits positive permissions from the *Authorize payment* use case, and negative permissions from the *Write a check* use case. Thus, the *Supervisor* role has an inconsistent authorization for that operation.

Table 5.4. Identified Operations.

Use Case	Record Invoice Arrival	Verify Invoice Validity	Authorize Payment	Write a Check
Operations	Invoice :: **Read**	Invoice :: **Read**	Invoice :: **Read**	Invoice :: **Read**
	Invoice :: **Record**	Agreement :: **Read**	Invoice :: **Authorize**	Check :: **Write**
		invoice :: **Write prices**		
		invoice :: **Verify**		

Table 5.5. Access Control Table for Operations.

Role\Operation	Read:: Invoice	Record:: Invoice	Read:: Agreement	WritePrice:: Invoice	Verify:: Invoice	Authorize:: Invoice	Write ::Check
Clerk	✓	✓	×××	×××	×××	×××	✓
Purchasing Officer	✓	✓✓	✓	✓	✓	×××	✓✓
Supervisor	✓ ×	✓✓	✓✓	✓✓	✓✓	✓	×

5.3.4 Resolving Operations Authorization Conflicts

Conflict resolution in the *access control table* for the operations has to consider two issues:

1. **Invalidating Use Case level permissions:** Any resolution may result in invalidating an entry in a complete and consistent *access control table* for Use cases. For example, the *Supervisor* role in the running example has both positive

and negative authorization for the *Read* operation on the *Invoice* objects, in Table 5.5. If the conflict resolution policy enforces negative authorizations over the positive ones, the system ends up preventing the *Supervisor* role from executing the *Authorize payment* use case because one of the required operations is denied.

2. **Violating access control constraints:** Although, the step in Section 5.3.2 resolves conflicts, the end result may violate access control policy. For example, the *write a check* permission may be granted to the *Supervisor* role that has an explicit negative access control policy (according to the access control policy number 3 in Section 5.2). As a result, required access control constraints are violated. The *Access control schemas* must be complete and consistent. Visually, all cells in the access control table should have one of {✓, ×}. In addition, the following observations help in resolving conflicts at this level.

First of all, although some access control policies conceptually apply to several roles, in reality only a fewer roles may violate them (e.g., access control policies should be enforced on fewer roles). For example, suppose an access control policy states that no role can record, verify and authorize the same invoice. According to Table 5.3 only the *supervisor* role can perform all three use cases on the same invoice because the *Supervisor* is the only role with positive permissions to execute all three use cases. Thus, the integrity constraint only applies to the *Supervisor* role. This example shows how an access control policy that applies to all roles can result in one role violating it.

Second, flow constraints between use cases imply dependencies between conflict resolution strategies. For example, policy #4 in Section 5.2 states that a role cannot perform any two use cases on the same invoice. It is not efficient to enforce this integrity constraint on each use case. However, integrity constraints should be enforced before the execution of the second and third use case because, in sequential use cases only at those use cases, at least one use case would have been executed. Thus, the decisions about which use case can optimally enforce the integrity constraint of DSOD policies are not straightforward. The algorithm in Figure 5.6 chooses the optimal use case among those that have dependencies enforced between them.

Applying a policy on one use case is trivial task. However, when applying a policy on a set of use cases, a decision on the details of enforcement is essential in reducing the overhead of validating such policies. A set of use cases may depend on each other where one cannot start until the previous one has completed.

```
Int n  //Total number of entities that the policy is enforced on.
Int m  //Minimum number of entities that must not be invoked by
       //the same subject.
Int z  //Total number of other use cases not in the current tree.
Int q  //A use case, q=|m-z|.
Int i  //The level of q. Level: it comprises a set of use cases
       //that have the same order in a dependent tree.
        //Entity can be use case or operation.
If n=m then
     if there are no dependent entities trees then
          for each independent entity do
          Write the integrity constraint on the entity.
     else  //there are dependent entities trees
          if there is only one dependent entities tree then
               write the integrity constraint on the last entity
               of this tree.
          else //there are more than one dependent entity tree.
               for each independent entity do
                  Write the integrity constraint on that entity.
               for each dependent entities tree do
                    write the integrity constraint on the last
                    entity of each tree.
          End If
     End If
else // m<n
     if there are no dependent entities trees then
          for each independent entity do
               Write the integrity constraint on the entity.
     else  //there are dependent entities trees
           for each independent entity do
                    Write the integrity constraint on it.
           for each dependent entities tree do
               if m ≤ z then
                    k=1
               else
                    k=i
               End If
               write integrity constraints on use cases from the
               kth level to the highest level of the dependent
               tree.
           End loop
     End If
End If
```

Figure 5.6. An Algorithm for Enforcing Integrity Constraint of DSOD Policies

Assume there is an access control policy that must be enforced on four use cases. The policy is to prevent a user from performing any three use cases on the same object. Three of the use cases are dependent on each other sequentially, while the fourth is independent of the rest. According to the new Algorithm in Figure 5.6, this policy should be enforced as a prerequisite on the independent use case, while, for the three dependent use cases, the policy should be enforced on the second and third use cases because a user may invoke the first, then the independent, then the second use case on the same object that violates the policy. As an advantage, systems do not have to validate the policy on the first use case because I am sure that the policy will not be violated and because, after performing this use case, two more use cases must executed due to the dependency on the first use case. The Algorithm in Figure 5.6 can also apply to operations, instead of to the use case.

5.3.5 Drawing The Refined Use Case Diagram

Although use case diagrams visually represent the behavioral requirements of a proposed software system, they are not sufficient to represent existing access control policies. At best, the use case diagram shows some access control by stating the roles that actors are permitted to invoke.

Thus, having visual representations of access control policies is very much in accordance with the objectives of the UML. I refined the use cases diagram for this purpose as in Figure 5.7. The refined use case diagram represent all possible access control policies (positive, negative, explicit, implicit and integrity constraints), which provides clear visual access control policies. The refined the use case diagrams to have many desirable features as follows:

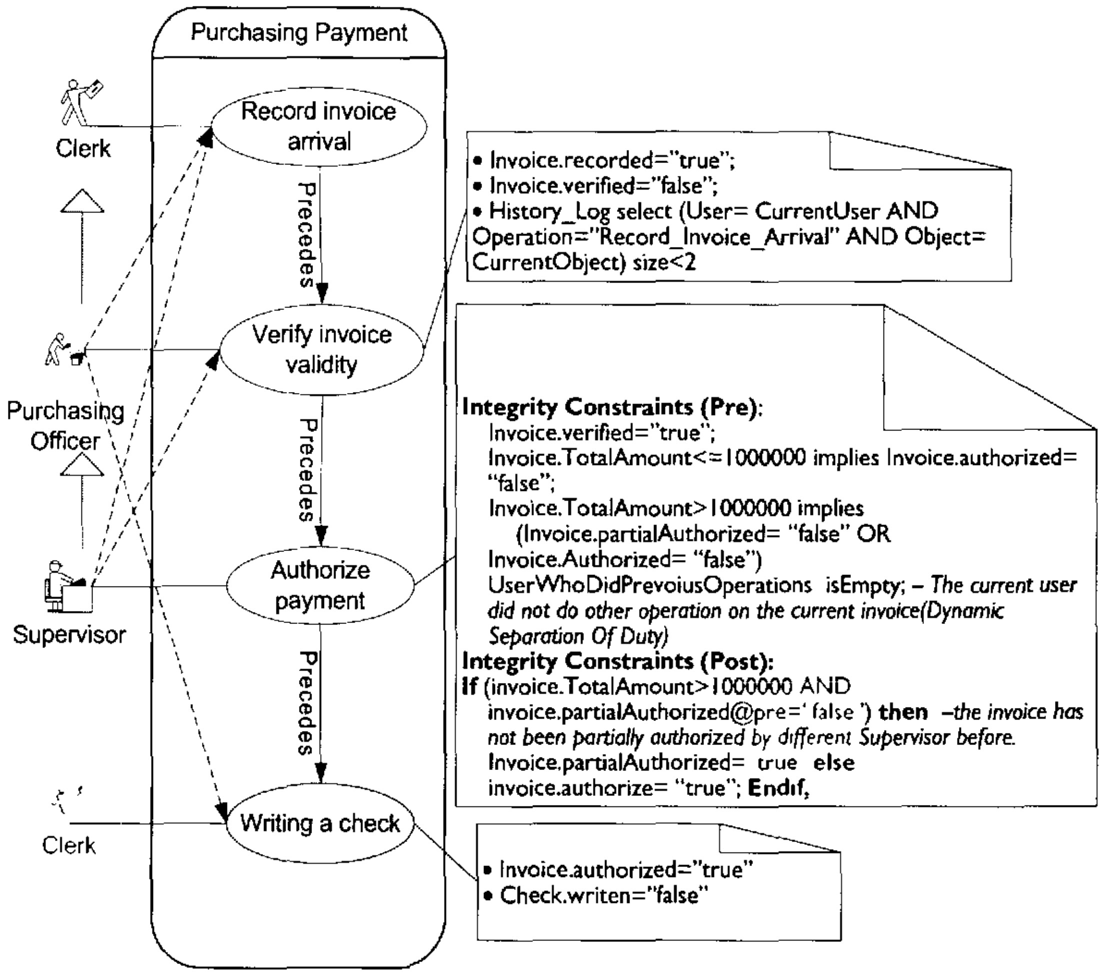
- I explicitly associate actors with all use cases that they are authorized (explicitly or implicitly) to invoke. Thus, the absence of an association between an actor and a use case is read as a prohibition.
- The new refined use case diagram adapts a relationship, which is introduced by the Open Modeling Language (OML), called *Precedes* [FHG97]. The relationship is used to specify dependencies and order of invocation among use cases.
- Use cases diagrams should be enhanced with *access control schemas*, in order to specify details of access control policies for both actors to invoke use cases and for subjects to invoke abstract operations. *Access control schema* is represented as attached constraints to each use case. Although, this may clutter the diagram, especially when integrity constraints are complex, it provides useful information about access control polices.

The access control policies in the running example in Section 5.2 are represented in the refined diagram as follow:

- Policy #1 is represented by *Precedes* relationship to show the dependence and flow of use cases.
- Policy #2 and #3 are represented by showing all explicit and implicit authorizations between actors and use cases where the absence of link between actor and use case means a negative authorization. Note that the associations of the original use case diagram do not represent all possible positive authorizations.

The absence of authorizations between actor and use case do not mean negative authorizations.

- Policy #4 and #5 are shown as constraint notes attached to use cases that are derived from the integrity constraint clause of the *access control schema.*



```
• Invoice.recorded="true";
• Invoice.verified="false";
• History_Log select (User= CurrentUser AND
Operation="Record_Invoice_Arrival" AND Object=
CurrentObject) size<2
```

```
Integrity Constraints (Pre):
   Invoice.verified="true";
   Invoice.TotalAmount<=1000000 implies Invoice.authorized=
   "false";
   Invoice.TotalAmount>1000000 implies
      (Invoice.partialAuthorized= "false" OR
   Invoice.Authorized= "false")
   UserWhoDidPrevoiusOperations  isEmpty; – The current user
   did not do other operation on the current invoice(Dynamic
   Separation Of Duty)
Integrity Constraints (Post):
If (invoice.TotalAmount>1000000 AND
   invoice.partialAuthorized@pre=' false ') then  –the invoice has
   not been partially authorized by different Supervisor before.
   Invoice.partialAuthorized=  true  else
   invoice.authorize= "true"; Endif,
```

```
• Invoice.authorized="true"
• Check.writen="false"
```

Figure 5.7. The Refined Use Case Diagram

5.4 Conclusion

I designed artifacts and a methodology to use them in specifying access control policies during the requirement specification and analysis phases. My use case extension and enhancement specifies access control policies in a formal and precise manner, and is capable of deriving access permissions along hierarchies. In addition, I presented meta-policies, algorithms and methodologies to resolve conflicting permissions before proceeding to the design phase. I introduced the access control table as visual representation of access permissions and extend the use case diagram to completely specify them.

Chapter 6

AUTHUML

The previous chapter extended the use case model to specify access control policies. Also, it presented a methodology to analyze access control requirements for small scale software systems. In this chapter, I present AuthUML, a framework to formally verify the compliance of access control requirements with the access control policies during the specification phase of UML based software lifecycle. AuthUML differentiates from the methodology in the previous chapters by focusing on large-scale software systems. AuthUML concentrates more on the verifying rather than representation of access control policies. AuthUML is based on Prolog stratified logic programming rules. Thus, tools can be implemented to automate the verifications of requirements for large software systems.

AuthUML is based on FAF [JSSS01] of Jajodia et al., and is an attempt to advance its application to the requirements specification phase of the software development life cycle. Therefore, AuthUML is a customized version of FAF that is to be used in requirements engineering. Therefore, AuthUML uses similar components of FAF with some modification in the language and the process to suit the Use Case model used in UML. Because FAF specifies authorization modules in computing systems, FAF is invoked per each authorization request. Contrastingly, AuthUML is to be used by requirements engineers to avoid conflicts and incompleteness of accesses. Therefore, while FAF is used frequently to process each access control request during execution, AuthUML is to be used less frequently during the requirements engineering phase to analyze the access control requirements.

AuthUML uses Prolog style stratified logic programming rules to specify policies that ensure desirable properties of requirements. Because requirements are specified using actors invoking Use Cases, AuthUML uses predicates to specify which actors (subjects) are permitted or prohibited from invoking any given Use Case. Moreover, it uses rules to specify the policies that need to be enforced on the system.

The remainder of this chapter is organized as follows. Section 6.1 presents the process of applying AuthUML. Section 6.2 describes the syntax and semantics of AuthUML. Section 6.3, 1.4 and 1.5 describe the first, second and third major phases of AuthUML respectively.

6.1 AuthUML Process

AuthUML consists of three main phases where each consists of several steps. As shown in Figure 6.1, AuthUML takes authorizations from requirements specifications and then analyzes them to produce complete, consistent and conflict-free authorizations.

The first phase starts with a set of access control requirements, where they are transformed into a unified representation in the form of access predicates. Also, the first phase ensures that all specified accesses are consistent and conflict-free. The second phase ensures that all accesses specified for Use Cases are consistent, complete and conflict-free, without considering the operations used to describe their functionality. During the third phase, AuthUML analyzes the access control requirements on operations. All three phases consists of rules that are customizable to reflect policies used by the security requirements engineers.

From the access control requirements that are provided in the form of AuthUML predicates, the second phase propagates accesses based on subject and/or object hierarchies. Any inconsistencies that may occur due to such propagated accesses are resolved using conflict resolution rules. After this, all accesses that are explicit (i.e., given directly in requirement) or implicit (derived) are consistent, but may not be complete, i.e., not all accesses for all subjects and Use Cases may be specified. Therefore, using predefined rules and policies (i.e., closed or open policies) in the next step (5 in Figure 6.1) completes them. Therefore, accesses specified before step 6 are directly obtained from requirements, propagated due to hierarchy or consequences of applying decision policies. Thus, it is necessary to validate the consistency of the finalized accesses against the original requirements and to check for conflicts between them. If AuthUML finds any inconsistency or conflict among accesses at this step, it will notify the requirement engineer in order to fix it and run the analysis again.

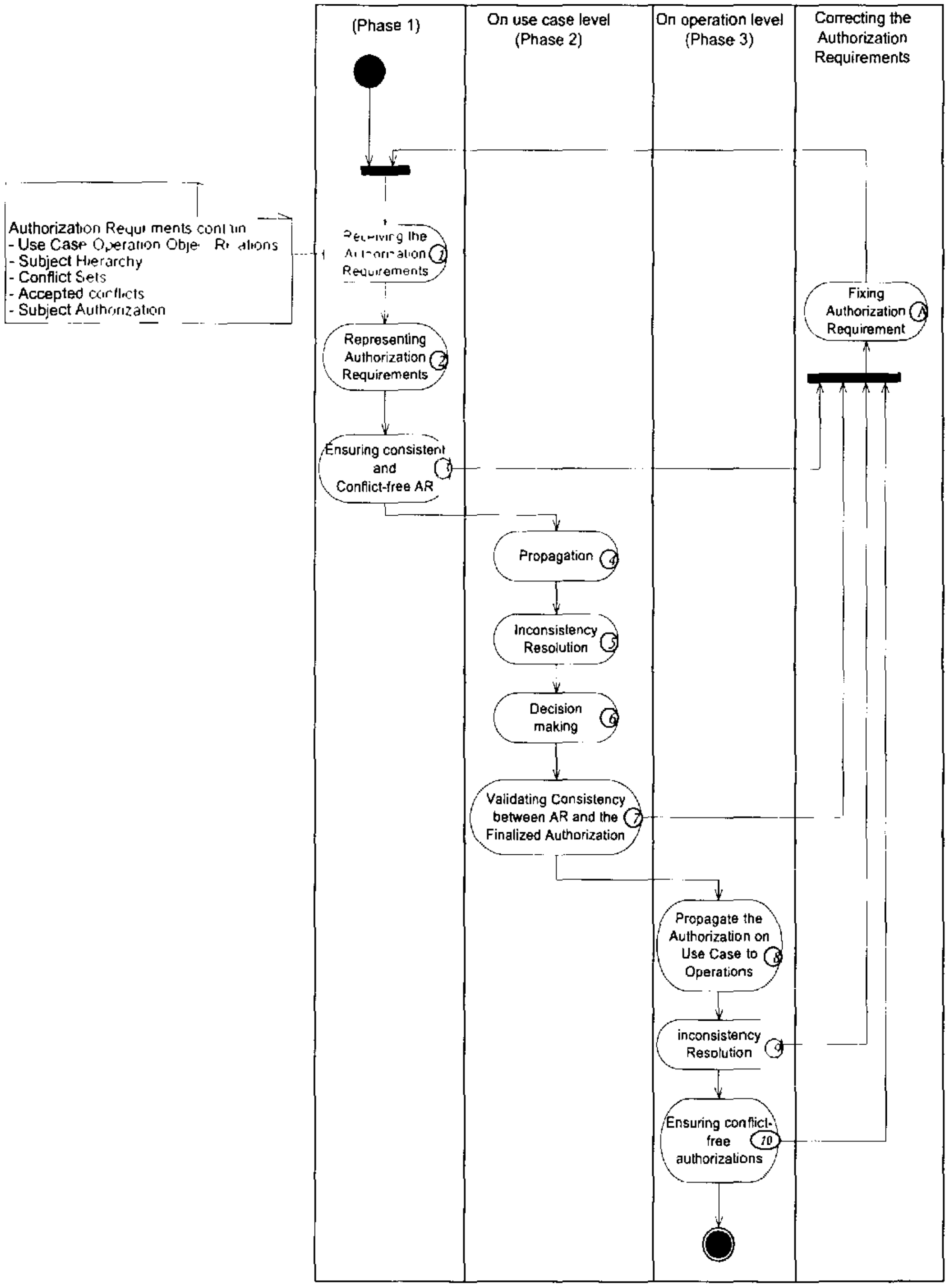

(Phase 1)
On use case level (Phase 2)
On operation level (Phase 3)
Correcting the Authorization Requirements
Authorization Requirements contain
- Use Case Operation Object Relations
- Subject Hierarchy
- Conflict Sets
- Accepted conflicts
- Subject Authorization
Receiving the Authorization Requirements 1
Representing Authorization Requirements 2
Ensuring consistent and Conflict-free AR 3
Propagation 4
Inconsistency Resolution 5
Decision making 6
Validating Consistency between AR and the Finalized Authorization 7
Propagate the Authorization on Use Case to Operations 8
inconsistency Resolution 9
Ensuring conflict-free authorizations 10
Fixing Authorization Requirement A

Figure 6.1. AuthUML Architecture

The third phase of AuthUML applies the same process to operations used to describe Use Cases. This phase does not have a decision step, as in the second phase, because each Use Case propagates its accesses to all its operations. As a result, accesses specified during this phase are complete. In addition, access specifications of operations at the end of this phase are consistent because the inconsistency resolution step in the operation level will attempt and resolve all inconsistencies. However, if it cannot do so, the process will stop and notify the requirement engineer about the inconsistency to be fixed by manual intervention. Up to this step, accesses are consistent and complete, but may not be free of application specific conflict. Thus, the purpose of the last step of this phase is to detect those conflicts.

There is a difference between the access specifications fed into AuthUML and those that come out of it, i.e., finalized access specifications are consistent, complete and free of application specific conflicts. This outcome is the main advantage of my work. Thus, AuthUML focuses on access control requirements as early as possible to avoid any foreseeable problems before proceeding to other phases of the development life cycle. As the development process proceeds through its life cycle, changes of the access control requirement may occur. For example, the Use Cases may be changed to invoke different operations, or refined/new operations may be added. Consequently, already accepted accesses may need to be reanalyzed. Therefore, it is necessary to go back and run the AuthUML again to preserve the consistency, and to ensure complete and conflict-free accesses. Thus, my framework is flexible enough that it allows changes in the access control specifications.

The architecture of AuthUML differs from the architecture of FAF in two aspects. First, AuthUML analyzes accesses in two levels, Use Cases and operations in order to scrutinize accesses in course-grain and fine-grain levels, respectively. Second, steps 2, 6 and 9 are introduced in AuthUML to detect inconsistencies and conflicts between different levels of accesses that are absent in FAF. Moreover, AuthUML receives a bulk of access control requirements but not just one access request at a time. Thus, as I will show later, AuthUML produces accesses only if there are sufficient rules to resolve all application level conflicts.

6.2 AuthUML Syntax and Semantics

6.2.1 Individuals and Terms of AuthUML

The individuals of AuthUML are the Use Cases, operations, objects and subjects. Use Cases specify actors and their intended usage of the envisioned system. Such usage - usually, but not always - is specified in terms of the interactions between the actors and the system, thereby specifying the behavioral requirements of the proposed software. Each Use Case consists of a set of operations that are used to describe the Use Case. Each operation *operates* on an object, and operations are the only way to query or manipulate objects. Subjects are permitted to invoke a set of Use Cases and thereby all operations describing that Use Cases. I use subjects as actors in UML or role in Role-based Access control (RBAC) [SCFY96, SS00]. I denote UC, OP, OBJ, and S as set of Use Cases, operations, objects and subjects respectively. An access permission can be permitted or prohibited, that is, modeled as a positive or a negative action, respectively. AuthUML syntax is built from constants and variables that belong to four individual sorts. Namely, signed Use Cases, signed operations, (unsigned) objects and (unsigned) subjects. They are represented respectively as $\pm uc$, $\pm op$, obj, and s, where variables are represented as $\pm X_{uc}$, $\pm X_{op}$, X_{obj}, and X_s.

6.2.2 Predicates of AuthUML

I use FAF predicates with some customizations and some new predicates to model requirements as follows:

Following predicates are used to model structural relationships and called rel-predicates.

1. A binary predicate UC_OP(Xuc,Xop) means operation Xop is invoked in Use Case Xuc.
2. A binary predicate OP_OBJ(Xop,Xobj) means operation Xop belongs to object Xobj.
3. A binary predicate before(Xop,X'op) means that Xop must be invoked before X'op.
4. A ternary predicate inUCbefore(Xuc,Xop,X'op) means Use Case Xuc invokes Xop before X'op.

Following predicates are used to model hierarchies and called hie-predicates.

1. A binary predicate in(Xs,X's), means Xs is below X's in the subject hierarchy.
2. A binary predicate dirin(Xs,X's) mean Xs is directly below X's in the subject hierarchy.

Following predicates are used to model conflicts and called con- predicates.

1. A binary predicate conflictingSubject(X_s,X'_s) means subject X_s and X'_s are in conflict with each other.
2. A binary predicate conflictingSubject$_{Content}$(X_s,X'_s,Y) means that both subject X_s and X'_s must not invoke Y, where Y can be a use case or an operation. This predicates is more specific to stating the conflict for a particular content, e.g., the operations.
3. A binary predicate conflictingUC(X_{uc},X'_{uc}) means that Use Cases X_{uc} and X'_{uc} are in conflict with each other.
4. A binary predicate conflictingOP(X_{op},X'_{op}) means operations X_{op} and X'_{op} are in conflict with each other.
5. A ternary predicate ignore(X,Y,Y') represents an explicit instruction by the requirements engineer to ignore a conflict among X,Y and Y' where X, Y and Y' are either subjects, operations or Use Cases.

The following predicates are used in the first phase of AuthUML to authorize, detect assignment conflict or detect inconsistency in the access control requirements:

1. opInConUC(X_{op},X_{uc},X'_{uc}), means X_{op} is an operation in two conflicting Use Cases X_{uc} and X'_{uc}, and conOpInUC(X_{op},X'_{op},X_{uc}) means that X_{op} and X'_{op} are two

conflicting operations in Use Case X_{uc}, and flowConInUC(X_{UC},X_{OP},X'_{OP}) means that X_{OP} and X'_{OP} are invoked in a way that violate execution order.

2. A binary predict cando$_{UC}$, where cando$_{UC}$ (X_s,$\pm X_{uc}$) means subject X_s can or cannot invoke the Use Case X_{uc} depending on the *sign* of X_{uc}, positive (+) or nagative(−).
3. A binary predicate alertReq(X_s,X_{uc}) to inform the requirements engineer that there is either an inconsistency (between access control requirements) or a conflict (between subjects or Use Cases) on the access of X_s on X_{uc}.

Following predicates are used in the second phase of AuthUML to authorize, detect conflicts and inconsistencies at the Use Case level:

1. A ternary predicate overUC(X_s,X'_s,$\pm X_{uc}$) meaning X_s's permission to invoke $\pm X_{uc}$ overrides that of X'_s.
2. A binary predicate dercando$_{UC}$ with the same argument as cando$_{UC}$. dercando$_{UC}$(X_s,$\pm X_{uc}$) is a permission derived using modus ponens and stratified negation [ABW88].
3. A binary predicate do$_{UC}$, where do$_{UC}$(X_s,$\pm X_{uc}$) is the final permission/prohibition for subject X_s to invoke Use Case X_{uc} depending on if the *sign* of X_{uc} is + or −.
4. A binary predicate alert$_{UC}$(X_s,X_{uc}) to inform the requirements engineer that there is either an inconsistency (between the access control requirement and the final outcome of this phase) or conflict (between subjects or use cases) on the accesses that involve X_s and X_{uc}

Following predicates are used in the third phase of AuthUML to authorize, detect conflicts and inconsistencies at the operation level:

1. A binary predicate $\mathrm{dercando}_{OP}(X_s, \pm X_{op})$ is similar to $\mathrm{dercando}_{UC}$ except the second argument is an operation instead of a Use Cases.
2. A binary predicate $\mathrm{do}_{OP}(X_s, \pm X_{op})$ is similar to do_{UC} , but the second argument is an operation.
3. $\mathrm{cannotReslove}(X_s, X_{uc}, X'_{uc}, X_{op})$ is a 4-ary predicate representing an inconsistency that can not be resolved at the operation level with the given rules.
4. A binary predicate $\mathrm{alert}_{OP}(X_s, X_{op})$ informs the requirements engineer that there is a conflict between subjects or operations on the authorization that involve X_s and X_{op}.

Assumptions

- The *subject* I used refers to a role (as in RBAC) or an actor (in UML) and not to an end user of a software system. The role is a named set of permissions and users may assume a role in order to obtain all of its permissions.
- Every Use Case must have at least one operation (i.e., $\forall x \in UC\ \exists y \in OP\ UC\_OP(x,y)$) and every operation must belong to one and only one object (i.e., $\forall x \in OP\ \exists! y \in OBJ\ OP\_OBJ(x,y)$).
- Each positive access of a Use Case to a subject means that all operations of that use case are also positively authorized to the same subject. This is consistent with [Sen02]. Conversely, a prohibited Use Case to a subject must have at least one prohibited operation to that subject.

As already stated, cando represents an access permission obtained from requirements and dercando represents an access derived using (to be described shortly) rules. Both cando and dercando do not represent a final decision, but only an intermediate result. For example, although $\text{cando}_{UC}(X_s,+X_{uc})$ is obtained from requirements does not mean that subject X_s will be allowed to finally execute Use Case X_{uc}. The reason being that propagation, conflict resolution and decision policies may change the authorization expressed in $\text{cando}_{UC}(X_s,+X_{uc})$. However, $\text{do}_{UC}(X_s,+X_{uc})$ if derived represents the final authorization decision.

6.2.3 Rule of AuthUML

An AuthUML rule is of the form $L \leftarrow L_1, \ldots L_n$ where L is a positive literal and $L_1, \ldots L_n$ are literals satisfying the conditions stated in Table 4.1.

An Example:

$$\text{cando}_{UC}(\text{supervisor}, +\text{"}\textit{authorize payment}\text{"}) \leftarrow \tag{1}$$

$$\text{dercando}_{UC}(X_s,+X_{uc}) \leftarrow \text{cando}_{UC}(X'_s,+X_{uc}),\ \text{in}(X_s,X'_s) \tag{2}$$

$$\text{do}_{UC}(X_s,+X_{uc}) \leftarrow \text{cando}_{UC}(X_s,+X_{uc}),\ \text{cando}_{UC}(X_s,-X_{uc}) \tag{3}$$

Rule 1 says that supervisor can access Use Case "authorize payment." Rule 2 specifies the inheritance of authorizations in the subject hierarchy. Rule 3 expresses the permissions take precedence policy of resolving conflicts.

Table 6.1. Rules Defining Predicate

Phase	Stratum	Predicate	Rules defining the predicate
Phase 1	0	rel-predicates	base relations.
		hic-predicates	base relations.
		con-predicates	base relations.
		ignore(X,Y,Y')	Explicit instructions to ignore the X,Y conflict.
	1	opInConUC(X_{op},X_{UC},X'_{UC})	body may contain hie, ignore, rel predicates.
		conOpInUC(X_{op},X'_{op},X_{UC})	
		flowConInUC(X_{UC},X_{op},X'_{op})	
	2	cando$_{UC}$(X_s,$\pm X_{uc}$)	body may contain hie-, con- and rel-predicates
	3	alert$_{Req}$(X_s,X_{uc})	body may contain literal from strata 0 to 2
Phase 2	4	over$_{UC}$(X_s, X'_s ,$\pm X_{uc}$)	body may contain literals from strata 0 to 3
	5	dercando$_{UC}$(X_s $\pm X_{uc}$)	body may contain predicates from strata 0 to 4 Occurrences of dercando$_{UC}$ must be positive.
	6	do$_{UC}$(X_s,$+X_{uc}$)	body may contain predicates from strata 0 to 5
	7	do$_{UC}$(X_s $-X_{uc}$)	body contains one literal ¬do$_{UC}$(X_s $+X_{uc}$)
	8	alertUC (X_s, X_{uc})	body may contain literal from strata 0 to 7
Phase 3	9	dercando$_{OP}$(X_s,$\pm X_{op}$)	body may contain predicates from strata 0 to 7. Occurrences of dercando$_{OP}$ must be positive
	10	do$_{OP}$(X_s, $+X_{op}$)	body may contain predicates from strata 0 to 9
	11	do$_{OP}$(X_s, $-X_{op}$)	body contains one literal ¬do$_{OP}$(X_s, $+X_{op}$)
	12	cannotReslove(X_s, X_{uc} X'_{uc} X_{op})	body may contain literal from strata 0 to 11
	13	alertOP(X_s,X_{OP})	body may contain literal from strata 0 to 11

6.2.4 AuthUML Semantics

Table 4.1 shows the stratification of rules used in AuthUML. Rules constructed according to these specifications forms a local stratification. Accordingly, any such rule based form has unique stable model and that stable model is also a well-founded model, ala Gelfond and Lifschitz [GL88]. As done in FAF, I can materialize AuthUML rules also, thereby making the AuthUML inference engine efficient.

6.3 Phase I: Analyzing Information in Requirements Documents

This section goes though steps 1 and 2 of AuthUML and shows how the AuthUML processes the access control requirements.

6.3.1 Representing Authorizations

Following [JSSS01], I assume that requirement engineers already specify access control requirements and that it is not in the scope of this chapter to go further on that subject. Authorization requirements consist of:

1. Permissions for the subject to invoke the Use Cases
2. The Subject hierarchy.
3. Structural relationships (Use Case - Operation - Object Relations).
4. Conflicting subjects, Use Cases and operations sets.
5. Conflicts of interest.

All of the above must be written in this step in the form of AuthUML rules in order to be used during subsequent steps. They are represented as follow:

1. At this step, access permissions are written in the form of cando_{uc} rules representing explicit authorization obtained from the requirement specification. Rule 4 and 5 are examples:

$\text{cando}_{UC}(\text{clerk}, +\text{recordInvoiceArrival}) \leftarrow$ (4)

$\text{cando}_{UC}(\text{supervisor}, -\text{writeCheck}) \leftarrow$ (5)

Rule (4) permits the clerk to invoke the "recordInvoiceArrival" Use Case. Rule (5) prohibits the supervisor to invoke the "writeCheck" Use Case.

2. Subject hierarchy is represented using the in predicate to indicate which subject inherits what. For example, in(purchasingOfficer,clerk) means that the purchasing officer is a specialized subject of clerk that inherits all its permissions.

3. Structural relationships represent the relations between use case and its operations, operations and its object, and the flow between operations in a use case. $\text{UC\_OP}(X_{uc},X_{op})$ says that X_{op} is an operations invoked in the Use Case X_{uc}. $\text{OP\_OBJ}(X_{op},X_{obj})$ says that operation X_{op}, belongs to object X_{obj}. In addition, $\text{before}(X_{op},X'_{op})$ means that X_{op} must be executed before X'_{op} is executed and $\text{inUCbefore}(X_{uc},X_{op},X'_{op})$ means that Use Case X_{uc} calls for executing X_{op}, before X'_{op}.

4. Application definable conflicts occurring among subject, Use Case and operations are represented respectively by $\text{coflictingSubjects}(X_s,X'_s)$, $\text{conflicting}_{UC}(X_{uc},X'_{uc})$ and $\text{coflicting}_{OP}(X_{op},X'_{op})$.

5. Requirement engineers may decide to accept some conflicts as in [GH91, GH92, NE01]. AuthUML uses ignore(X,Y,Y') to accept a conflict between Y and Y'. The main goal of the ignore predicate is to only allow specified conflicts, but not others between access.

6.3.2 Ensuring Consistent And Conflict-Free Access Control Specifications

Access control requirements may specify inconsistencies where one requirement permits and another requirement denies the same permission. In addition, two conflicting subjects may be permitted to invoke the same Use Case or operation, or a subject may be permitted to invoke two conflicting Use Cases or operations. Latter kinds of permissions may violate the *Separation of Duty* principle [CW87].

In small systems, discovering conflicts can be easy, because of the small number of entities and engineers writing those requirements. However, detecting conflicts and inconsistencies between access control requirements in large system is more problematic. Therefore, AuthUML can specify rules that detect inconsistencies between the requirements that are specified by many security engineers. Detecting inconsistencies and conflicts at this stage prevent them from spreading to the following stages of the life cycle. This step of AuthUML takes access control requirements in the form of cando rules and automatically applies inconsistency and conflict detection rules to identify their existence, as follow:

$$\text{alert}_{Req}(X_s,X_{uc}) \leftarrow \text{cando}_{UC}(X_s,+X_{uc}),\ \text{cando}_{UC}(X_s,-X_{uc}) \tag{6}$$

$$\begin{aligned}\text{alert}_{UC}(X_s,X_{uc}) \leftarrow\ & \text{cando}_{UC}(X_s,+X_{uc}),\ \text{cando}_{UC}(X_s,+X'_{uc}),\\ & \text{conflicting}_{UC}(X_{uc},X'_{uc}),\ \neg\text{ignore}(X_s,X_{uc},X'_{uc})\end{aligned} \tag{7}$$

$$\begin{aligned}\text{alert}_{UC}(X_s,X_{uc}) \leftarrow\ & \text{cando}_{UC}(X_s,+X_{uc}),\ \text{cando}_{UC}(X'_s,+X_{uc}),\\ & \text{conflictingSubject}_{Content}(X_s,X'_s),\ \neg\text{ignore}(X_{uc},X_s,X'_s)\end{aligned} \tag{8}$$

$$\text{opInConUC}(X_{op},X_{uc},X'_{uc}) \leftarrow \text{UC\_OP}(X_{uc},X_{op}), \text{UC\_OP}(X'_{uc},X_{op}), \text{conflicting}_{UC}(X_{uc},X'_{uc}), \neg\text{ignore}(X_{op},X_{uc},X'_{uc}) \quad (9)$$

$$\text{conOpInUC}(X_{UC},X_{op},X'_{op}) \leftarrow \text{UC\_OP}(X_{uc},X_{op}), \text{UC\_OP}(X_{uc},X'_{op}), \text{conflicting}_{OP}(X_{op},X'_{op}),\neg\text{ignore}(uc,X_{op},X'_{op}) \quad (10)$$

$$\text{flowConInUC}(X_{uc},X_{op},X'_{op}) \leftarrow \text{UC\_OP}(X_{uc},X_{op}), \text{UC\_OP}(X_{uc},X'_{op}), \text{before}(X_{op},X'_{op}), X_{op}\neq X'_{op}, \neg\text{inUCbefore}(X_{uc},X_{op}, X'_{op}) \quad (11)$$

Rule 6 says that, if there are two requirements where one grants and the other denies the invocation of the same Use Case to the same subject, an alert message will be raised to the security engineer that identifies those that lead to the inconsistency. Rule 7 says that, if a subject is permitted to invoke two conflicting Use Cases that are not explicitly allowed by the ignore predicate, an alert message is triggered in order to facilitate manual intervention. Rule 8 says that, if a Use Case is permitted to be invoked by two conflicting subjects, a manual intervention need to be sought. Rule 9 and 10 are related to the conflicting assignments of operations to Use Cases. Rule 9 detects having operations in two conflicting Use Cases and rule 10 detects having two conflicting operations in the same Use Case. Rule 11 says that, if two operations used in one Use Case violates the order in which they are to be called, the first two conflicts can be ignored if the requirement engineer explicitly uses the "ignore" predicate.

Notice that detectable conflicts that appear at this step are structural in nature. That is, they are conflicts or inconsistencies independent of the permissions or prohibitions assigned to execute them.

6.4 Phase II: Applying Policies to Use Cases

The previous phase analyzes statically given, access control requirements without using any policies and produces consistent and conflict-free accesses. This phase (steps 3, 4, 5 and 6) applies policies that are specified using AuthUML rules relevant to Use Cases. Such policies may add new permissions or change existing ones.

6.4.1 Propagation Policies

Most systems use some hierarchies to benefit from inheritance. This step may generate new permissions according to chosen propagation policies. All explicit or derived permissions are transformed to the form of $\mathtt{dercando}_{\mathtt{OP}}$ rules (derived authorizations). Some examples of propagation policies are listed in [JSSS01] and represented as AuthUML rules in Table 6.2.

6.4.2 Inconsistency Resolution Policies

In complex systems with many Use Cases, permission propagation may introduce new permissions that in turn may result in new inconsistencies. Inconsistency resolution policies resolve such inconsistencies. Examples are listed in [JSSS01] and represented as AuthUML rules in Table 6.3. The rules in Table 6.3 define inconsistency resolution policies. For example, for the *denial take precedence* with an *open* policy, if there is no denial, permission is granted for such subject. However, in the case of a closed policy, the previous definition is not enough because there must be a permission in the absence of a prohibition. The last rule completes the rule base prohibiting every access that is not permitted.

Table 6.2. Rules for Enforcing Propagation Policies on Subject Hierarchy.

Propagation policy	Rules
No propagation	$dercando_{UC}(X_s, +X_{uc}) \leftarrow cando_{UC}(X_s, +X_{uc})$
	$dercando_{UC}(X_s, -X_{uc}) \leftarrow cando_{UC}(X_s, -X_{uc})$
No overriding	$dercando_{UC}(X_s, +X_{uc}) \leftarrow cando_{UC}(X'_s, +X_{uc}), in(X_s, X'_s)$
	$dercando_{UC}(X_s, -X_{uc}) \leftarrow cando_{UC}(X'_s, -X_{uc}), in(X_s, X'_s)$
Most specific overrides	$dercando_{UC}(X_s, +X_{uc}) \leftarrow cando_{UC}(X'_s, +X_{uc}), in(X_s, X'_s),$ $\neg over_{UC}(X_s, X'_s, +X_{uc})$
	$dercando_{UC}(X_s, -X_{uc}) \leftarrow cando_{UC}(X'_s, -X_{uc}), in(X_s, X'_s),$ $\neg over_{UC}(X_s, X'_s, -X_{uc})$
	$over_{UC}(X_s, X_{s'}, +X_{uc}) \leftarrow cando_{UC}(X''_s, -X_{uc}), in(X_s, X''_s), in(X''_s, X'_s),$ $s'' \neq s'$
	$over_{UC}(X_s, X_{s'}, -X_{uc}) \leftarrow cando_{UC}(X''_s, +X_{uc}), in(X_s, X''_s), in(X''_s, X'_s),$ $s'' \neq s'$
Path overrides	$dercando_{UC}(X_s, +X_{uc}) \leftarrow cando_{UC}(X_s, +X_{uc})$
	$dercando_{UC}(X_s, -X_{uc}) \leftarrow cando_{UC}(X_s, -X_{uc})$
	$dercando_{UC}(X_s, +X_{uc}) \leftarrow cando_{UC}(X'_s, +X_{uc}), \neg cando_{UC}(X_s, -X_{uc}),$ $dirin(X_s, X'_s)$
	$dercando_{UC}(X_s, -X_{uc}) \leftarrow cando_{UC}(X'_s, -X_{uc}), \neg cando_{UC}(X_s, +X_{uc}),$ $dirin(X_s, X'_s)$

Table 6.3. Rules for Enforcing Inconsistency Resolution and Decision Policies.

Inconsistency	Decision	Rules
Denial take precedence	open	$do_{UC}(X_s,+X_{uc}) \leftarrow \neg dercando_{UC}(X_s,-X_{uc})$
Denial take precedence	closed	$do_{UC}(X_s,+X_{uc}) \leftarrow dercando_{UC}(X_s,+X_{uc}),$ $\neg dercando_{UC}(X_s,-X_{uc})$
permission take precedence	open	$do_{UC}(X_s,+X_{uc}) \leftarrow dercando_{UC}(X_s,+X_{uc})$
		$do_{UC}(X_s,+X_{uc}) \leftarrow \neg dercando_{UC}(X_s,-X_{uc})$
permission take precedence	closed	$do_{UC}(X_s,+X_{uc}) \leftarrow dercando_{UC}(X_s,+X_{uc})$
Nothing take precedence	open	$do_{UC}(X_s,+X_{uc}) \leftarrow \neg dercando_{UC}(X_s,-X_{uc})$
Nothing take precedence	closed	$do_{UC}(X_s,+X_{uc}) \leftarrow dercando_{UC}(X_s,+X_{uc}),$ $\neg dercando_{UC}(X_s,-X_{uc})$
Additional closure rule		$do_{UC}(X_s,-X_{uc}) \leftarrow \neg do_{UC}(X_s,+X_{uc})$

6.4.3 Decision Policies

Decision policies complete authorizations so that every subject must have either a permission or a prohibition to execute each Use Case and operation. Following are some decision policies that have been suggested:

Closed Policy: Accesses without permissions are prohibited.

Open Policy: Accesses without prohibitions are permitted.

This is the last step that finalizes all accesses of Use Cases to subjects that are consistent with each other and complete. They are written in the form of do_{UC} rules. AuthUML like FAF ensure the completeness of access control decision by enforcing the following.

$$do_{UC}(X_s,-X_{uc}) \leftarrow \neg do_{UC}(X_s,+X_{uc}) \qquad (12)$$

6.4.4 Alerting the Requirements Engineer of Changes to Use Case Accesses

As stated, final accesses of the last step are consistent with each other, but it may have changed the original requirements. Also, there may not be sufficient rules to resolve application specific conflicts. This step uses the alert_{UC} predicate to inform the requirements engineer of such changes or problems.

$$\mathrm{alert}_{UC}(X_s, X_{uc}) \leftarrow \mathrm{cando}_{UC}(X_s, +X_{uc}),\ \mathrm{do}_{UC}(X_s, -X_{uc}) \qquad (13)$$
$$\mathrm{alert}_{UC}(X_s, X_{uc}) \leftarrow \mathrm{cando}_{UC}(X_s, -X_{uc}),\ \mathrm{do}_{UC}(X_s, +X_{uc})$$

Rule 13 says that an alert message will be raised if there is an access control requirement and a contradicting final authorization for the same subject on the same Use Case.

Once informed by AuthUML the requirements, the engineer can revisit potential problems and, hopefully, resolve them before proceeding to apply fine-grain policies that specify operation level accesses.

6.5 Phase III: Applying Policies to Operations

The previous phase produces consistent and conflict-free Use Cases. This phase (step 7, 8 and 9) analyzes operations to ensure consistent, conflict-free and complete permissions to invoke operations.

6.5.1 Propagating Permissions to Operations

This phase applies fine-grain access control policies to operations. Recall that Use Cases are described using operations and some execution order among them. Because any Use Case contains one or more operations, permission to invoke a Use Case propagates to its operations. Following rules specify such propagation policies.

$$\mathrm{dercando}_{OP}(X_s,-X_{op}) \leftarrow \mathrm{UC\_OP}(X_{uc},X_{op}),\ \mathrm{do}_{UC}(X_s,-X_{uc}) \qquad (14)$$
$$\mathrm{dercando}_{OP}(X_s,+X_{op}) \leftarrow \mathrm{UC\_OP}(X_{uc},X_{op}),\ \mathrm{do}_{UC}(X_s,+X_{uc})$$

Rule (14) says that, if an operation is part of a Use Case, the permission of the Use Case propagates to that operation.

6.5.2 Inconsistency Resolution for Operations

Because an operation can be called on behalf of more than one Use Case and, thus, can inherit permissions from more than one Use Case, applying rules such as (14) may introduce conflicts. Therefore, conflict resolution must be applied to operations. As I stated before, I assume that each positive permission of a Use Case is inherited by all its operations. Conversely, a prohibited Use Case must have at least one prohibited operation.

An operation may be called in two Use Cases with contradicting permissions for the same subject, with the result that the subject will have been granted a permission and a prohibition to execute the same operation. One policy that can resolve this contradictory situation is to retain the permission to execute the operation for the subject only if another operation belonging to the prohibited Use Case already has a prohibition for the same subject. In doing so, I preserve the assumption that, as long as there is at least one prohibition on operation for a subject in a Use Case, that Use Case has a prohibition for the same subject. Rule 15 specifies this conflict resolution policy as an AuthUML rule:

$$\begin{aligned}do_{OP}(X_s,+X_{op}) \leftarrow & dercando_{OP}(X_s,+X_{op}),\ dercando_{OP}(X_s,-X_{op}),\\ & UC\_OP(X_{uc},X_{op}),\ do_{UC}(X_s,-X_{uc}),\ UC\_OP(X_{uc},X'_{op}),\\ & dercando_{OP}(X_s,-X'_{op}), X'_{op} \neq X_{op}\end{aligned} \tag{15}$$

6.5.3 Completing Accesses for Operations

Therefore, after the application of rule 15, AuthUML ensures the following:

1. There is no operation with contradictory authorizations for the same subject.
2. For every subject, all operations of a Use Case are permitted if the Use Case is permitted.

The next two rules ensure that all permission of a subjects to invoke operations will be represented as do predicates and, therefore, either granted or denied, but not both. These rules were used in FAF also.

$$do_{OP}(X_s,+X_{op}) \leftarrow dercando_{OP}(X_s,+X_{op}), \neg dercando_{OP}(X_s,-X_{op}) \tag{16}$$

$$do_{OP}(X_s,-X_{op}) \leftarrow \neg do_{OP}(X_s,+X_{op}) \tag{17}$$

6.5.4 Alerting the Requirements Engineer of Irreconcilable Conflicts

Continuing with the example given at the end of Section 6.5.2, if there is no X'_{op} prohibiting X_s, rule 15 cannot resolve the inconsistency. Hence, AuthUML will raise a conflict message to the requirements engineer informing its inability to resolve the contradiction, as stated in rule 18.

$$\begin{aligned}\text{cannotReslove}(X_s, X_{uc}, X'_{uc}, X_{op}) \leftarrow{} & \text{dercando}_{OP}(X_s, +X_{op}), \\ & \text{dercando}_{OP}(X_s, -X_{op}),\ \neg\text{do}_{op}(X_s, +X_{op}),\ X_{uc} \neq X'_{uc} \\ & \text{UC\_OP}(X_{uc}, X_{op}),\ \text{UC\_OP}(X'_{uc}, X_{op})\end{aligned} \tag{18}$$

$$\begin{aligned}\text{alert}_{OP}(X_s, X_{op}) \leftarrow{} & \text{do}_{OP}(X_s, +X_{op}), \text{do}_{OP}(X_s, +X'_{op}), \\ & \text{conflictingOP}(X_{op}, X'_{op}),\ \neg\text{ignore}(X_{op}, X_s, X'_s)\end{aligned} \tag{19}$$

$$\begin{aligned}\text{alert}_{OP}(X_{op}, X_s) \leftarrow{} & \text{do}_{OP}(X_s, +X_{op}), \text{do}_{OP}(X'_s, +X_{op}), \\ & \text{conflictingSubject}_{Content}(X_s, X'_s, X_{op}),\ \neg\text{ignore}(X_{op}, X_s, X'_s)\end{aligned} \tag{20}$$

Rule 19 triggers an alert message if it finds a subject Xs that has an authorization to invoke two operations that conflict with each other. Rule 20 triggers an alert message if it finds two conflicting subjects that have authorizations to invoke the same operation. Both rules will not hold if the requirement engineer explicitly allows that conflict by using the ignore predicate.

At the end of phase 3, from the finalized authorization one can generate an access control list (ACL) of all positive and negative permissions of all subject to all operations.

6.6 Conclusions

AuthUML diverges form others work in this area by focusing on analyzing access control requirements at the requirement specification stage rather than modeling them with extra syntactic enrichments to UML. I have developed AuthUML, a framework that analyze access control requirements to ensure that the access control requirements are consistent, complete and conflict-free. The framework propagates access permissions on subject hierarchies and solves inconsistencies between authorizations by enforcing predefined policies that are written using the logical language of AuthUML. To assure fine-grain analysis of access control requirements, AuthUML considers access control requirements for both Use Case and its operations. This work aims toward bridging the gap between Logic programming and formal security engineering.

Chapter 7

FLOWUML

The previous chapter introduced AuthUML that analyze access control requirement. This chapter introduces FlowUML that analyzes information flow control requirements.

7.1 Introduction

The information flow control policy is an important aspect of security that restricts information flowing between objects. In order to formally analyze information flow policies during the requirements and design stages of the software life cycle, I developed *FlowUML*. It is a flexible framework using locally stratified logic programming rules to enforce user specifiable information flow policies on UML based designs. It utilizes the following steps:

1. Extracts information flows from the UML sequence diagram as predicates.
2. Derives all inherited and indirect flows.
3. Checks for compliances with specified policies.

FlowUML utilizes stated steps for pragmatic reasons. First, information exchanges between objects are drawn using sequence, activity or collaboration diagrams, at the early stage of development. I choose sequence diagrams for illustrative purposes, and my methodology applies to others as well. Second, at this stage, it is easier to re-draw and visualize direct and transitive information flows using these diagrams. Third, indirect flows and those that result from inheritance may not be visually obvious and, therefore, automated support in deriving their consequences could aid the visual design process. Fourth, FlowUML extracts information flow based on specified policies.

Policies in FlowUML are specified at two levels of granularity. At the coarser level, a flow is a directed arc going from its source to a sink. At the finer level, flow is qualified based on the semantics of methods, attributes passed between the caller and the callee, and the roles played by caller and callee objects in the design.

FlowUML does not assume any meta-policy, and it is flexible in using many of them. Although, there are many important contributions in flow control models and policies, FlowUML advances over others in applying a formal framework in the early phases of the software development life cycle.

The remainder of this chapter is organized as follows. Section 7.2 shows information flow specifications embedded in the UML sequence diagrams. Section 7.3 provides a running example. Section 7.4 describes the notations and assumptions. Section 7.5 explains the FlowUML process. Section 7.6 has the syntax and semantics of FlowUML. Section 7.7 shows the expressibility of FlowUML by means of the running example. Section 7.8 describes the larger scope of FlowUML.

7.2 Flow Specification in the UML

The Use Case, which is one of the UML diagram models requirements, specifies the usage scenario between the system and its intended users [BRJ99]. These are followed by interaction diagrams specifying how objects in the use case interact with each other to achieve the end objective. Sequence diagrams are specific interaction diagrams that show such interactions as a sequence of messages exchanged between objects and ordered with respect to time. The sequence diagram shown in Figure 7.1 consists of four objects and two actors, where *Actor A* initiates the first flow as a request to read information from *Obj3* which then returns the requested information. Then, Actor A writes information to Obj4 and the *control* object, following which the flow to the *control* object triggers other flows.

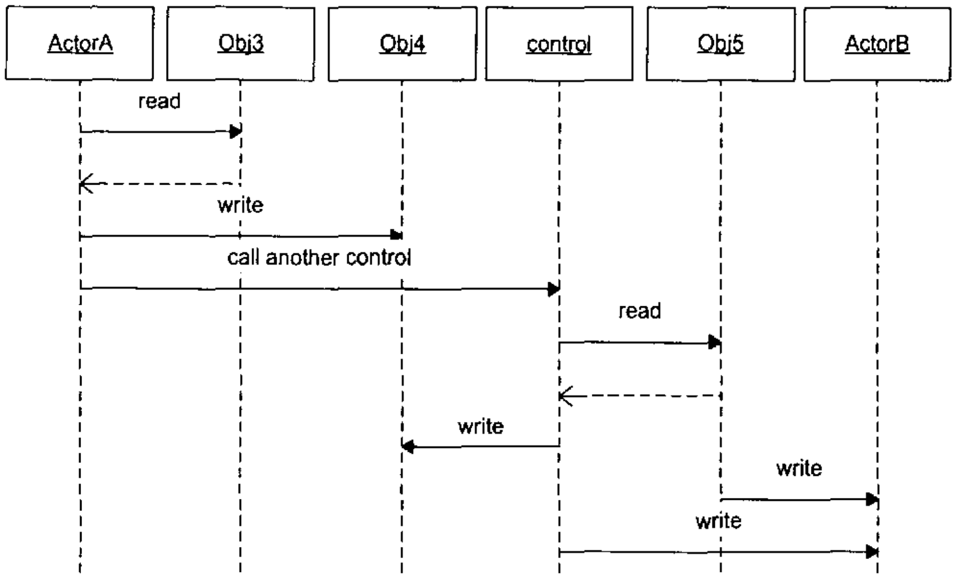


Figure 7.1: Sequence Diagram for Use Case2 (coarse grain)

As shown in Figure 7.1, every use case starts with an event from an actor followed by a sequence of interactions between internal objects and possible actors. During these interactions, information flows among objects and actors. Because sequence diagrams capture information flow, FlowUML extracts these flows from them. Thus, one could write a parser to automate this process from the output of tools such as Rational Rose [Ros03].

7.3 The Running Example

The example consists of two use cases. *Use case 1* has a simple scenario shown in Figure 7.2, and *use case 2* has a more complex scenario shown in Figure 7.3. The actor in *use case 1* reads information from object *Obj1* returned as an attribute *att1* and then writes it to *Obj2*. The actor in *use case 2* transfers information between two objects and then calls another object, leading to more information flow.

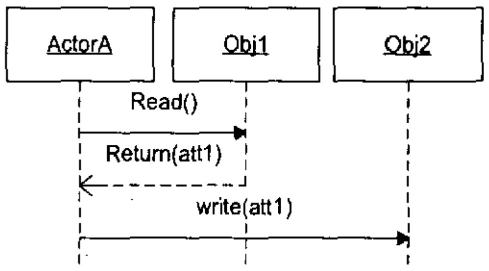


Figure 7.2: The Sequence Diagram for Use Case1

During the analysis stage of the software development, selected interactions are specified between objects. These interactions are further refined by specifying the message's attributes or parameters passed between objects. Therefore, the expressiveness and detail in a sequence diagram depends on the analysis stage. For example, the sequence diagram of use case 2 can be categorized into two views: coarse grain view shown in Figure 7.1 and the fine-grain view shown in Figure 7.3.

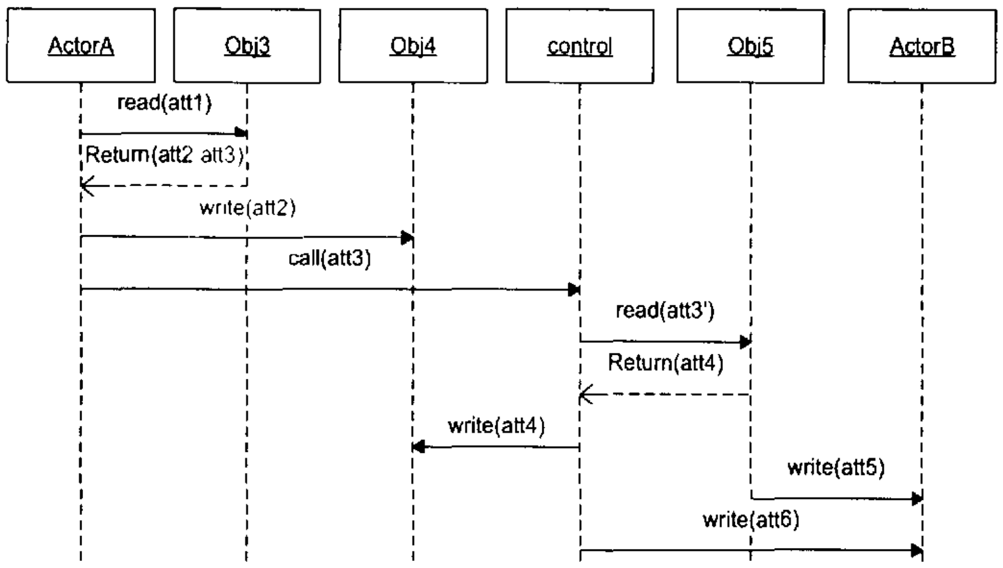


Figure 7.3: Sequence Diagram for Use Case 2 (fine grain)

7.4 Notations and Assumptions

7.4.1 Sources, Sinks and Object Types

Objects in sequence diagrams belong to three categories: *actors*, *entities* and *control objects*. Actors initiate flows and send/receive information to/from the system as externals. Every use case has at least one actor. Entity objects live longer and store information. Control objects provide coordination and application logic. Generally, actors and entity objects can be sources or sinks of information flows. Control objects do not store information, but they may create new information without storing them.

Axiom 1: *Sources and sinks of information flows are actors and entities.*

7.4.2 Method Names and Information Flows

Information flows in sequence diagrams arise due to attributes passed in method calls such as read and write. For example, the method call *read(att1)* that reads specific information from *obj3* exchanges information by sending *att1*. FlowUML uses any message as a flow of information, regardless of its name.

Axiom 2: *Any message with value is considered an information flow.*

7.4.3 Complex Message Constructs in the UML

Sequence diagrams construct complex messages from simple ones using four constructs: *creation messages* used to create new objects, *iteration messages* used to send data multiple times, and *conditional messages*. I consider a creation message as an information flow if the caller passes a creation parameter value to the callee. I consider an iterated message to constitute a single information flow. I consider a conditional message to be an information flow, regardless of the truth-value of the condition. I consider a simple message as a flow if it passes information.

7.4.4 Attribute Dependencies Across Objects

First, information flowing into an object may flow out unaltered; this is called *exact attribute flow* in FlowUML. Second, some attributes flowing out of an object may be different in name, but always have the same value as an attribute that flows into the same object. FlowUML requires this information to be in the *similar attribute table.* Third, some attributes flowing out of an object may depend upon others that flow into the same object. FlowUML requires this information to be in the *derived attribute table.* The reason for knowing these is that, during the requirements specification stage, exact method details may not be available – but, without such properties, it is not possible to derive the consequences of information flowed across objects. I formalize these in the following definition and axioms:

Definition 1: *An exact attribute is one that flows through an object, but does not change its value. When an attribute flows out of an object that depends upon a set of attributes flowing in to the same object, I say that the second attribute is derived from the first set.*

Axiom 3: *Attribute names are unique over a system.*

Axiom 4: *For every attribute that flows into an object, another attribute flows out of that object with the same value, but with a different attribute name, and both attribute names are listed in the similar attribute table and this is considered an exact attribute flow.*

For example, an input attribute named *familyName* flows out of *object A* as *lastName*. If their values are the same and listed in the *similar attribute table, lastName* is considered an exact flow of *familyName*. If an attribute *dateOfBirth* flows into an object, another attribute named *age* flows out of the same object and both are listed in the *derived attribute table, age* is considered a derived of *dateOfBirth*.

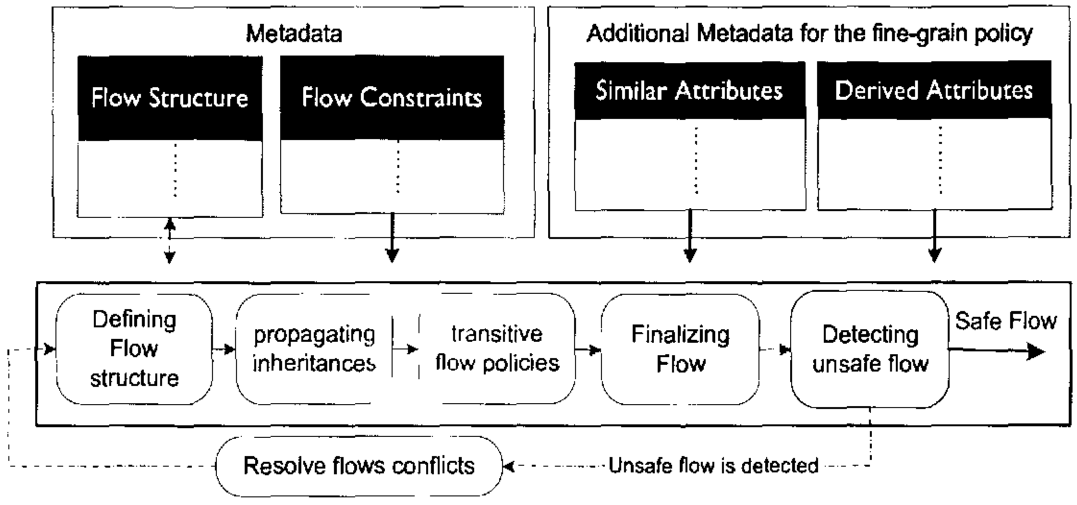


Figure 7.4: Steps in FlowUML

7.5 Verification Process

FlowUML verifies flow policies using five sequenced steps and four metadata sources as shown in Figure 7.4. They are defining basic structure, propagating flow information through inheritance hierarchies, inferring transitive flows, finalizing flows and detecting unsafe flows. These steps are the same for coarse grain and fine-grain policies. However, the details of information and the metadata sources are different when analyzing coarse- or fine-grain policies. Coarse-grain policies have less detail. Fine-grain policies incorporate attribute values and their dependencies.

7.5.1 Coarse-grain Policy Analysis

In coarse grain policy analysis, the available details of information about flow are the objects and how they interact with each other. The analysis starts with the first step, called *defining flow structure.* During this step, FlowUML extracts flow structure (objects, and their interaction) from sequence diagrams. This information is transformed into some basic FlowUML predicates. During the second step, basic predicates are propagated using the actor (or role) hierarchy. This step derives information flows implied due to inheritance. Although information flows can be transitive, the previous step does not derive those. Hence, the third step derives all transitive flows. The fourth step complements the third step by filtering results of the third step to those flows that satisfy properties of interests as specified in policies. For example, in Figure 7.1 the third step derives a flow from *Obj3* to *Actor A* directly, flow from *Obj3* to *Obj4* through *Actor A*, flow from *Obj3* to control through *Actor A* etc. The fourth step, filters out unwanted flows according to predefined policies. For example, a policy may only filter in non-transitive flows between only entities and actors; thus, only flows from *Obj3* to *Actor A* and the one from A*ctor A* to *Obj4*, but not the flow from *Obj3* to *Obj4,* may be of relevance. The last step detects flows that violate specified policies. At the moment, FlowUML does not attempt to resolve them automatically.

7.5.2 Fine-grain Policy Analysis

As stated in Section 7.4, fine-grain policies require two kinds of additional information. The first, given in the *similar attribute table,* contains distinct attribute names that always contain the same data value. The second, given in the *attribute derivation table,* lists attribute dependencies. This information is useful in controlling the flow of sensitive information. For example, in Figure 7.3, if there is a record in the *attribute derivation table* stating that *att6* is derived from *att3* in the *control* object, FlowUML concludes that there is a transitive flow from *Actor A* to *Actor B.*

7.6 Syntax and Semantics

FlowUML terms are either variables or constants belonging to five individual sorts: actors, objects, use cases, attributes and times. Constants belong to sets A, Obj, UC, Att, and T, and variables are represented as X_a, X_{obj}, X_{uc}, X_{att} and X_t, respectively. FlowUML uses a set of predicates, as summarized in Tables 1 and 2 and categorized as follow:

Basic predicates for supporting the specification of flows, called FlowSup.

1. A unary predicate isEntity(X_{obj}) meaning X_{obj} is an entity.
2. A unary predicate isActor(X_{obj}) meaning X_{obj} is an actor.

3. A binary predicate specializedActor(X_a,X'_a) meaning X'_a is a specialized actor of X_a.

4. A binary predicate precedes(X_{uc}, X'_{uc}) meaning use cases X_{uc} and X'_{uc} are executed in that order.

5. A binary predicate sameAtt(X_{att}, X'_{att}) meaning attributes X_{att} and X'_{att} have the same value.

6. A ternary predicate derAtt(X_{att}, X'_{att}, X_{obj}) meaning that the flowing-out attribute X'_{att} is derived from the flowing-in attribute X_{att}, an that derivation is occurring at object X_{obj}.

7. A ternary predicate ignoreFlow(X_a, X_{obj}, X'_{obj}) meaning to exclude the flow from object X_{obj} to object X'_{obj} from being considered as a violation of flow control policies.

8. A 6-ary predicate ignoreFlows(X_{a1}, X_{obj1}, X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2}) to exclude the flow from X_{obj1} to X'_{obj} and the flow from X_{obj2} to X'_{obj2} from being considered as a violation of flow control policies. Two similar predicates ignoreFlows$_{att}$(X_{a1},X_{att},X_{obj1},X'_{obj1},X_{a2},X_{obj2},X'_{obj2}) and ignoreFlow$_{att}$(X_a,X_{att},X_{obj}, X'_{obj}) are used in fine-grain policies.

Predicates for specifying flow constraints, called ConstSup.

1. A binary predicate dominates(X_{obj},X'_{obj}) meaning that the security label of object X'_{obj} dominates or equals the security label of object X_{obj}. It is used in multi-level security policies.

2. A binary predicate $\mathrm{ACL}(X_{obj},X'_{obj},X_{AT})$ meaning that object X_{obj} is in the access control list of object X'_{obj}. It is used in discretionary access control policies. X_{AT} is an operation such as *read* or *write*.
3. A binary predicate $\mathrm{conflictingActors}(X_{obj}, X'_{obj})$ meaning that actors X_{obj} and X'_{obj} are in conflict with each other for a specific reason. For example, both actors cannot flow information to the same object or there must not be a flow of information between them.
4. A binary predicate $\mathrm{conflictingEntities}(X_{obj}, X'_{obj})$ meaning both entities X_{obj} and X'_{obj} are in conflict with each other for specific reason. For example, each entity belongs to different competitive company and information must not flow in between.

Predicates for coarse-grain policies

1. A 6-ary predicate $\mathrm{Flow}(X_a,X_{obj},X'_{obj},X_t,X_{uc}, X_{op})$ meaning there is a simple flow initiated by actor X_a from object X_{obj} to object X'_{obj} at time X_t in use case X_{uc} and operation X_{op}.
2. A 5-ary predicate $\mathrm{mayFlow}(X_a,X_{obj},X'_{obj},X_t, X_{uc})$ is the transitive closure of the previous predicate.
3. $\mathrm{mayFlow}_{\mathrm{interUC}}(X_a,X_{obj},X'_{obj},X_t,X_{uc})$ is similar to $\mathrm{mayFlow}$ but the scope of $\mathrm{mayflow}_{\mathrm{interUC}}$ is between use cases instead of focusing in one use case. Note that I want to know the beginning and ending operations rather than the beginning and ending use cases.
4. A 4-ary predicate $\mathrm{finalFlow}(X_aX_{obj},X'_{obj},X_t)$ meaning a finalized flow.

5. $\mathtt{finalFlow}_{\mathtt{interUC}}(X_a,X_{obj},X'_{obj},X_t)$ is similar to $\mathtt{finalFlow}$, but it covers flows between use cases.
6. A ternary predicate $\mathtt{unsafeFlow}(X_a,X_{obj}, X'_{obj})$ meaning there is an unsafe flow from X_{obj} to X'_{obj} initiated by actor X_a.
7. A 6-ary predicate $\mathtt{unsafeFlows}(X_{a1},X_{obj1}, X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2})$ meaning there are two unsafe flow. The first, initiated by actor X_{a1} flows from X_{obj1} to X'_{obj1}. The second, initiated by actor X_{a2} flows from object X_{obj2} to object X'_{obj2}. They are unsafe because together they violate a flow constraint.
8. A ternary predicate $\mathtt{safeFlow}(X_a,X_{obj},X'_{obj})$ meaning that the flow initiated by actor X_a from object X_{obj} to object X'_{obj} is safe.

Predicates for fine-grain policies

The predicates used to specify fine-grain access control policies are similar to the ones for coarse grain policy, but include X_{att} as an attribute flowing between objects. They are as follows:

$\mathtt{Flow}_{att}(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc}, X_{op})$,

$\mathtt{mayFlow}_{att}(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc})$,

$\mathtt{mayFlow}_{\mathtt{interUC\_att}}(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc})$,

$\mathtt{finalFlow}_{att}(X_a,X_{att},X_{obj},X'_{obj},X_t)$,

$\mathtt{finalFlow}_{\mathtt{interUC\_att}}(X_a,X_{att},X_{obj},X'_{obj},X_t)$,

$\mathtt{unsafeFlow}_{att}(X_a,X_{att},X_{obj},X'_{obj})$,

$\mathtt{unsafeFlows}_{att}(X_{a1},X_{att1},X_{obj1},X'_{obj1},X_{a2},X_{att2},X_{obj2},X'_{obj2})$, safeFlow$_{att}$($X_a$, X_{att1},X_{obj},X'_{obj}).

7.6.1 Semantics of FlowUML

A FlowUML rule is of the form $L \leftarrow L_1, \ldots L_n$ where L , $L_1, \ldots L_n$ are literals satisfying the conditions stated in Table 7.1 and Table 7.2. Rules constructed according to these specifications form a locally stratified logic program and, therefore, has a unique stable model and that stable model is also a well-founded model [GL88].

Table 7.1. FlowUML's Strata for Coarse-grain Policies

Phase	Stratum	Predicate	Rules defining the predicate
Coarse-grain	0	FlowSup predicates	base relations.
		ConstSup predicates	base relations
	1	Flow(X_a, X_{obj}, X'_{obj}, X_t, X_{us}, X_{op})	body may contain FlowSup predicates
	2	mayFlow(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})	body may contain literal from strata 0 to 2
	3	mayFlow$_{interUC}$(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})	body may contain literal from strata 0, 1 and 3
	4	finalFlow(X_a, X_{obj}, X'_{obj}, X_t) finalFlow$_{interUC}$(X_a, X_{obj}, X'_{obj}, X_t) unsafeFlow(X_a, X_{obj}, X'_{obj})	body may contain literal from strata 0 to 3
	5	unsafeFlows(X_{a1}, X_{obj1}, X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2})	body may contain literal from strata 0 to 4
	6	safeFlow(X_a, X_{obj}, X'_{obj})	body may contain literal from strata 0 to 5

Table 7.2. FlowUML's Strata for Fine-grain Policies

Phase	Stratum	Predicate	Rules defining the predicate
Fine-grain	0	FlowSup predicates	base relations.
		ConstSup predicates	base relations
	1	$Flow_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op})$	body may contain FlowSup predicates
	2	$mayFlow_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc})$	body may contain literal from strata 0, 1 and 2
	3	$mayFlow_{interUC\ att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc})$	body may contain literal from strata 0, 1 and 3
	4	$finalFlow_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t)$ $finalFlow_{interUC\ att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t)$	body may contain literal from strata 0 to 3
	5	$unsafeFlow_{att}(X_a, X_{att}, X_{obj}, X'_{obj})$ $unsafeFlows_{att}(X_{a1}, X_{att1}, X_{obj1}, X'_{obj1}, X_{a2}, X_{att2}, X_{obj2}, X'_{obj2})$	body may contain literal from strata 0 to 4
	6	$safeFlow_{att}(X_a, X_{att}, X_{obj}, X'_{obj})$	body may contain literal from strata 0 to 5

7.7 Applying FlowUML

This section shows how some samples of FlowUML policies and how they can be used to detect unsafe flows.

7.7.1 Basic Flow Predicates

Examples of flow information available in Figure 7.2 are given in rules (1). The rules (2) to (6) are instances of *FlowSup* predicates valid in Figures 2 and 3.

$Flow_{att}(ActorA,att1,Obj1,ActorA,1,uc1,op1) \leftarrow$ (1)

$Flow_{att}(ActorA,att1,ActorA,Obj2,2,uc1,op2) \leftarrow$

isEntity(obj1)← (2)

isActor(actorA)← (3)

precedes(uc1,uc2)← (4)

sameAtt(att3, att3')← (5)

derAtt(att3, att6, control)← (6)

7.7.2 Propagation Policies

The second step applies a policy that propagates flows along actor hierarchies. In the example, if there is a specialized actor say *Actor C* of *Actor B* then *Actor C* receives *att5* and *att6*. Policies stating acceptable inheritances can be stated in example rules such as (7) to (9).

$$\mathrm{Flow}(X'_{a}, X_{obj}, X'_{obj}, X_{t}, X_{uc}, X_{op}) \leftarrow \mathrm{Flow}(X_{a},X_{obj},X'_{obj},X_{t},X_{uc},X_{op}), \mathrm{specializedRole}(X_{a}, X'_{a}) \quad (7)$$

$$\mathrm{Flow}(X_{a}, X''_{obj}, X'_{obj}, X_{t}, X_{uc}, X_{op}) \leftarrow \mathrm{Flow}(X_{a},X_{obj},X'_{obj},X_{t},X_{uc}, X_{op}), \mathrm{isActor}(X_{obj}), \mathrm{specializedActor}(X_{obj},X''_{obj}) \quad (8)$$

$$\mathrm{Flow}(X_{a}, X_{obj}, X''_{obj}, X_{t}, X_{uc}, X_{op}) \leftarrow \mathrm{Flow}(X_{a},X_{obj},X'_{obj},X_{t},X_{uc}, X_{op}) ,\mathrm{isActor}(X'_{obj}),\mathrm{specializedActor}(X'_{obj},X''_{obj}) \quad (9)$$

Rule 7 says that every actor that plays a specialized role of an actor that initiates a flow also initiates an inherited flow. In rules 8 and 9, for every flow from or to an actor, respectively, rules 8 and 9 add new information flow for every specialized actor of the actor who sends or receives the information, respectively.

7.7.3 Transitive Flow Policies

Policies written for the third step recursively construct transitive flows from basic ones. This step is sufficiently flexible to accommodate various options discussed in Section 7.4.5. Due to space constraints I show some examples in rules 10 through 13. Rule 10 declares a basic flow to be a transitive flow and rules 11 specifies all possible information flows between any types of objects and rules 12 specifies possible flows that goes through an intermediate object that is not an actor or an entity. The flow between *Obj5* and *Obj4* are examples of such flows. Rule 13 specifies all flows that respect the precedent order between their use cases. The flow between *Obj1* and *Obj3* is such an example.

$$\text{mayFlow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \text{Flow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \qquad (10)$$

$$\begin{aligned} &\text{mayFlow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \text{Flow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \\ &\text{mayFlow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \text{Flow}(X_a, X_{obj}, X_{BTWobj}, X_t, X_{uc}, X_{op}) \\ &\qquad\qquad \text{mayFlow}(X_a, X_{BTWobj}, X'_{obj}, X'_t, X_{uc}),\ X_t < X'_t \end{aligned} \qquad (11)$$

$$\begin{aligned} &\text{mayFlow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \text{Flow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \\ &\text{mayFlow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \text{Flow}(X_a, X_{obj}, X_{BTWobj}, X_t, X_{uc}, X_{op}), \\ &\qquad\qquad \text{mayFlow}(X_a, X_{BTWobj}, X'_{obj}, X'_t, X_{uc}), \\ &\qquad\qquad X_t < X'_t,\ \neg(\text{isActor}(X_{BTWobj});\ \text{isEntity}(X_{BTWobj})) \end{aligned} \qquad (12)$$

$$\begin{aligned} &\text{mayFlow}_{\text{interUC}}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \text{Flow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \\ &\text{mayFlow}_{\text{interUC}}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \\ &\qquad\qquad \text{Flow}(X_a, X_{obj}, X_{BTWobj}, X_t, X_{uc}, X_{op}), \\ &\qquad\qquad \text{mayFlow}_{\text{interUC}}(X_a, X_{BTWobj}, X'_{obj}, X'_t, X'_{uc}), \\ &\qquad\qquad ((X_t < X'_t,\ X_{uc} = X'_{uc});\ \text{precedes}(X_{uc}, X'_{uc})) \end{aligned} \qquad (13)$$

Previous examples show coarse-grain flow policies. I now show some fine-grain flow polices. The first example permits flows using different attribute names to contain the same value from the *similar attribute table*. Rule 14 states that if *att3* and *att3'* in Figure 7.3 have the same value then the flow of *att3* from *actor A* to *Obj5* is permitted.

$$\begin{array}{l} \text{mayFlow}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \\ \quad \text{Flow}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \\ \text{mayFlow}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \\ \quad \text{Flow}(X_a, X_{att}, X_{obj}, X_{BTWobj}, X_t, X_{uc}, X_{op}) \\ \quad \text{mayFlow}(X_a, X'_{att}, X_{BTWobj}, X'_{obj}, X'_t, X_{uc}), \\ \quad X_t < X'_t, (\text{sameAtt}(X_{att}, X'_{att}); X_{att} = X'_{att}) \end{array} \quad (14)$$

Rule 15 specifies that all transitive flows are accepted provided that attribute derivation is confined to intermediate objects. For example, if Figure 7.3 had an entry in the *derived attribute table* stating that *att6* is derived from *att3* in the *control* object, the flow from actor A to actor B is permitted.

$$\begin{array}{l} \text{mayFlow}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \\ \qquad \text{Flow}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \\ \text{mayFlow}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow \\ \qquad \text{Flow}(X_a, X_{att}, X_{obj}, X_{BTWobj}, X_t, X_{uc}, X_{op}) \\ \qquad \text{mayFlow}(X_a, X'_{att}, X_{BTWobj}, X'_{obj}, X'_t, X_{uc}), \\ \qquad X_t < X'_t, (\text{sameAtt}(X_{att}, X'_{att}); \\ \qquad X_{att} = X'_{att}, \text{derAtt}(X_{att}, X'_{att}, X_{BTWobj})) \end{array} \quad (15)$$

7.7.4 Finalizing Flows

After propagating flows along inheritance hierarchies and extending them to become transitive by using recursive rules, the fourth step provides filtering policies to choose desired flows. An example is rule 16 that chooses all possible information flow that starts and ends in actors, entities or both inside a single use case. Rule 17 does the same, but across use cases.

$$\begin{aligned} &\mathrm{finalFlow}(X_a, X_{obj}, X'_{obj}, X_t) \leftarrow \mathrm{mayFlow}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}),\\ &\quad(\mathrm{isActor}(X_{obj}); \mathrm{isEntity}(X_{obj})),(\mathrm{isActor}(X'_{obj}); \mathrm{isEntity}(X'_{obj}))\end{aligned} \tag{16}$$

$$\begin{aligned} &\mathrm{finalFlow}_{interUC}(X_a, X_{obj}, X'_{obj}, X_t) \leftarrow\\ &\qquad\mathrm{mayFlow}_{interUC}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}),\\ &\qquad(\mathrm{isActor}(X_{obj}); \mathrm{isEntity}(X_{obj})),\\ &\qquad(\mathrm{isActor}(X'_{obj}); \mathrm{isEntity}(X'_{obj}))\end{aligned} \tag{17}$$

7.7.5 Detecting Unsafe Flows with Respect to Policies

This section shows FlowUML specification of known flow control policies, and how to detect unsafe information flows with respect to them.

Mandatory Access Control (MAC) restricts subjects in accessing objects that are classified higher than them [BL75]. In order to do so, all objects are labeled with sensitivity levels and all users have clearance levels. Rule 18 ensures that, if information flows from *Obj1* to *Obj2* and the *Obj2* does not dominate or equals the security label of *Obj1,* this is considered unsafe.

$$\begin{aligned} &\mathrm{unsafeFlow}(X_a, X_{obj1}, X_{obj2}) \leftarrow \mathrm{finalFlow}(X_a, X_{obj1}, X_{obj2}, X_t),\\ &\qquad\neg\mathrm{dominates}(X_{obj1}, X_{obj2})\end{aligned} \tag{18}$$

Discretionary access control (DAC) allows subjects to access objects solely based on the subject's identity and authorization. Object owners have the discretion to allow other subjects to access their objects using access control lists (ACL). Rule 19 specifies unauthorized information flows from an actor to an object and rule 20 specifies unauthorized flows from an object to an actor.

$$\text{unsafeFlow}_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}) \leftarrow \text{finalFlow}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X_t), \neg\text{ACL}(X_{obj1}, X_{obj2}, w) \quad (19)$$

$$\text{unsafeFlow}_{att}(X_a, X_{att1}, X_{obj2}, X_{obj1}) \leftarrow \text{finalFlow}(X_a, X_{att1}, X_{obj2}, X_{obj1}, X_t), \neg\text{ACL}(X_{obj1}, X_{obj2}, r) \quad (20)$$

Static separation of duty (SsoD) prevents actors that have conflicts, e.g., the account payable manager and the purchasing manager accessing the same object. Policies can restrict a particular information flow between two conflicting actors such as is specified in rule 21.

$$\text{unsafeFlow}(X_a, X_{obj1}, X_{obj3}) \leftarrow \text{finalFlow}(X_a, X_{obj1}, X_{obj2}, X_t), \text{finalFlow}(X_a, X_{obj2}, X_{obj3}, X'_t), \text{conflictingActors}(X_{obj1}, X_{obj3}), \text{isActor}(X_{obj1}), \text{isActor}(Xobj3), Xt<X't \quad (21)$$

Another example policy restricts passing two attributes by the same actor, as stated in rule 22. A third example in rule 23 prevents two conflicting actors from passing the same attribute to the same object.

$$\text{unsafeFlows}_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X'_a, X_{att2}, X_{obj3}, X_{obj2}) \leftarrow \text{finalFlow}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X_t), \text{finalFlow}(X'_a, X_{att2}, X_{obj3}, X_{obj2}, X'_t), \text{isActor}(X_{obj1}), \text{isActor}(X_{obj3}), X_{obj1}=X_{obj3}, X_{att1}\neq X_{att2} \quad (22)$$

$$\begin{aligned}\text{unsafeFlows}_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X'_a, X_{att1}, X_{obj3}, X_{obj2}) \leftarrow\; & \text{finalFlow}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X_t),\\ & \text{finalFlow}(X'_a, X_{att1}, X_{obj3}, X_{obj2}, X'_t),\\ & \text{conflictingActors}(X_{obj1}, X_{obj3}),\\ & \text{isActor}(X_{obj1}), \text{isActor}(X_{obj3})\end{aligned} \quad (23)$$

In detecting unsafe information flows, FlowUML raises an alert to the analyst to resolve it and run FlowUML again. However, the analyst can tolerate particular violations, as shown in rule 24 that modifies rule 18. Rule 24 states that, if an information flows from *Obj1* to *Obj2* and *Obj2* does not dominate or equals the security label of *Obj1* and the security analyst has not tolerated it before, it is an unsafe flow. I allow this option because some specification methods tolerate known inconsistencies [NE01,[GH91,GH92].

$$\begin{aligned}\text{unsafeFlow}(X_a, X_{obj1}, X_{obj2}) \leftarrow\; & \text{finalFlow}(X_a, X_{obj1}, X_{obj2}, X_t),\\ & \neg\text{dominates}(X_{obj1}, X_{obj2}), \neg\text{ignoreFlow}(X_a, X_{obj}, X'_{obj})\end{aligned} \quad (24)$$

Rules 25, 26 and 27 declare any flow that includes an unsafe flow fragment to be unsafe. Rules 28 and 29 are related to unsafeFlows predicate that detects two flows to be unsafe, the rules mark every single flow in that predicate as a single unsafe flow.

$$\begin{aligned}\text{unsafeFlow}(X_a, X_{obj1}, X_{obj3}) \leftarrow\; & \text{mayFlow}(X_a, X_{obj1}, X_{obj3}, X_t),\\ & \text{unsafeFlow}(X_a, X_{obj2}, X_{obj3}),\\ & \text{mayFlow}(X_a, X_{obj1}, X_{obj2}, X_t)\end{aligned} \quad (25)$$

$$\begin{aligned}\text{unsafeFlow}(X_a, X_{obj2}, X_{obj4}) \leftarrow\; & \text{mayFlow}(X_a, X_{obj2}, X_{obj4}, X_t),\\ & \text{unsafeFlow}(X_a, X_{obj2}, X_{obj3}),\\ & \text{mayFlow}(X_a, X_{obj3}, X_{obj4}, X_t)\end{aligned} \quad (26)$$

$$\begin{aligned}\text{unsafeFlow}(X_a, X_{obj1}, X_{obj4}) \leftarrow\ & \text{mayFlow}(X_a, X_{obj1}, X_{obj4}, X_t), \\ & \text{unsafeFlow}(X_a, X_{obj2}, X_{obj3}), \\ & \text{mayFlow}(X_a, X_{obj1}, X_{obj2}, X_t), \\ & \text{mayFlow}(X_a, X_{obj3}, X_{obj4}, X_t)\end{aligned} \tag{27}$$

$$\begin{aligned}&\text{unsafeFlow}_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}) \leftarrow \\ &\qquad \text{unsafeFlows}_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X'_a, X_{att2}, X_{obj3}, X_{obj2})\end{aligned} \tag{28}$$

$$\begin{aligned}&\text{unsafeFlow}_{att}(X_a, X_{att2}, X_{obj3}, X_{obj2}) \leftarrow \\ &\qquad \text{unsafeFlows}_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}, X'_a, X_{att2}, X_{obj3}, X_{obj2})\end{aligned} \tag{29}$$

The last step is a completion rule specifying that every flow is safe provided that it cannot be derived to be unsafe, as shown in rule 30.

$$\text{safeFlow}(X_a, X_{obj}, X'_{obj}) \leftarrow \neg\text{unsafeFlow}(X_a, X_{obj}, X'_{obj}) \tag{30}$$

7.8 The larger scope of FlowUML

This section describes the larger scope and applicability of FlowUML in incorporating security models to the software development life cycle. First, as shown in Figure 7.5 FlowUML transforms the geometric information in the UML views to a set of predicates that can be used as basic predicates in flow control policies written as Horn clauses. I have shown how such policies can be used to derive compliance of sequence diagrams to flow control policies. Because the UML pictorial sequence diagrams are saved as text files (such as .mdl files of Rational Rose [Ros03]) with an appropriate translator, I can now automate policy compliance checking of flow control policies by using appropriate logic programming engines, such as Prolog.

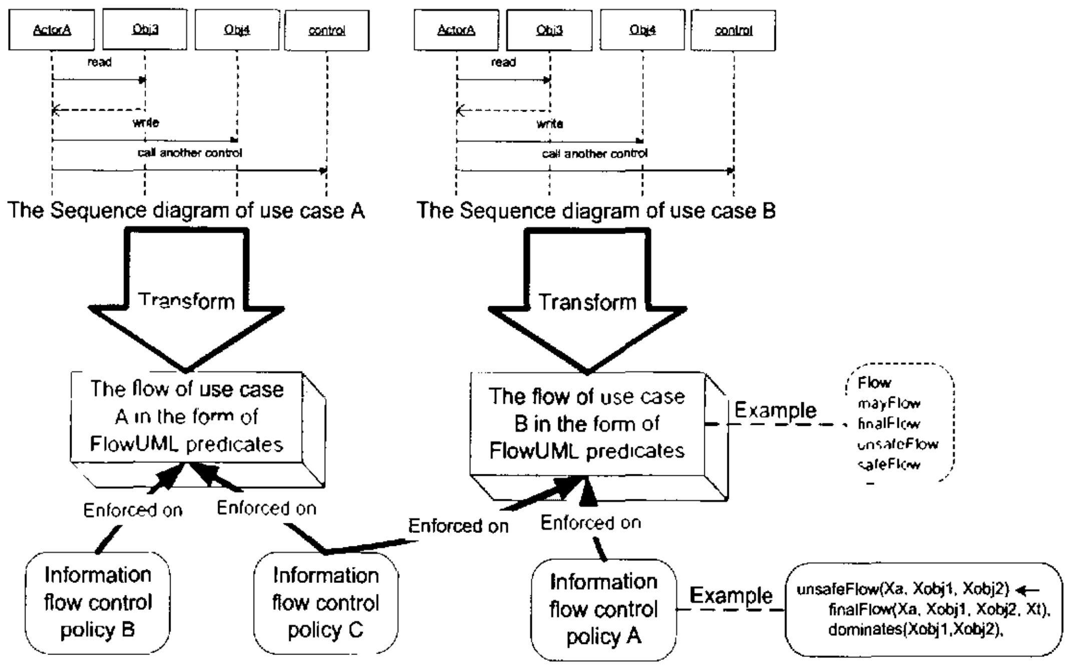

ActorA
Obj3
Obj4
control
read
write
call another control
The Sequence diagram of use case A
The Sequence diagram of use case B
Transform
Transform
The flow of use case A in the form of FlowUML predicates
The flow of use case B in the form of FlowUML predicates
Example
Flow
mayFlow
finalFlow
unsafeFlow
safeFlow
Enforced on
Enforced on
Enforced on
Enforced on
Information flow control policy B
Information flow control policy C
Information flow control policy A
Example
unsafeFlow(Xa, Xobj1, Xobj2) ←
finalFlow(Xa, Xobj1, Xobj2, Xt),
dominates(Xobj1,Xobj2),

Figure 7.5. FlowUML's Scope

Second, other than the basic predicates used to capture the geometric information given in the UML sequence diagrams, other Horn clauses of FlowUML constitute policies that are applicable at the early stages of the software design cycle. Thus, this division of predicate layers shows the clear separation of the basic geometry of the design from policy. As shown in the right hand side of Figure 7.5, the latter constituting of recursive rules are applied to a design constituting instances of basic predicates. Therefore, FlowUML can be considered an example framework to write policies applicable to the UML .

This separation of policy from the application has many consequences. The first is that it facilitates applying any policy to any design As shown in Figure 7.5, policies B and C can be separately applied to the sequence diagram of use case A. Similarly, as shown, policies A and B can be separately applied to the sequence diagram of use case B. This shows that more than one policy applies to one design diagram and that one policy applies to more than one diagram.

Third, the same process can be used to check the consistency of two design diagrams with respect to a given security policy. That is, if two design diagrams are compliant with a given policy, as far as that policy is concerned, they are indistinguishable. I developed this concept further in designing *a notion of policy based equivalence* of design diagrams in the UML .

Fourth, if the UML policies can be separated from designs as shown here, a policy composition framework for the UML policy compositions along the lines of [BCVS00],[WJ02] can be developed.

Last, by capturing more rules related to geometry of sequence diagrams, one may be able to capture deficiencies in the diagrams. If successful, this may lead to a policy-based, reverse engineering, framework for the UML diagrams.

7.9 Conclusions

FlowUML is a logic programming based framework to specify and verify the compliance of information flow requirements with the information flow control policies in the UML based designs at the early phases of the software development life cycle. I have demonstrated the flexibility and the expressiveness by showing how existing information flow control policies can be verified with FlowUML specifications.

Chapter 8

INTEGRATING THE ANALYSIS OF ACCESS AND FLOW CONTROL POLICIES

The previous two chapters addressed both policies separately. However, because both policies have a tight relationship between them, integrating the analysis of both policies will improve the validation and enforcement of access and flow control policies during the software development life cycle. I integrate in this chapter both AuthUML and FlowUML. I provide a collaboration framework and I show how to transform one framework output to other framework. Also, I show the flexibility and scalability behind the new integrated framework of AuthUML and FlowUML.

The integration is desired because, as I will show shortly, both access and flow control policies are tightly coupled to each other at the design stage, they both overlap, and each one relies and/or provides useful information to the other.

The reminder of the chapter is organized as follows. A running example to be used during the chapter is presented in Section 8.2. The integration of between AuthUML and FlowUML is described in Section 8.3. Section 8.4 describes the flexibility and scalability of the two frameworks.

8.1 Running Example

In this section, I introduce an example to be used during the rest of the chapter. Figure 8.1 shows an example of an abstract use case. It shows the different use cases and the actors who are allowed or not allowed to invoke them. The example represents a purchasing process where a Clerk prepares an order, and then a Purchasing Officer places the order. Later, the Clerk writes a check for the order and the Manager signs it. The actors (roles) are ordered in a hierarchy where every actor inherits the permissions of its higher actors. For example, the Purchasing Officer inherits the authorizations of the Clerk and, thus, the Purchasing Officer is allowed to prepare orders and write checks implicitly. The links between actors and use cases is considered as permission while the link with *Deny* is considered a denial execution. Each use case consists of one or more operations to achieve the objective of the use case. Sometimes an operation can be part of two use cases. Figure 8.2 shows the sequence diagram of *the Prepare order* use case.

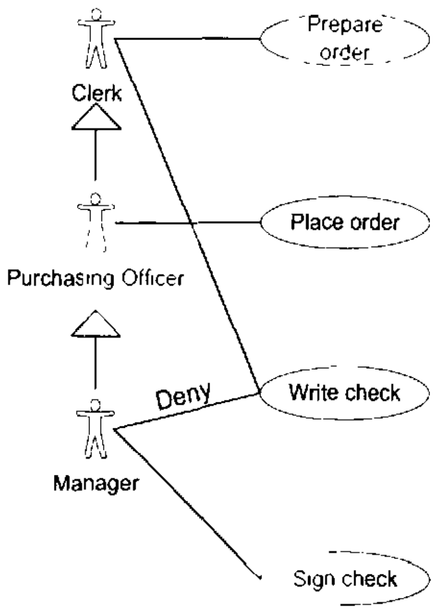


Figure 8.1 Use Cases of the Purchasing Process

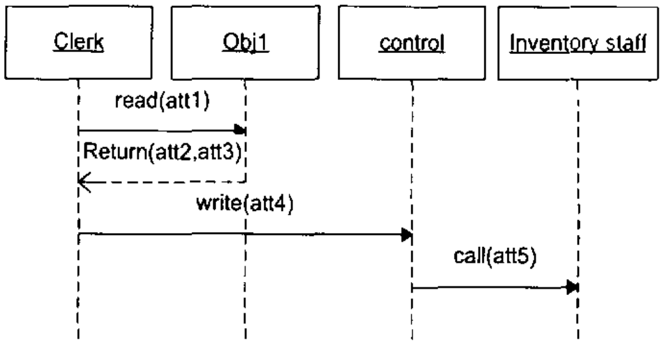


Figure 8.2 . Example of Sequence Diagram

The following is the result of extracting the information flow from the sequence diagram in Figure 8.2 that shows an example of a sequence diagram corresponding to the *Prepare order* use case.

Flow_{att}(Clerk, att_1, Clerk, Obj_1, 1, "Prepare order", read)← (6)
Flow_{att}(Clerk, att_2, Obj_1, Clerk, 2, "Prepare order", read)←
Flow_{att}(Clerk, att_3, Obj_1, Clerk, 2, "Prepare order", read)←
Flow_{att}(Clerk, att_4,Clerk,control,3,"Prepare order", write)←
Flow_{att}(Clerk,att_5,control,Inventory staff,4,"Prepare order",call)←

8.2 AuthUML and FlowUML Collaboration

AuthUML analyzes the access control, but does not analyze the control of information flow. For example, Actor A has an authorization to access operation *op1,* but the invocation of *op1* will result in sending information to actor *B* who may not be allowed to see that information. Conversely, FlowUML provides deeper analysis of authorization because it goes deep in to the attribute level of operations and considers implicit operations that are not directly called by actors that occur as consequences of initiated operations. Thus, FlowUML provides new useful information to AuthUML that will not be available otherwise. Therefore, FlowUML complements AuthUML by providing it with information that was unavailable otherwise.

FlowUML is applied between phase 2 and 3 of AuthUML for several reasons. First, FlowUML lists all operations of a use case that are taken from the sequence diagram.

Second, it reduces redundancy by extracting operations of use cases and improves analysis efficiency by ensuring the proper flow of information before analyzing the authorizations operation level. Third, FlowUML provides new valuable data to AuthUML that in turn well validate more access control policies.

8.2.1 Collaboration Process

The collaboration between AuthUML and FlowUML is shown in Figure 8.3 and in more detail in Figure **8.4**. Because the new collaboration process requires inserting FlowUML between phases two and three of AuthUML, the combined process consists of five stages.

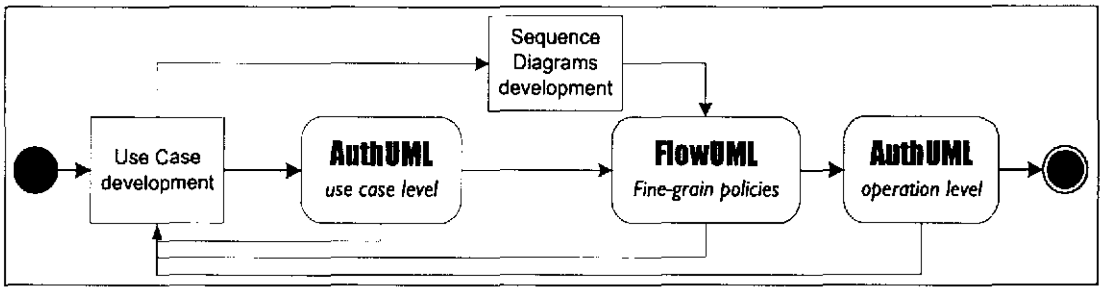


Figure 8.3. AuthUML and FlowUML Collaboration

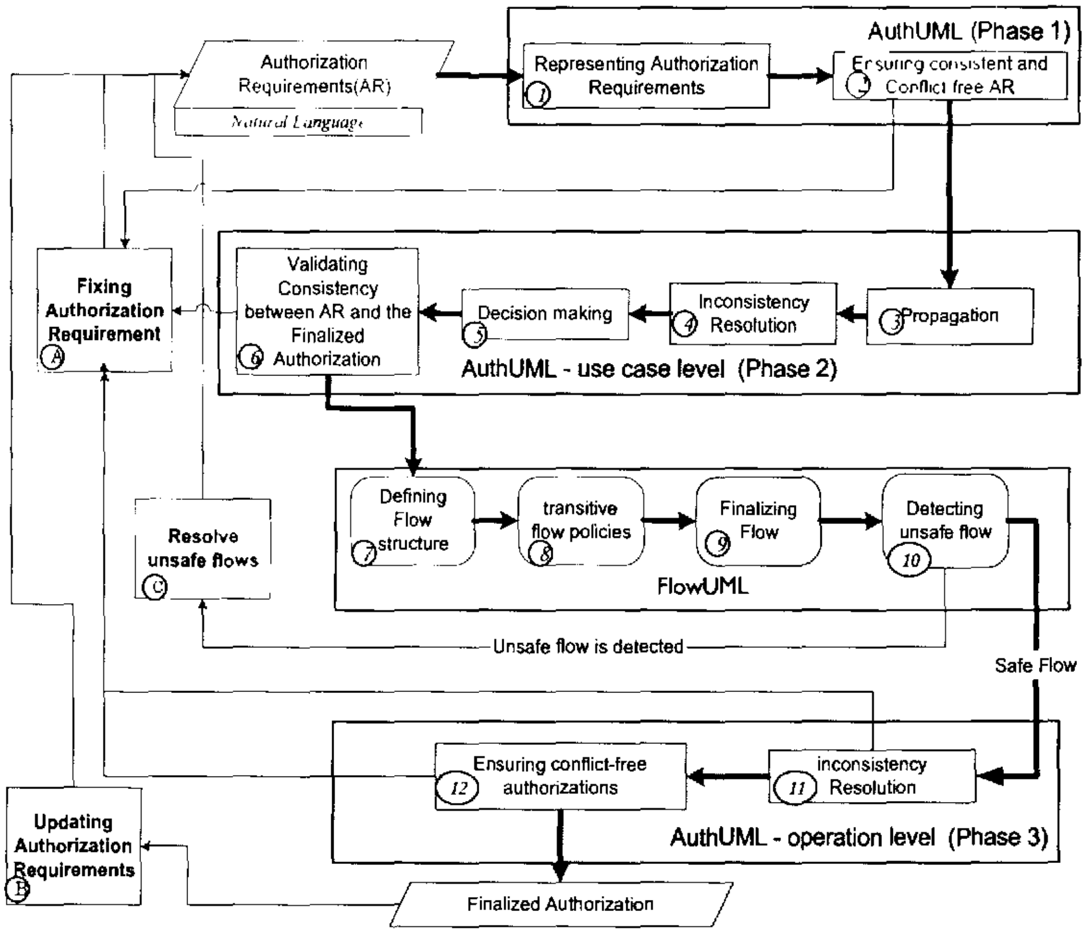

AuthUML (Phase 1)
Authorization Requirements(AR)
Natural Language
Representing Authorization Requirements
1
Ensuring consistent and Conflict free AR
2
Fixing Authorization Requirement
A
Validating Consistency between AR and the Finalized Authorization
6
Decision making
5
Inconsistency Resolution
4
Propagation
3
AuthUML - use case level (Phase 2)
Defining Flow structure
7
transitive flow policies
8
Finalizing Flow
9
Detecting unsafe flow
10
FlowUML
Resolve unsafe flows
C
Unsafe flow is detected
Safe Flow
Ensuring conflict-free authorizations
12
inconsistency Resolution
11
AuthUML - operation level (Phase 3)
Updating Authorization Requirements
B
Finalized Authorization

Figure 8.4. Comprehensive AuthUML and FlowUML Collaboration

8.2.1.1 The Collaboration of Predicates

AuthUML and FlowUML are integrated well with each other to improve the analysis of both access and flow control policies. Both frameworks' predicates collaborate seamlessly with each other where one predicate is built on top of the other one. Figure **8.5** shows how the predicates of both frameworks are merged into one strata to provide the needed collaboration.

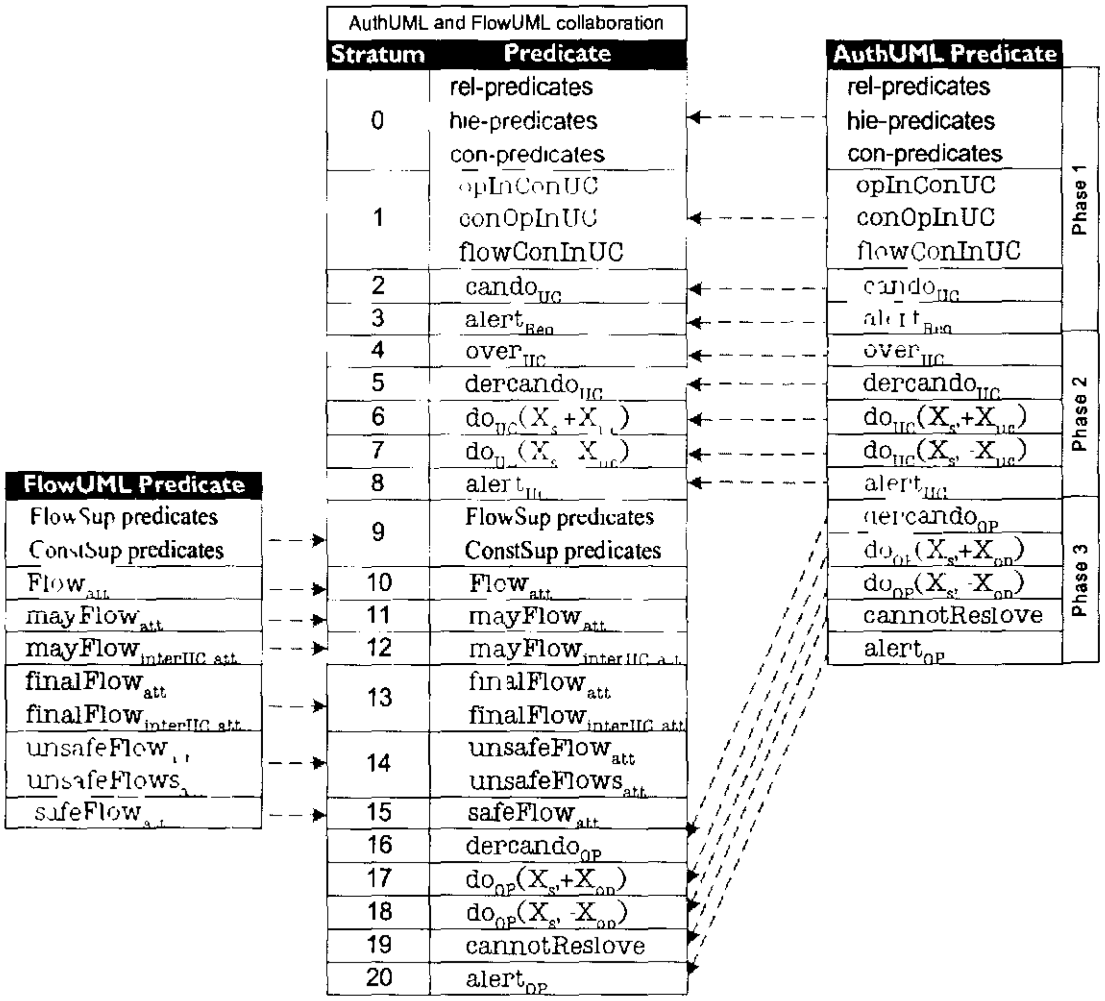

AuthUML and FlowUML collaboration
Stratum
Predicate
0 rel-predicates hie-predicates con-predicates
1 opInConUC conOpInUC flowConInUC
2 cando$_{UC}$
3 alert$_{Req}$
4 over$_{UC}$
5 dercando$_{UC}$
6 do$_{UC}$(X$_s$,+X$_{uc}$)
7 do$_{UC}$(X$_s$, -X$_{uc}$)
8 alert$_{UC}$
9 FlowSup predicates ConstSup predicates
10 Flow$_{att}$
11 mayFlow$_{att}$
12 mayFlow$_{interUC\ att}$
13 finalFlow$_{att}$ finalFlow$_{interUC\ att}$
14 unsafeFlow$_{att}$ unsafeFlows$_{att}$
15 safeFlow$_{att}$
16 dercando$_{OP}$
17 do$_{OP}$(X$_s$,+X$_{op}$)
18 do$_{OP}$(X$_s$, -X$_{op}$)
19 cannotReslove
20 alert$_{OP}$
AuthUML Predicate
Phase 1
Phase 2
Phase 3
FlowUML Predicate
FlowSup predicates
ConstSup predicates
Flow$_{att}$
mayFlow$_{att}$
mayFlow$_{interUC\ att}$
finalFlow$_{att}$
finalFlow$_{interUC\ att}$
unsafeFlow$_{att}$
unsafeFlows$_{att}$
safeFlow$_{att}$

Figure 8.5. AuthUML and FlowUML Collaboration Predicates

8.2.2 The Collaboration Process in Detail

8.2.2.1 Use Case Development

Before any analysis, I assume that the use cases are already developed for the system that will be analyzed, but at this level the use case can be in an abstract form where only the use case objective and the authorized actors are defined. Figure 8.1 shows an example of a use case in such an abstract view.

8.2.2.2 Applying AuthUML at the Use Case Level

Because only the abstract use cases are developed at this time, only the first and second phases of AuthUML are applied. This stage analyzes the authorization requirement of the use case and ensures that they are consistent, complete and free of application-specified conflicts. The finalized authorizations produced by this stage are in the form of $do_{uc}(X_s, \pm X_{uc})$. This outcome will be used in the FlowUML to validate the correctness of the sequence diagram from the point of view of authorization of information flow.

8.2.2.3 Sequence Diagram Development

After the use cases are developed at the abstract level, the next step is to develop the details of each use case by identifying the operations and the information flow between them. Such details are represented by one of the interaction diagrams type. For my work, I choose sequence diagrams. Figure 8.2 shows an example of a sequence diagram of a *Prepare order* use case. Every use case in Figure 8.1 is represented in a sequence diagram that shows all operations of a use case and how they interact with objects.

8.2.2.4 Applying FlowUML

FlowUML extracts the information flow between objects within a use case and between use cases to validate the enforcement of information flow control policies and to ensure that every flow is safe. Also, FlowUML identifies all operations that are part of a use case. Note that at this stage both coarse and fine-grain policies apply. However, because the third phase of AuthUML addresses the operation level access content, I choose to demonstrate only the analysis of fine-grain policies of FlowUML.

After extracting the information flow from the sequence diagram, the original FlowUML needs to detect all derived (implicit) authorizations of all actors from the actor hierarchy. However, because AuthUML has already derived those authorizations at the second stage I can derive the inherited authorized information flow by using the finalized authorization predicates produced by AuthUML. The following rules show how to derive inherited information flow using AuthUML predicates.

$$\mathrm{Flow}_{att}(Xs, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \leftarrow \quad do_{UC}(X_s, +X_{uc}), \; \mathrm{Flow}_{att}(X'_s, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \tag{7}$$

$$\mathrm{Flow}_{att}(X_s, X_{att}, X_s, X'_{obj}, X_t, X_{uc}, X_{op}) \leftarrow do_{UC}(X_s, +X_{uc}), \; in(X_s, X''_s) \; \mathrm{Flow}_{att}(X'_s, X_{att}, X''_s, X'_{obj}, X_t, X_{uc}, X_{op}) \tag{8}$$

$$\mathrm{Flow}_{att}(X_s, X_{att}, X_{obj}, X_s, X_t, X_{uc}, X_{op}) \leftarrow do_{UC}(X_s, +X_{uc}), \; in(Xs, X''_s) \; \mathrm{Flow}_{att}(X'_s, X_{att}, X_{obj}, X''_s, X_t, X_{uc}, X_{op}) \tag{9}$$

Note that the inherited authorization was first derived during the first phase of AuthUML then finalized in the form of $do_{UC}(X_s, +X_{uc})$ at the end of that phase.

The previous rule states that, if there is an actor who is authorized to invoke a use case and there is a flow of an operation that is part of the same use case and it is initiated by any actor (not necessary the same actor of the use case), there will be a new flow predicate that states that the authorized actor of the use case will be authorized to initiate the same flow of that operation. For example, the *Prepare order* use case in Figure 8.1 has several operations as shown in Figure 8.2. The operation *read(att1)* is extracted as:

$Flow_{att}(Clerk, att_1, Clerk, Obj_1, 1, \text{"Prepare order"}, read)$ (10)

Given that the Purchasing officer has an implicit authorization to invoke the *Prepare order* because s/he is a specialized actor of Clerk, AuthUML authorization is written as:

$do_{UC}(\text{"Purchasing officer"}, +\ \text{"Prepare order"})$ (11)

Thus, by combining predicates 10 and 11, the Purchasing officer will also be authorized to initiate the *read* operation that is written as:

$$Flow_{att}(\text{"Purchasing officer"}, att_1, \text{"Purchasing officer"}, Obj_1, 1, \text{"Prepare order"}, read) \quad (12)$$

From the collaboration prospective, FlowUML identifies all operations of the use case and presents it as an instance of the $Flow_{att}(X_s, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op})$. FlowUML provides data to AuthUML that allows the validation of more access control policies, that were not available without FlowUML. I show some examples as follows:

Access Control List (ACL) of each operation: It enumerates all allowed actors who can access a particular operation.

$$ACL(X_a, X_{op}) \leftarrow Flow_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{op})$$
$$ACL(X_a, X_{op}) \leftarrow Flow_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{op}), isActor(X_{obj}) \quad (13)$$

All information received by an actor: It lists all attributes that may flow to a particular actor. For example, it states whether an actor may receive the trade secret attribute.

$$recList(X_a, X_{att}) \leftarrow finalFlow_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t), isActor(X'_{obj}) \quad (14)$$

All information sent by an actor: It lists all attributes that an actor may send. For example, it states whether an actor may write/modify the salary.

$$sentList(Xa, XattXobj) \leftarrow finalFlowatt(Xa, Xatt, Xobj, X'obj, Xt), isActor(Xobj) \quad (15)$$

8.2.2.5 Applying AuthUML at the Operation Level

At this stage, AuthUML is applied again, but at the operation level. Although, the flow between objects is safe after applying FlowUML, the derived information may violate the operation level authorizations. Thus, AuthUML is applied at the last phase to ensure consistent, complete and conflict-free authorizations at the operations level.

Because transformation from FlowUML to AuthUML (phase 3) provides all the operations of use cases, the transformation eliminates the first step in phase 3 of the original AuthUML that derives authorizations of all operations of a use cases. This transformation follows rule 16.

$$\mathrm{dercando}_{OP}(X_a, +X_{op}) \leftarrow \mathrm{Flow}_{att}\,(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}) \quad (16)$$

The previous rule says that, if there is a flow that is a part of an operation *op* and it is initiated by an actor *A*, there is a positive authorization of actor *A* to invoke *op*. For example, consider the *Prepare order* use case which was shown in Figure 8.2 and written in FlowUML in the form of $\mathrm{Flow}_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op})$ predicate as in 6. Note that the Flow_{att} predicate specifies the use case and the operations that are affected by the flow. Thus, from that attribute of Flow_{att} , I can deduce the operations of use cases. The following are the results of applying rule 16 based on the predicates in 6.

$$\begin{aligned}
&\mathrm{dercando}_{OP}(\mathrm{Clerk}, \mathrm{read}) \leftarrow \mathrm{Flow}_{att}(\mathrm{Clerk}, \mathrm{att}_1, \mathrm{Clerk}, \mathrm{Obj}_1, 1, \\
&\qquad\qquad \text{“Prepare order”}, \mathrm{read}) \\
&\mathrm{dercando}_{OP}(\mathrm{Clerk}, \mathrm{write}) \leftarrow \mathrm{Flow}_{att}(\mathrm{Clerk}, \mathrm{att}_4, \mathrm{Clerk}, \mathrm{control}, \\
&\qquad\qquad 3, \text{“Prepare order”}, \mathrm{write}) \\
&\mathrm{dercando}_{OP}(\mathrm{Clerk}, \mathrm{call}) \leftarrow \mathrm{Flow}_{att}(\mathrm{Clerk}, \mathrm{att}_5, \mathrm{control}, \\
&\qquad\qquad \mathrm{Inventory\ staff}, 4, \text{“Prepare order”}, \mathrm{call})
\end{aligned} \quad (17)$$

In addition, rule 16 will be applied to any derived information flow. In the following rule, the body is already derived from rules 7, 8 and 9:

$$\text{dercando}_{OP}(\text{Purchasing officer, read}) \leftarrow \text{Flow}_{att}(\text{Purchasing officer}, \text{att}_1, \text{Clerk}, \text{Obj}_1, 1, \text{“Prepare order”}, \text{read}) \quad (18)$$

AuthUML considers positive and negative authorization, while FlowUML considers only positive flows. The previous rule transforms only the positive authorization. However, a workaround is possible to derive negative authorization to operations, as shown in rule 19. The rule says that for every flow that is part of an operation and where that operation is part of a use case and there is an actor who has a negative authorization to that use case, I can deduce that there is a negative authorization for the same actor to invoke the operation. In another words, I refer to the flow predicate merely to know which operation is part of the use case that has a negative authorization for that actor. For example, in the running example in Figure 8.1, there is a negative authorization for Manager to invoke use case *Write check*. This negative authorization is written as $\text{do}_{UC}(\text{Manager},-\text{“Write check”})$. Suppose there is a flow inside that use case written as $\text{Flow}_{att}(\text{Clerk}, \text{att}_1, \text{Obj}_1, \text{Obj}_2, 1, \text{“Write check”}, \text{issue})$. Thus, I can deduce from the two available predicates that the Manager cannot invoke operation *issue*.

$$\text{dercando}_{OP}(X'_a,-X_{op}) \leftarrow \text{Flow}_{att}(X_a, X_{att}, X_{obj}, X'_{obj}, X_t, X_{uc}, X_{op}), \text{do}_{UC}(X'_a,-X_{uc}) \quad (19)$$

8.3 The Flexibility of AuthUML and FlowUML

Both AuthUML and FlowUML analyze the requirement specification and ensure the proper compliance with both access and flow control policies. However, this is not the total advantage of both frameworks because they are also designed with flexibility in mind. The flexibility is achieved in three ways. First, by transforming UML geometries (use case and sequence diagram) into a set of logic-based predicates, the predicates provide flexibility because they are potentially amenable to automated reasoning that is useful in analysis [Rus01] [NE00]. Second, by transforming the policies from text–based to a set of Prolog-style rules, this allows the enforcement of policies and the validation of requirements against policies to be applicable and less complicated. Third, flexibility is achieved by isolating the requirement specifications – that are written in predicates – from the policies – that is written in rules. The isolation is the core flexibility of the frameworks that allows the application of different polices for each portion of the requirements.

Figure **8.6.** Illustration of the isolations and flexibly of the frameworks. The figure consists of three columns. The first column represents the geometry of UML that is used in my frameworks (the use case and the sequence diagram). First, each abstract use case is instantiated by a sequence diagram to illustrate the information flow of the use case. Second, the sequence diagram can be used to identify the required operations of each use case.

In the second column, it transforms the requirements expressed in UML into predicates. First, it transforms the authorization of each use case or group of use cases into AuthUML predicates. Such transformed authorization can be separated into groups or combined together; groups are desired to apply different policies or to validate different policies to each group. For example, one can apply strict policies on sensitive use cases, while applying less restrictive policies on public-use use cases. Second, the sequence diagram of each use case is transformed into FlowUML predicates. The same applies here concerning whether to separate each use case flow or to combine all them together to create a broader analysis. Third, the authorization of operations is transformed to AuthUML predicates. Also, the decision of combining predicates can be made here.

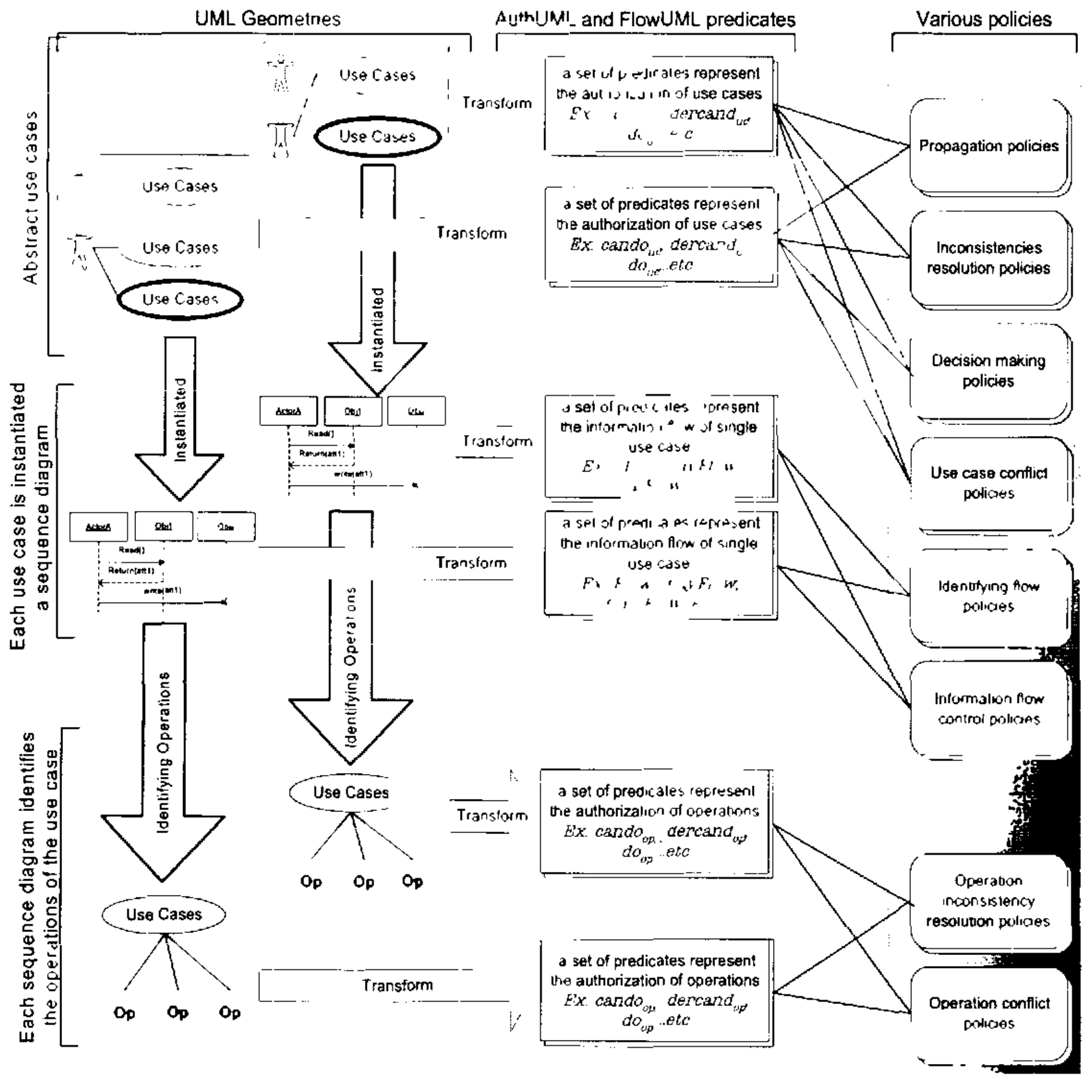

UML Geometries
AuthUML and FlowUML predicates
Various policies
Abstract use cases
Each use case is instantiated a sequence diagram
Each sequence diagram identifies the operations of the use case
Use Cases
Transform
Instantiated
Identifying Operations
Op
a set of predicates represent the authorization of use cases
Ex. cando, dercand, do, etc
a set of predicates represent the information flow of single use case
a set of predicates represent the authorization of operations
Propagation policies
Inconsistencies resolution policies
Decision making policies
Use case conflict policies
Identifying flow policies
Information flow control policies
Operation inconsistency resolution policies
Operation conflict policies

Figure 8.6. The Comprehensive Scope of Applying AuthUML and FlowUML

The third column of Figure 8.6, shows the various policies that are written in Prolog-style rules. As shown in the figure, a policy can be applied to any portion of the requirements that are specified in predicates in the second column. Also, each portion of the requirements can be enforced or validated by different policies. For example, one can apply the *negative takes precedence* policy in case an inconsistency occurs in just one portion, while applying another policy for other requirements. Note that the policies in the third column consist of two types. The first type will be used to validate the compliance of requirements, e.g., Mandatory Access Control (MAC) or Discretionary Access Control (DAC) policies. The second type is enforced in the analysis process, e.g., whether to apply the *permission take precedence* or the *negative takes precedence*.

This separation of policy from the application has many consequences. The first consequence is that it facilitates applying any policy to any design. The second consequence is that the same process can be used to check the consistency of two design diagrams with respect to a given security policy. That is, if two design diagrams are compliant with a given policy, as far as that policy is concerned, they are indistinguishable. I developed this concept further in designing *a notion of policy-based equivalence* of design diagrams in UML. The third consequence is that, if UML policies can be separated from designs as shown here, a policy composition framework for UML along the lines of [BCVS00],[WJ02] can be developed. The last consequence is that, by capturing more rules related to geometry of sequence diagrams, one may be able to capture deficiencies in the diagrams. If successful, this may lead to a policybased, reverse engineering framework for UML diagrams.

8.4 Conclusion

In this chapter, I showed how access and flow control policies can be verified during the software development process. I showed how to improve the analysis of both policies by combing two existing frameworks (AuthUML and FlowUML). Such collaboration provides a more accurate analysis of access and flow control policies. FlowUML provides rich information about the authorizations details that are provided to AuthUML and were unavailable without FlowUML. Also, AuthUML analyzed abstract authorizations before analyzing the information flow, and analyzed the details of authorizations after the information flow is analyzed by FlowUML. I defined the process of collaborating both AuthUML and FlowUML and the necessary rules to transform the output of each framework to the other. I also showed how those two frameworks can provide the flexibility and scalability to enforce security policies.

Chapter 9

CONCLUSION

This chapter summarizes my contributions, in section 9.1, and possible future extensions of this dissertation, in section 9.2.

9.1 Conclusion

Security features must be considered with other software requirements during the software development process to improve the security of the software and to reduce the cost of patching vulnerabilities. This dissertation focused on integrating both access and flow control policies during the requirements specification analysis and design phases of the software development process. The two main themes of this dissertation are, first, to provide unified representation and specification of both access and flow control policies and, second, to provide frameworks to verify the compliance of access and flow control requirements with the access and flow control policies.

To provide a unified representation of access and flow control policies, I extended the de facto standard language of modeling, the UML. The advantage to extend the UML is to ease the adoption of the extension by the software developers. The extension provides the necessary elements and a methodology to verify and design both access and flow control policies correctly. The extension focuses on static and dynamic access and flow control policies, where other work focuses on static policies only. I believe that the extension encourages the software analyzer to integrate access and flow control policies with other functional requirements. I have shown how the new extension specifies and models existing access and flow control models, such as Role-based Access Control (RBAC), Information Flow Control for Object-Oriented Systems, and Distributed Authorization Processor.

To provide formal verification and detection of improper access and flow control requirements, I developed two frameworks that are based on logic programming. The first framework, AuthUML, verifies that the access control requirements are consistent, complete, and conflict-free. AuthUML allows the analyzer to work at two different levels of detail, the Use Case level and the operation level, each where it is useful for the analyzer.. The second framework I developed, FlowUML, verifies the proper enforcement of information flow control policies at the requirements specification phase of UML-based designs. Also, FlowUML provides the same flexibility as AuthUML by providing two levels of analysis. I showed how the two frameworks can be used to verify existing access and flow control policies, such as Separation of Duty principle, Mandatory Access Control (MAC), and RBAC.

The final contribution was the integration of AuthUML and FlowUML to form one framework. The integration combines the strengths of both frameworks to improve analysis and detection violations of access and flow control policies.

I believe that the usage of the frameworks will improve the enforcement of access and flow control policies because such frameworks detect violations of policies at the early phases of the software development process and limit them from propagating to other phases, where the cost of detecting and solving them is amplified.

9.2 Future Work

As stated it on the section 9.1, the need to integrate security into software development is a large-scope goal. It requires the integration of different features of security such as access control policies, flow control, privacy, encryption, and availability. Also, it requires the integration of those features in all phases of the software life cycle such as the requirements specification, analysis, design, implementation, and testing.

I believe this dissertation has met the goal of integrating two security features: access and flow control policies. Also, it has met the goal of integrating those policies at the first two phases of the software life cycle: requirements specification and analysis.

In general, there is a room for future research, for example integrating other security features during all phases of the software life cycle, or integrating access and flow control policies with different phases.

Other UML diagrams, such as the state chart and deployment diagram, have not been studied in this dissertation. Also, detailed analysis and representation of the separation of duty principle with regards to software development is another research area, because most research in the of separation of duty principle has focused on the system point of view rather than the software point.

ABSTRACT

INCORPORATING ACCESS AND FLOW CONTROL POLICIES IN REQUIREMENTS ENGINEERING

Khaled Alghathbar, Ph.D.

George Mason University, 2004

Dissertation Director: Dr. Duminda Wijesekera

Access and flow control policies have not been well integrated into functional specifications throughout the software development life cycle. Access and flow control policies, and security in general, are generally considered to be non-functional requirements that are difficult to express, analyze, and test. Accordingly, most security requirements are considered after the analysis of the functional requirements. Ignoring non-functional requirements or paying less attention to them during the early development process results in low-quality, inconsistent software, dissatisfied stakeholders, and extra time and cost to re-engineer. Therefore, integrating security with other functional requirements as early as possible during the software life cycle improves the security of the software and reduces the cost of maintenance.

The main focus of this dissertation is to incorporate both access and flow control policies with other functional requirements during the requirements specification and analysis

phases of the software development life cycle. I have developed a unified representation language and formal verification frameworks for both access and flow control policies.

As the Unified Modeling Language (UML) is the de facto standard modeling language, I extended it with the necessary elements to represent access and flow control policies. This extension allows software developers to model access and flow control policies in a unified way. The advantage of extending UML to incorporate access and flow control policies is easing its adoption by software developers. This extension addresses what others have not addressed, such as the representation and modeling of dynamic access and flow control policies, negative authorizations, and inherited authorizations.

In large software systems, there are a large number of access and flow control policies to enforce; thus, inconsistent, incomplete, and conflicting sets of policies may be specified. Therefore, there is need for a formal and automated language and tools to detect problems due to improper policies. For the analysis of access control policies, I developed AuthUML, a framework, based on logic programming, that analyzes access control requirements in the requirements phase to ensure that they are consistent, complete, and conflict-free. The framework is a customized version of Flexible Access Framework (FAF) of Jajodia et al. and it is suitable for UML-based requirements engineering. It analyzes access control policies at two different levels: Use Cases and conceptual operations.

For the analysis of information flow control policies, I developed FlowUML, a logic-based system that verifies the proper enforcement of information flow control policies at the requirements specification phase of UML-based designs. FlowUML uses logic programming to verity the compliance of information flow control requirements with information flow polices. FlowUML policies can be written at a coarse-grain level or in a finer-grain level; these two levels provide a comprehensive and wide application of policies.

Finally, because of the overlap of access and flow control policies, I integrated the analysis of both policies into one framework that reduces redundant process, provides more useful analysis information, and improves overall analysis in general.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Incorporating Access and Flow Control Policies in Requirements Engineering

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University.

By

Khaled S. Alghathbar

B.S. King Saud University, Saudi Arabia, June 1998
M.S. George Mason University, Fairfax, VA, August 2001

Director: Dr. Duminda Wijesekera
Information and Software Engineering

Spring Semester 2004
George Mason University
Fairfax, VA

ABSTRACT

INCORPORATING ACCESS AND FLOW CONTROL POLICIES IN REQUIREMENTS ENGINEERING

Khaled Alghathbar, Ph.D.

George Mason University, 2004

Dissertation Director: Dr. Duminda Wijesekera

Access and flow control policies have not been well integrated into functional specifications throughout the software development life cycle. Access and flow control policies, and security in general, are generally considered to be non-functional requirements that are difficult to express, analyze, and test. Accordingly, most security requirements are considered after the analysis of the functional requirements. Ignoring non-functional requirements or paying less attention to them during the early development process results in low-quality, inconsistent software, dissatisfied stakeholders, and extra time and cost to re-engineer. Therefore, integrating security with other functional requirements as early as possible during the software life cycle improves the security of the software and reduces the cost of maintenance.

The main focus of this dissertation is to incorporate both access and flow control policies with other functional requirements during the requirements specification and analysis

phases of the software development life cycle. I have developed a unified representation language and formal verification frameworks for both access and flow control policies.

As the Unified Modeling Language (UML) is the de facto standard modeling language, I extended it with the necessary elements to represent access and flow control policies. This extension allows software developers to model access and flow control policies in a unified way. The advantage of extending UML to incorporate access and flow control policies is easing its adoption by software developers. This extension addresses what others have not addressed, such as the representation and modeling of dynamic access and flow control policies, negative authorizations, and inherited authorizations.

In large software systems, there are a large number of access and flow control policies to enforce; thus, inconsistent, incomplete, and conflicting sets of policies may be specified. Therefore, there is need for a formal and automated language and tools to detect problems due to improper policies. For the analysis of access control policies, I developed AuthUML, a framework, based on logic programming, that analyzes access control requirements in the requirements phase to ensure that they are consistent, complete, and conflict-free. The framework is a customized version of Flexible Access Framework (FAF) of Jajodia et al. and it is suitable for UML-based requirements engineering. It analyzes access control policies at two different levels: Use Cases and conceptual operations.

For the analysis of information flow control policies, I developed FlowUML, a logic-based system that verifies the proper enforcement of information flow control policies at the requirements specification phase of UML-based designs. FlowUML uses logic programming to verity the compliance of information flow control requirements with information flow polices. FlowUML policies can be written at a coarse-grain level or in a finer-grain level; these two levels provide a comprehensive and wide application of policies.

Finally, because of the overlap of access and flow control policies, I integrated the analysis of both policies into one framework that reduces redundant process, provides more useful analysis information, and improves overall analysis in general.

INCORPORATING ACCESS AND FLOW CONTROL POLICIES IN REQUIREMENTS ENGINEERING

by

Khaled S. Alghathbar

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

[illegible]	Dr. Duminda Wijesekera, Dissertation Director
[illegible]	Dr. Edgar Sibley, Committee Chairman
[illegible]	Dr. David Schum
[illegible]	Dr. Francesco Parisi-Presicce
[illegible]	Dr. Stephen G. Nash, Associate Dean for Graduate Studies and Research
[illegible]	Dr. Lloyd J. Griffiths, Dean, School of Information Technology and Engineering

Date: ____________________

Spring Semester 2004
George Mason University
Fairfax, VA

<u>INCORPORATING ACCESS AND FLOW CONTROL POLICIES IN REQUIREMENTS ENGINEERING</u>

by

Khaled S. Alghathbar

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

________________________	Dr. Duminda Wijesekera, Dissertation Director
________________________	Dr. Edgar Sibley, Committee Chairman
________________________	Dr. David Schum
________________________	Dr. Francesco Parisi-Presicce
________________________	Dr. Stephen G. Nash, Associate Dean for Graduate Studies and Research
________________________	Dr. Lloyd J. Griffiths, Dean, School of Information Technology and Engineering

Date: ____________________ Spring Semester 2004
George Mason University
Fairfax, VA

DEDICATION

To my parents, my wife and my son for their constant love, patience, sacrifices, support and encouragement that made this great accomplishment possible.

ACKNOWLEDGEMENTS

Prophet Mohammed (Peace Be Upon Him) said: "Whoever does not give thanks to the people does not give thanks to Allah "

First and foremost, I would like to express my deepest gratitude to Allah for his inspiration and guidance to achieve this work.

I would like to express my sincere gratitude and appreciation to my dissertation director Professor Duminda Wijesekera, for his unlimited support and advising. He enlightened broadways to thorough thinking and new bodies of knowledge.

Also, I convey my appreciation to Professor Edger Sibley to be the chairman of my doctoral committee and for his invaluable guidance and encouragement. The thanks extend to the members of my dissertation committee, Professor David Schum and Professor Francesco Parisi-Presicce, for all their invaluable comments and suggestions.

My warmest appreciations for my wife and son here in the United States and to my parents, brothers, sisters, nephews, nieces and friends back in Saudi Arabia for all their prayers, patients, support and love that comforted me during my studies abroad.

Thanks to all my Saudi colleagues in GMU for their support and encouragement through out my master and doctorate degrees.

I acknowledge the financial support of Kind Saud University of Saudi Arabia during my higher education.

- To everyone who helped me directly or indirectly -

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Problem Statement

Today, most security requirements, such as access and flow control policies, are considered only after the completion of functional requirements because security requirements are considered non-functional requirements, which are difficult to express, to analyze, and to test, and because languages used to specify access and flow control policies (such as FAF [JSSS01], PERMIS [CO02], Author-X [BBC+00] and PDL [LBN99]) are separate from the languages used to model functional requirements (such as UML) during the software development life cycle. Consequently, security considerations may not be properly engineered during the software development life cycle, and less secure systems may result.

1.2 Thesis Statement

Devanbu and Stubblebine [DS00] challenged the academic community to adopt and extend standard modeling languages such as UML to include security-related features. I accepted this challenge by showing that:

- It is possible to incorporate access and flow control policies with other functional requirements during the early phases of the software development life cycle by extending the Unified Modeling Language (UML) to include security features as first class citizens.
- It is possible to develop tools that help software analysts and designers to verify the compliance of the access and flow control requirements with with policy before proceeding to other phases of the software development process.

I substantiated my claim by:

1. Extending the metamodel of UML to incorporate access and flow control policies in the design.
2. Enhancing and extending the Use Case model by providing a unified specification of access and flow control policies using object constraint language (OCL).
3. Designing a formal framework to detect inconsistency, incompleteness, and application-definable conflict among access control policies.
4. Designing a formal framework that verifies the compliance of information flow requirements with information flow control policies.
5. Integrating both frameworks to analyze both access and flow control policies at the same time.

1.3 Significance of Contributions

Access and flow control policies in software security have not been well integrated with functional specifications during the requirements engineering and modeling phases of the software development life cycle. Security is considered to be a non-functional requirement (NFR) [CNY+00]. Such requirements are difficult to express, analyze, and test; therefore, they are usually evaluated subjectively. Because NFRs tend to be properties of a system as a whole [CNY+00, NE00],, most security requirements are considered after the analysis of the functional requirements [DS00]. The consequences of ignoring NFR are low-quality and inconsistent software, unsatisfied stakeholders, and more time and cost in re-engineering [CNY+00]. Therefore, integrating security into the software life cycle throughout all its phases adds value to the outcome of the process.

It is important to specify access control policies precisely and in sufficient detail, because ambiguities in requirements specifications can result in erroneous software [GW89]. In addition, careful consideration of requirements – including NFRs – will result in reducing project cost and time, because errors that are not detected early can propagate into the other phases of the software development life cycle, where the cost of detection and removal is high [DS00] [Boe81]. By analyzing large projects in IBM, GTE, and TRW, Boehm [Boe81] computed the cost of removing errors in general made during the various phases of the development life cycle, as shown in table 1.

Table 1: Relative Cost to Correct an Error

Phase where the error is found	Cost ratio
Requirements	1
Design	3-6
Code	10
Development test	15-35
Acceptance test	40-75
Operation	30-1000

In UML-based software design methodologies, requirements are specified using Use Cases at the beginning of the life cycle. Use Cases specify actors and their intended usage of the envisioned system. Nevertheless, a Use Case is written in natural language, which lacks the precision and specification of security [DS00]. Therefore, there is a need to provide a unified language for representing security features, such as access and flow control policies [DS00, CNY+00], in the early phases of the software development life cycle. This language must allow software developers to model access control policies in a unified way and it must be compatible with other requirements modeling languages.

In addition, there is a need to verify the compliances of security requirements with the security policies before proceeding to other phases of the software development life cycle [NE00, Pfl98, Rus01]. I used Logic as the underlying language because it is potentially amenable to automated reasoning [NE00, Rus01].

My dissertation partially fulfills Devanbu and Stubblebine's challenge [DS00], because totally satisfying their requirement has to consider all aspects of security in all phases of the software development life cycles. My contributions meet the challenge in the requirements, analysis and design phases only by specifying and verifying access and flow control policies there.

1.4 Summary of Contributions

My dissertation introduced several contributions that assist software developers to specify and analyze access and flow control policies during the first three phases of the software development process, requirement specification, analysis, and design phases. The following summarize my contributions:

6. I extended the UML Metamodel in a way that allows systems designers to model dynamic and static access control policies as well as flow control policies in a unified way. The extension provides a better way to integrate and impose authorization policies on commercial off-the-shelf (COTS) and mobile code. I showed how this extension allows non-security experts to represent access control models, such as Role-Based Access Control (RBAC) and workflow policies, in an uncomplicated manner.
7. I extended the Use Case model to specify access control policies precisely and unambiguously with sufficient details in the UML's Use Case. I added to Use Cases by using something analogous to operation schemas [SS00], which I called *access control policy schemas*. The extension employs the Object Constraint Language (OCL) [OCL01], which is more formal than the existing Use Case language (natural language) for specifying access and flow control policies.

8. I developed a framework called AuthUML that formally verifies the compliance of access control requirements with the access control policies during the requirement specification and analysis phases using Prolog style stratified logic programming.
9. I developed a framework called FlowUML to verify the proper enforcement of information flow control policies on the requirements.
10. I incorporated the analysis of both access and flow control requirements by integrating both AuthUML and FlowUML. The incorporation of both frameworks improves the analysis and detection of improper access and flow control requirement.

Based on my work in this dissertation I published several papers.

1.5 Organization of the Dissertation

Chapter 2 summarizes the literature that is related to my work, it also analyzes and compares the work with what I presented in this dissertation. Chapter 3 summarizes background works that are used as bases for my extensions, such as the UML, FAF and Operation Schemas. Chapter 4 presents the extension of the UML Metamodel to design access and flow control policies, and it shows the application of the extension on different existing access control models. Chapter 5 presents the extension of the Use Case to formally specify access and flow control requirements, and it shows the extension of the Use Case diagram and how to analyze the access control requirements visually. Chapter 6 introduces AuthUML, a framework to verify and detect improper access control requirements. Chapter 7 presents the FlowUML, a framework that analyzes information flow control requirement and detects violation of information flow control policies. Chapter 8 incorporates the analysis of both AuthUML and FlowUML and produces a coherent framework to verify both access and flow control requirements. Finally, summary of my contributions and discussion of future research are presented in chapter 9.

Chapter 2

LITERATURE REVIEW

Several new papers have been published in this area; those works concentrate on different aspects of security features and software development phases. However, there are some drawbacks in those works that need to be improved; further, some additional issues need to be addressed.

There are several aspects of security that need to be integrated into the software development process such as access control policies, flow control policies, authentication, integrity, and encryptions. Likewise, there are different phases of software development such as requirements specification, analysis, design, implementation, and testing, that require security to be integrated with them for better secure software systems.

In this dissertation, I have focused on five aspects of integrating access and flow control policies during requirement engineering. First, I extended the UML metamodel to allow the proper specification of access and flow control policies. Second, I extended the Use Case model to formally specify access and flow control policies. Third, I developed a framework to verify the access control requirements. Fourth, I developed a framework to verify the flow control requirements. Both frameworks detect improper access and flow control requirements as early as possible during the software process. The following sections summarize the literatures related to each aspect.

2.1 Extending the UML Metamodel and Use Case Model

Lodderstedt *et al.* [LBD02] proposed a methodology to model access control policies and integrate them into a model-driven software development process. The work was based on RBAC as a security model. My work is differs from [LBD02] by concentrating on specifying dynamic access control policies (e.g. dynamic separation of duty) and workflow as well as static access control policies. Furthermore, I focused on dynamic design modeling while Lodderstedt's focus was on static design model. Also, my prospective view of enforcing constraints is from the flow view not from the static view. There are several issues missing from the work of Lodderstedt *et al.* First, history-related constraints cannot be modeled with Lodderstedt's method. Second, the metamodel is not flexible enough to model all access control policies, because it is based on RBAC only. Third, the metamodel cannot restrict people in senior roles from performing certain junior operations and it cannot specify conflict among users, operations, or roles.

Fernandez-Medina *et al.* [FPS01] introduced a language called Object Security constraint Language (OSCL). OSCL extends the Object Constraint Language (OCL) [WK99] to specify security constraints to represent multi-level security systems. Also, Fernandez-Medina *et al.* in [FMM+02] proposed an extension to the Use Case and Class models of the UML. The extensions of Use Case diagram which they introduced were stereotypes: <<safe-UC>> and <<accredited -actor>> as an indication of a secure Use Case and authorized actor. Their work is focused on database security and shows how to model multilevel security on the static diagram such as Class diagram by introducing tagged values to classes, attributes, operations, and association ends where those tagged values indicate the security level of the element. However, this extension did not represent dynamic authorization and workflow policies. Also, the extension was limited to multilevel security model. Finally, the extension did not address the type of authorization that is granted to the accredited actor, nor the integrity constraints associated with such authorizations.

Brose *et al.* [BKL02] extended the UML to support the automatic generation of the access control policies to configure a CORBA-based infrastructure for view-based access control. It stated permissions and prohibitions of accessing system's objects (read, write, execute...etc) explicitly by writing notes that are attached to actors in the Use Case diagrams. However, their work was based on static specification of access policies but it could not model dynamic access control policies such as *Dynamic Separation of Duty* nor it could enforce some flow requirement such as the order of operations in a specific workflow systems. Although, that work covered most parts of the software development life cycle, it did not integrate access control policies in the interaction diagrams such as the Sequence diagram, and that what I presented in this dissertation. In addition, the specification language of that work was natural language which is imprecise. Therefore, I used the OCL to specify the constraints more precisely. Finally, that work considered role hierarchies, but no propagation or conflict resolution policies have been addressed for the inherited authorizations.

BIBLIOGRAPHY

[ABW88] K. Apt, H. Blair, A. Walker. Towards a theory of declarative knowledge. *In J. Minker, editor, Foundations of deductive databases*, pages 89-148. Morgan Kaufmann, San Mateo, 1988.

[AS00] G.-J. Ahn and R. Sandhu, Role-based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, pages *207-226, Vol. 3, No. 4*, November 2000.

[AS01] G.-J. Ahn, M. Shin. Role-Based Authorization Constraints Specification Using Object Constraint Language. *In the proceedings of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 20 - 22, 2001 Massachusetts.

[AW03] K. Alghathbar, D. Wijesekera. Extending the UML To Model Dynamic Authorization Policies. *In proc. of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03)*, Rio de Janeiro, Brazil. June 5-7, 2003.

[AgW03] K. Alghathbar, D. Wijesekera. *Extending the UML To Model Dynamic Authorization Policies*. In proc. of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03), Rio de Janeiro, Brazil. June 5-7, 2003.

[BA99] E. Bertino and V. Atluri. The specification and enforcement of authorization constraints in workflow management. *ACM transactions on Information Systems Security*, February 1999.

[BBC+00] E. Bertino, M. Braun, S. Castano, E. Ferrari, and M. Mesiti. Author-x: A javabased system for XML data protection. *In IFIP Workshop on Database Security*, pages 15-26, 2000.

[BCVS00] P. Bonatti, S. De Capitani di Vimercati, P. Samarati. A Modular Approach to Composing Access Control Policies. Proc. of the Seventh ACM

Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000.

[BKL02] G. Brose, M. Koch, K.-P. Löhr. Integrating Access Control Design into the Software Development Process. *In the Proceedings of the sixth biennial world conference on the Integrated Design and Process Technology (IDPT)*, Pasadena, CA. June 2002.

[BL75] D. Bell and L. LaPadula. Secure computer system: United exposition and Multics interpretation. *Technical Report, ESD-TR-75-306, MITRE Corp.* MTR-2997. Bedford, MA, 1975.

[Boe81] B. Boehm. *Software engineering economics*. Englewood Cliffs, NJ: Prentice Hall. 1981.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[Bro00] G. Brose. A typed access control model for CORBA. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Weidner, editors, *Proc. European Symposium on Research in Computer Security (ESORICS),* LNCS 1895, pages 88-105. Springer, 2000.

[CNY+00] L. Chung, B. Nixon, E. Yu, J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers (2000).

[CO02] D. Chadwick, A. Otenko. The PERMIS X.509 Role Based Privilege Management Infrastructure. *In the Proceedings of the 7th Acm Symposium On Access Control Models And Technologies (SACMAT 2002).* Montrerey, California USA. 3-4 June 2002.

[CW87] D. D. Clark, D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. *In the proc. IEEE Symposium on Security and Privacy*. 1987.

[CWJ03] S. Chen, D. Wijesekera, S. Jajodia. FlexFlow: A Flexible Flow Control Policy Specification Framework. *In proceedings of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*. Estes Park, Colorado. August 2003.

[DM89] J. Dobson, J. McDermid: A Framework for Expressing Models of Security Policy. *In the proc. IEEE Symposium on Security and Privacy*. 1989.

[DS00] P. T. Devanbu and S. Stubblebine. Software engineering for security:A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[FH97] E. B. Fernandez and J. C. Hawkins, Determining role rights from use cases, *In the Procs. 2nd. ACM Workshop on Role-Based Access Control*, November 1997, 121-125.

[FHG97] D. Firesmith, B. Henderson-Sellers, I. Graham, *OPEN Modeling Language (OML) Reference Manual*. SIGS Books. 1997.

[FMM+02] E. Fernandez-Medina, A. Martinez, C Medina, And M. Piattini. Integrating Multilevel Security in the Database Design Process. *In the Proceedings of the sixth biennial world conference on the Integrated Design and Process Technology (IDPT)*, Pasadena, CA. June 2002.

[FPS01] E. Fernadez-Medina, M.G. Piattini, M.A Serrano. Specification of Security Constraints in UML. *In the 35th International Carnahan Conference on Security Technology (ICCST)*, London, UK, October 2001.

[FS99] M. Fowler, K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (2nd Edition)*, Addison-Wesley, 1999.

[FSBJ97] E. Ferrari, P. Samarati, E. Bertino, S. Jajodia. Providing Flexibility in Information Flow Control for Object-Oriented Systems. *Proc. IEEE Symp. on Research in Security and Privacy,* Oakland, Calif., May 1997, pp. 130-140

[GH91] D. Gabbay and A. Hunter. Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Phase1 - A Position Chapter, *Proceedings of Fundamentals of Artificial Intelligence Research '91*, 19-32, Springer-Verlag.

[GH92] D. Gabbay and A. Hunter. Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Phase2. *In Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, 129-136, LNCS, Springer-Verlag, 1992.

[GL88] M. Gelfond, V. Lifschitz. The stable model semantics for logic programming. *In Proceedings, 5th International Conference and Symposium on Logic Programming* Seattle, Wash. pp. 1070–1080. 1988

[Gom00] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison Wesley, 2000

[GW89] D. Gause, G. Weinberg. *Exploring Requirements: Quality Before Design*, Dorset House, New York, NY (1989).

[JCJO92] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaad. *Object-Oriented Software Engineering A Use Case Driven Approval.* Addison-Wesley, 1992.

[JSSS01] S. Jajodia, P. Samarati, M. Sapino, V. S. Subrahmanian, Flexible support for multiple access control policies, *ACM Trans. on Database Systems*, Vol. 26, No. 2, June 2001, pages 214-260.

[Jur01] J. Jurjens. Towards development of secure systems using UMLsec. In H. Hussmann, editor, *In the Proceedings of 4th International Conference of Fundamental Approaches to Software Engineering*, LNCS, pages 187-200. Springer, 2001.

[KP00] M. Koch, F. Parisi-Presicce, Access Control Policy Specification in UML, *In Proc. of Critical Systems Development with UML, satellite workshop of UML 2002*, TUM-I0208, pages 63-78, 2002.

[KP03] M.Koch, F.Parisi-Presicce, Formal Access Control Analysis in the Software Development Process, *In Proc. ACM Workshop on Formal Methods in Security Engineering (FMSE2003)*, Washington D.C., Oct 2003, pp. 67-76.

[Kra02] R. Kraft. Designing a Distributed Authorization Processor for Network Services on the Web *Proc. ACM Workshop on XML Security,* Fairfax, Virginia 2002.

[LBD02] T. Lodderstedt, D. Basin, J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. *In the proceedings of the 5th International Conference on the Unified Modeling Language,* Dresden, Germany. P ages 426-441. Springer, October 2002.

[LBN99] J. Lobo, R. Bhatia and S. Naqvi. A Policy Description Language. *In Proc. AAAI.* July, 1999.

[Mye99] A. Myers. JFlow: Practical mostly-static information flow control. *In Proc 26th ACM Symp. on Principles of Programming Languages (POPL),* pages 228--241, San Antonio, TX, January 1999.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering.* ACM Press, 2000.

[NE01] B. Nuseibeh, S. Easterbrook and A. Russo, Making Respectable in Software Development, *Journal of Systems and Software*, 56(11), November 2001, Elsevier Science Publishers.

[OCL01] Object Management Group. OMG Object Constraint Language Specification, Version 1.5, 2001. *http://www.omg.org/cgi-bin/doc?formal/03-03-01*

[OMG01] Object Management Group. OMG Unified Modeling Language Specification, Version 1.4, 2001. *http://www.omg.org/technology/documents/formal/uml.htm.*

[Pfl98] S. Pfleeger. *Software Engineering:Theory and Practice*. Prentice-Hall. 1998.

[RLK+03] I. Ray, N. Li, D. Kim and R. France, Using Parameterized the UML to Specify and Compose Access Control Models, *Proceedings of the Sixth IFIP WG 11.5 Conference on Integrity and Control in Information Systems*, Lausanne, Switzerland, November 2003.

[Ros03] Rational Rose. *http://www.rational.com.*2003.

[RUP04] Rational Software Corporation. Rational Unified Process. *http://www.rational.com/products/rup/index.jsp.* 2004

[Rus01] J. Rushby. Security Requirements Specifications: How and What? *In the proceedings of Symposium on Requirements Engineering for Information Security (SREIS)*. Indianapolis, IN. March, 2001.

[SA00] M. Shin, G.-J. Ahn. The UML-based Representation of Role-based Access Control. *In Proceedings of the 5th IEEE International Workshop on Enterprise Security (WETICE 2000)*, NIST, MD, June 14-16, 2000.

[SBC+97] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information flow control in object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):524–538, July-Aug. 1997.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):3"7, 1996.

[Sen02] S. Sendall. Specifying Reactive System Behavior, *Ph.D. thesis, Swiss Federal Institute of Technology - Lausanne (EPFL)*, May 2002

[SS00] S. Sendall, A. Strohmeier: From Use Cases to System Operation Specifications. *In the conference of the Unified Modeling Language conference, 2000.*

[SS94] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Comm.*, Vol. 32, Num. 9, September 1994.

[SZ97] R. Simon, M. Zurko. Separation of duty in role-based environments. *In the Proceedings of the 10th Computer Security Foundations Workshop*, Rockport, MA, June, 1997.

[WK99] J. Warmer, a. Kleppe. *The Object Constraint Language Precise Modeling with UML*. Addison Wesley, 1999.

[WJ02] D. Wijesekera, S. Jajodia, Policy Algebras for Access Control -The predicate Case, *Proc. 8th ACM Conference on Computer and Communications Security*, Washington, DC, November 17-22, 2002, pages 171-180.

APPENDICES

Publications

The contributions of this dissertation have been published/ in submission in several international referred conferences and journals. The following lists those publications:

1. K. Alghathbar, D. Wijesekera. Analyzing Flow Control Policies in Requirements Engineering. In the Proc. of the IEEE 5th International Workshop on Policies for Distributed Systems and Networks. Yorktown Heights, New York. June 7-9, 2004.
2. K. Alghathbar, D. Wijesekera. Incorporating Access Control Policies in Requirements Engineering. Journal of Computer and Information Science (IJCIS). vol. 5, no. 3, Mar. 2004.
3. K. Alghathbar, D. Wijesekera. *authUML: A Three-phased framework to analyze access control specifications in Use Cases*. In proc. of the Workshop on Formal Methods in Security Engineering (FMSE), Washington, DC. October 2003. ACM Press.
4. K. Alghathbar, D. Wijesekera. *Consistent and Complete Access Control Policies in Use Cases*. In proc. of the 6th International Conference on the Unified Modeling Language (UML'03), San Francisco, CA. October 2003.

5. K. Alghathbar, D. Wijesekera. *Modeling Dynamic Role-based Access Constraints using the UML.* In proc. of the 1st International Conference on Software Engineering Research & Applications (ICSERA'03), San Francisco, CA. June 2003.
6. K. Alghathbar, D. Wijesekera. *Extending the UML To Model Dynamic Authorization Policies.* In proc. of the International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03), Rio de Janeiro, Brazil. June 5-7, 2003.

CURRICULUM VITAE

Khaled S. Alghathbar was born on September 26, 1976, in the Kingdom Saudi Arabia and is a citizen of Kingdom of Saudi Arabia. He received the B.S. in Information Systems from King Saud University, Saudi Arabia, in 1998 and the M.S. in Information Systems from George Mason University, Fairfax, VA, in 2001. During 1998-1999, he was a teacher assistant in the College of Computer Science and Systems in King Saud University, Saudi Arabia. He received the following certificates from George Mason University: Information Systems Security, Electronic Commerce, Information Engineering and Software Engineering. In addition, he is a Certified Information Systems Security Professional (CISSP), Microsoft Cerified Systems Engineering in Secuirty (MCSE: Security) and a certified CompTIA Security+.

العنوان:	Incorporating Access and Flow Control Policies in Requirements Engineering
المؤلف الرئيسي:	Al Ghasbar, Khaled. S
مؤلفين آخرين:	Wijesekera, Duminda(Super.)
التاريخ الميلادي:	1998
موقع:	فيرفاكس، فرجينيا
الصفحات:	1 - 155
رقم MD:	618333
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة دكتوراه
الجامعة:	George Mason University
الكلية:	Volgenau School of Engineering
الدولة:	الولايات المتحدة الأمريكية
قواعد المعلومات:	Dissertations
مواضيع:	المتطلبات، الوصول، التحكم، هندسة الحاسبات، البرمجيات
رابط:	https://search.mandumah.com/Record/618333

Incorporating Access and Flow Control Policies in Requirements Engineering

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University.

By

Khaled S. Alghathbar

B.S. King Saud University, Saudi Arabia, June 1998
M.S. George Mason University, Fairfax, VA, August 2001

Director: Dr. Duminda Wijesekera
Information and Software Engineering

Spring Semester 2004
George Mason University
Fairfax, VA