MASTER IN COMPUTER SCIENCE

Implementation of access control using aspect-oriented programming

Montrieux, Lionel

Award date:
2009

Awarding institution:
University of Namur

Link to publication

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique

Implementation of Access Control using Aspect-Oriented Programming

Lionel Montrieux

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique
Année académique 2008-2009

Abstract

English

UMLsec is an extension of UML that allows one to define security-related properties on a set of UML diagrams, as well as to check whether or not the diagrams fulfil those properties. It is a sound and efficient way of making sure that security properties are actually enforced during the software modelling phase.

If a model does not fulfil a UMLsec property, we propose a way of (semi-)automatically modify it in order for the desired property to be correctly enforced.

But still, mistakes can easily arise when translating the UML diagrams to code. Therefore, we propose a way to automatically generate code from the model that will fulfil the same security properties as the model does. We compare the Object-Oriented and the Aspect-Oriented approaches, and select one to be implemented.

Keywords: UML, UMLsec, Object-Oriented Programming, Aspect-Oriented Programming, model transformation, security, automatic code generation, RBAC

Français

UMLsec est une extension d'UML qui permet de définir des propriétés ayant trait à la sécurité sur un ensemble de diagrammes UML, ainsi que de vérifier que le modèle respecte effectivement les propriétés énoncées. Il s'agit d'une manière efficace de s'assurer que les propriétés de sécurité sont effectivement mises en application durant la phase de modélisation.

Si un modèle ne satisfait pas une propriété UMLsec, l'on propose une méthode pour le modifier (semi-)automatiquement afin que la propriété désirée soit respectée par le modèle.

Toutefois, des erreurs peuvent aisément se glisser lors de la traduction des diagrammes UML en code. Par conséquent, l'on propose une méthode permettant de générer automatiquement le code à partir du modèle, afin qu'il satisfasse aux mêmes propriétés de sécurité que celui-ci. L'on compare les approches Orienté-Objet et Orienté-Aspect, et l'on en sélectionne une, qui sera implémentée.

Mots clefs :UML, UMLsec, Programmation Orienté-Objet, Programmation Orienté-Aspect, transformation de modèles, sécurité, génération automatique de code, RBAC

Acknowledgements

This thesis could never have been written without the help of a lot of people. I would like to take the opportunity to acknowledge them.

Dr. Jan Jürjens and Dr. Yijun Yu, from the Open University, Milton Keynes, UK, for their hospitality during my stay at the Open University and for their advices and support, as well as Prof. Bashar Nuseibeh, and everyone in the Computing department.

Prof. Pierre-Yves Schobbens and Hubert Toussaint, my supervisors at the University of Namur,for their support and many valuable remarks during all the stages of this thesis and the related internship at the Open University.

All the attendees of the Trento Security Workshop (January 19 - 20, 2009) that allowed we to talk about my work at The Open University, and for their valuable comments and advices.

Finally, this thesis couldn't have been written without the help, support and countless hours of proofreading of my family and friends.

Contents

List of Figures

Glossary

Advice (AOP) In Aspect-Oriented Programming, an advice inserts code before, after or around a specified pointcut. Advices are discussed in section 2.4.

AOP *Aspect-Oriented Programming* Aspect Oriented Programming is a programming paradigm that allows one to separated crosscutting concerns from the business logic code. Chapter 2 discusses the Aspect-Oriented paradigm in more details.

AppArmor AppArmor is an open source implementation of MAC for the Linux kernel, originally written by Novell. It claims to be easier to configure and cause less overhead than SELinux.

ArgoUML ArgoUML is an Open-Source UML modelling software. It is used to create the UML models that will be loaded inside the UMLsec tool, described in chapter 6.

AspectJ AspectJ is an Aspect-Oriented programming language that extends the Java programming language capabilities. It is used in the UMLsec tool, both in the tool implementation and in the code the tool automatically generates from UMLsec models.

DAC *Discretionary Access Control* Discretionary Access Control is "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong" [oD85]

JAAS *Java Authentication and Authorization Service* JAAS is an authentication and authorization framework that is part of the Java SDK since JDK 1.4. It allows one to use any desired authentication or authorization mechanism by replacing the default ones with custom developments. A more complete description of JAAS can be found on section 5.3.1.

Joinpoint (AOP) In Aspect-Oriented Programming, a joinpoint is any kind of place in the code that can be recognised by the weaver. Joinpoints are discussed on section 2.4.

LSM *Linux Security Modules* The Linux Security Modules is a framework of the Linux kernel that allows one to define access control mechanisms. It allows users to load their favourite access control system as a kernel module, which makes changing between different implementations of an access control mechanism, or even between different access control mechanisms, easier.

MAC *Mandatory Access Control* Mandatory Access Control is "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorisation (i.e., clearance) of subjects to access information of such sensitivity" [oD85].

MLS *Multi-Level Security* Multi-Level Security allows a system administrator to define clearance levels, and to label resources with a minimum level that a user will need in order to access the protected resource.

NIST *National Institute for Standards and Technologies* The NIST is an American federal agency within the US Department of Commerce, whose goals include the promotion of standards.

OOP *Object-Oriented Programming* Object-Oriented Programming is a well-known programming paradigm that has been developed since the 1960s. It allows one to decompose code into modules and classes, which helps to achieve better modularisation of the code.

Permission A permission is an access control term that allows its holder to perform an action on a resource (a file, a process, ...)

Pointcut (AOP) In Aspect-Oriented Programming, a pointcut is a pattern that matches joinpoints. Pointcuts are discussed on section 2.4.

Poseidon Poseidon is a commercial fork of ArgoUML. It was historically used instead of ArgoUML to create models to be checked by the UMLsec tool.

RBAC *Role-Based Access Control* A widely-used access control model where users are assigned roles, and roles are granted permissions. RBAC allows the definition of permissions in a way that matches an organization's structure, making the access control policy easier to create and maintain.

SELinux *Security-Enhanced Linux* SELinux is an open source implementation of MAC, originally developed by the NSA, and later included in the Linux kernel tree.

TrustedBSD TrustedBSD is an open source framework that provides MAC capabilities to the BSD kernels.

UMLsec UMLsec is an extension of UML that uses the standard UML extension mechanisms, such as stereotypes and tagged values, to model security properties on a UML model.

Chapter 1

Introduction

Access control is a major concern in modern computer systems and software. Access to objects, data and programs needs to be restricted, either for legal reasons, like privacy protection or military secrets, or for business-related reasons, like trade secrets or other critical information, or even simply to protect a system from unauthorised users. Lots of access control models have been proposed, but one of the most famous is probably Role-Based Access Control (RBAC) [SFK00], that allows to set permissions in a way that matches the reality of an organization (a company, a government agency, an education institution, . . .).

Security in general, and access control in particular, being more and more crucial for the safety of systems and data, IT professionals started to realise that security is a problem that should be handled as soon as possible in the software development process, instead of being implemented in the end of the project, if there is some time left.

Consequently, several approaches have been developed to help taking security into account as early as possible in the development cycle. One of the is UMLsec [Jür05], which proposes to define security properties on UML models, using the UML extension mechanism. Moreover, UMLsec allows one to check his model against the desired security properties (including, but not limited to, access control properties), in order to make sure that the model does not have any unexpected flaw.

Although making sure that a UML model is secure regarding a set of security policies is a great way of gaining confidence in the security of the resulting software, problems can also, and most probably will, arise when the developers will translate the model into code. As perfect as the model can be, there will without any possible doubt be bugs and errors, and those might affect the security of the software. Therefore, automatically generating the code would allow one to make sure that it meets the same security standards as the model it comes from.

This document focuses on how to generate code from a UML model with UMLsec properties included in a way that enforces those properties exactly as in the original model. And since models can also **not** enforce the required UMLsec properties, this document also gives hints about how to automatically modify the model when the desired properties are not met, in order to correct it from a security point of view, without changing the business logic.

For the code generation, we try to use Aspect-Oriented Programming techniques and see how they can help achieving a better modularisation and more maintainable security policies.

Chapter 2 will be an introduction to Aspect-Oriented Programming, while chapter 3 will review several access control techniques, including of course RBAC. Then chapter 4 will introduce and extend UMLsec, as well and give thoughts about how to automatically modify a model to enforce UMLsec properties. Chapter 5 will then focus on producing code that enforces the same UMLsec properties as the model it is derived from, and chapter 6 will present the tool support for UMLsec.

Given the unusual presentation of this document, where my personal contributions are sometimes mixed with description of pre-existing works, let us quickly summarise which parts are descriptions of preexisting work, and which parts are original contributions.

Chapters 2 and 3 are purely preexisting work descriptions. They introduce the Aspect-Oriented Programming concepts and access control concepts that will be used later.

Chapter 4 is a mix of prexisting and original work. Section 4.1 describes a subset of UMLsec that will be used later, and is purely a summary of preexisting work. The next sections, however, are original work: discussion about extending some UMLsec properties to other UML diagrams on section 4.2, extension of the « rbac » property on section 4.3, conflicts between UMLsec properties in section 4.4, and finally, the automated hardening of unsecure UMLsec models on section 4.5, that has been done with the help of Dr. Jan Jürjens from The Open University.

Chapter 5 is also original work, at least for the code generation from UMLsec properties. Of course, code generation from plain UML diagrams was preexisting, but it is barely discussed in that chapter.

Finally, chapter 6 describes the current state of the UMLsec tool development, and therefore, preexisting code is mixed with original contributions. Those original contributions include the automatic correction of unsecure models (section 6.3), the code generation (section 6.4), part of the UML modelling tool migration (section 6.5, essentially bug tracking and fixing), and the development process improvements (section 6.6).

Chapter 2

Aspect-Oriented programming

2.1 Overview

Aspect-Oriented programming is a programming technique that has been developed since 1996 in Xerox Research Center, Palo Alto, CA. The idea behind Aspect-Oriented programming was to address problems that could not be solved conveniently using procedural or Object-Oriented programming paradigms. It appeared that a new way of designing code was needed to avoid those problems, that will be discussed in section 2.3. Aspect-Oriented programming will then be discussed in section 2.4. We will then talk about Aspect-Oriented languages in section 2.5.

2.2 The Object-Oriented approach

The Object-Oriented paradigm was first introduced in the 1960, in order to help the design process of increasingly complex software. The first language to introduce Object-Oriented concepts (like objects, classes, superclasses, methods, ...) was Simula [DMN], but the first programming language to be called an "Object-Oriented Programming language" was Smalltalk [Kay], developed in the 1970s at Xerox Labs.

The Object-Oriented concepts are now widely-used and well known, which is why we will not describe objects, classes, polymorphism, inheritance or encapsulation here.

The most interesting thing about Object-Oriented programming is that it allowed one to decompose a program into independent modules, each module implementing a functionality.

2.3 The limits of the Object-Oriented approach

Although the Object-Oriented paradigm allows one to better designed software, mainly because the object model is closer to real-life problems than the plain old procedural paradigm, some concerns are still really hard to model. Those concerns are called *cross-cutting concerns* [IKL+97], because they can't be addressed in just one or even a few classes, but instead, have an impact on the whole code, and therefore require one to write small pieces of code everywhere in the code base. Typical examples on cross-cutting concerns are logging, security, persistence, ...

Those modularisation problems can be classified into two categories: code tangling, and code scattering [Lad03]. They are described in the following sections.

2.3.1 Code tangling

When a module implements several concerns at the same time, we can talk about code tangling: a piece of code is actually doing several different things, which make it hard to write, develop, understand and maintain. Figure 2.1 (inspired by a figure from [Lad03]) illustrates code tangling: the business logic is mixed with calls to the logging API and the persistence API.

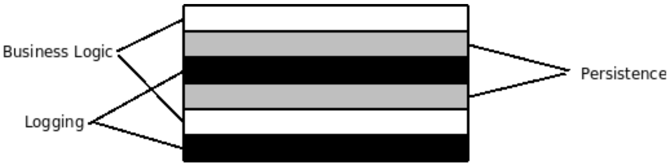


Figure 2.1: Code tangling

Figure 2.2 is a snippet of dummy Java-like code that illustrates the same tangling phenomenon in a class, where business logic code is mixed with logging API and persistence API calls.

2.3.2 Code scattering

Code Scattering, on the other hand, appears when a concern is implemented in several places or modules in the code. This is typically the case with cross-cutting concerns, whose implementation is spread all over the code base [Lad03]. Figure 2.3 (also inspired by a figure from [Lad03]) illustrates the code scattering phenomenon for the security concern.

2.4 Aspect-Oriented Programming concepts

2.4.1 Aspects

Aspects are an answer to the code scattering and code tangling problems. Object-Oriented Programming is a good way to model functional requirements, but it completely fails when non-functional requirements need to be modelled, leading to scattering and tangling.

Aspects are an extension to the traditional Object-Oriented paradigm that allow one to model cross-cutting concerns efficiently. Instead of spreading those concerns' implementation through the whole code, an aspect is created that will handle the cross-cutting concern.

It is important to notice that Aspect-Oriented programming does not *replace* Object-Oriented programming, but instead, extends it to address problems that Object-Oriented programming wasn't capable of dealing with in an efficient way.

```
public Client() {
    private String name = null;
    private int number = null;
    private int balance = null;
    private Logger logger = Logger.getLogger("Client");
    private Persistance persistance = Persistance.getDatabase();

    public Client(String name, int nbr) {
        this.name = name;
        this.number = number;
        this.balance = 0;
        logger.log("New␣client␣created:␣" + name + \
"\t␣with␣id:␣" + nbr);
        persistance.saveState();
    }
    public bool credit(int amount) {
        balance = balance + amount;
        logger.log("Credit␣added␣to␣account␣" + nbr + \
":␣" + amount);
        logger.log("New␣credit:␣" + balance);
        peristance.saveState();
        return true;
    }
    public bool debit(int amount) {
        if (amount <= balance) {
            balance = balance - amount;
            logger.log("Credit␣withdrawn␣from␣account␣" \
+ nbr + ":␣" + amount);
            logger.log("New␣credit:␣" + balance);
            persistance.saveState();
            return true;
        }
        else {
            logger.log("Can't␣debit␣" + amount + \
":␣insufficient␣balance");
            return false;
        }
    }
    public String getName() {
        return name;
    }
    public int getBalance() {
        return balance;
    }
}
```

Figure 2.2: A Java-like class illustrating the tangling problem (inspired from [Lad03])

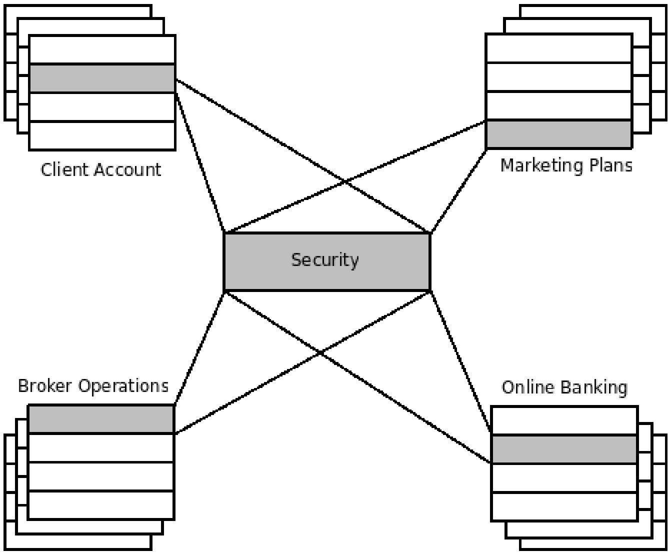


Figure 2.3: Code scattering

Aspect-Oriented Programming defines and uses four new concepts: joinpoints, pointcuts, advices and weaving, that we will introduce shortly.

Joinpoints are points in a program execution. It can be anything: an assignation, a method call, a constructor call, an exception, ...

Pointcuts are used to select joinpoints in the program execution. Pointcut definitions are designed to match specific joinpoints that will later be used to insert or replace code, using the aspect.

Advices are pieces of code that will be executed when matching a joinpoint in the program execution with a pointcut.

Joinpoints, pointcuts and advices are used together in Aspect-Oriented Programming: the pointcut captures joinpoints where code defined in an aspect has to be executed, and the advice is the place where the code to be executed is defined.

Weaving is the action of "integrating" aspects into the functional code. It is a composition mechanism that will produce executable code from the functional code and the aspects. Weaving rules can be defined that will determine the final result, like for example, the order in which aspects should be woven into the code.

The weaving program is called the *weaver*. It can be implemented in several ways. It can perform source-to-source weaving [Lad03], which means that each aspect will be weaved into the source code before the compilation. But it can also weave aspects into code that has already been compiled. The second approach is more difficult to implement, but allows one to add and remove aspects during the execution of the program.

2.5 Aspect-Oriented languages

There exist several Aspect-Oriented languages, extending several Object-Oriented or non-Object-Oriented languages. Here, we focus on one language, AspectJ, but many other languages exist.

2.5.1 AspectJ

AspectJ is an Aspect-Oriented extension to the Java programming language. The AspectJ project started several years ago as a Xerox [xer] research project, and is now released as Open Source software under the Eclipse Public Licence [Fouc], and hosted by the Eclipse Foundation [Foud]. It runs on any Java2 compatible platform.

The AspectJ project includes a compiler, a debugger, a program structure browser, a documentation generator, as well as integration with Eclipse, Netbeans, GNU Emacs, JBuilder and Ant [Cen02].

AspectJ is a widely-used Aspect-Oriented Programming language, for example in projects like Spring [Sou] or Tomcat [Foub], as well as in the UMLsec tool [Jü04].

In the following sections, we briefly describe how AspectJ implements the main Aspect-Oriented Programming concepts.

Aspects

Aspects in AspectJ are a special type of classes. The syntax is described in figure 2.4.

```
public aspect MyAspect {
    // variables

    // pointcuts

    // advices
}
```

Figure 2.4: Aspect syntax in AspectJ

By default, an aspect is a singleton class, but it is possible to change this and create several instances of an aspect. Aspect inheritance follows the Java rules for classes, which means that an aspect can only inherit from one (and no more than one) other aspect. Abstract aspects can also be defined if necessary.

Pointcuts

Pointcuts in AspectJ are defined inside an aspect. They can match any kind of joinpoint: method calls, method executions, objects initialisation, constructor calls, exceptions being thrown, access to attributes, . . .

But it is also possible to restrict a pointcut pattern to match only joinpoints inside a particular execution context (for example: only inside the execution flow of a method) or to avoid capturing some joinpoints.

Here is the syntax of an AspectJ pointcut [Lad03]:

```
[access specifier] pointcut pointcut-name([args]) : pointcut-definition
```

The *pointcut-definition* is a pattern that will select the joinpoints during the program execution.

Advices

There are three kinds of advices: `before()`, `after()` and `around()` advices. The `before()` advice will execute its code before the joinpoint, the `after()` advice will execute its code after the joinpoint, and the `around()` advice will execute its code instead of the joinpoint.

The advice syntax is as follows:

```
advice declaration : pointcut specification  advice code
```

Inside an advice, it is possible to access information about the captured joinpoint. It can be useful, for example, in order to get the name of the captured method.

Weaving

The AspectJ weaver is not a source-to-source weaver. Instead, it weaves aspects inside class files, which allows to add or remove aspects during the execution of the program.

When several aspects have been defined, the order in which they will be woven can be important, since it can change the behaviour of the resulting program. It is therefore possible to define the weaving order inside an aspect.

Example: a simple aspect

The following example will illustrate how a simple aspect written in AspectJ works. We will use an aspect to add logging features to a simple program. First, let's write two classes in Java, as we can see on figures 2.5 and 2.6. We have a *Client* class, and an *Account* class.

Now, suppose that we want to add logging to our simple program, using log4j. Specifically, we want every *Client* or *Account* creation to be logged on the `INFO` level. For debugging purposes, we also want every method call to be logged on the `TRACE` level.

```java
import java.util.ArrayList;
import java.util.Iterator;

public class Client {
    private ArrayList<Account> accounts = new ArrayList<Account>();
    private String name;
    private int number;

    public Client(String name, int number) {
        this.name = name;
        this.number = number;
    }

    public addAccount(Account account) {
        accounts.add(account);
    }

    public Account getAccount(int number) {
        Iterator iter = accounts.iterator();
        while (iter.hasNext()) {
            Account account = iter.next();
            if (account.getNumber == number)
                return account;
        }
    }

    public String getName() {
        return name;
    }

    public int getNumber() {
        return number;
    }

}
```

Figure 2.5: client.java

```java
public class Account {
    private int number;
    private int balance;

    public Account(int number) {
        this.number = number;
    }

    public credit(int amount) {
        balance = balance + amount;
    }

    public boolean debit(int amount) {
        if (amount > balance)
            return false;
        balance = balance − amount;
        return true;
    }

    public int getBalance() {
        return balance;
    }
}
```

Figure 2.6: account.java

Figure 2.7 describes the logging aspect. The `clientCreation()` (resp. `accountCreation()`) pointcut captures the call to any constructor of the class *Client* (resp. *Account*), and the `after` advice on line 15 (resp. 19) adds the call to the logging module, right after the constructor has been executed. The execution of methods is captured by the last pointcut, `methodCall()`, and the `before` advice and the `after` advice on lines 23 and 28 add calls to the logging API for every method call performed.

```
import org.apache.log4j.*;

public aspect Logging {
    Logger logger = Logger.getLogger("simple␣logger");

    pointcut clientCreation()
        : execution (Client.new(..));

    pointcut accountCreation()
        : execution (Account.new(..));

    pointcut methodCall()
        : execution(* *.*(..)) && !within(Logging);

    after() : clientCreation() {
        logger.info("New␣client");
    }

    after() : accountCreation() {
        logger.info("New␣account");
    }

    before() : methodCall() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        logger.trace("ENTERING␣" + sig.getName());
    }

    after() : methodCall() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        logger.trace("LEAVING␣" + sig.getName());
    }
}
```

Figure 2.7: logging.aj - a simple logging aspect

Chapter 3

Access control

3.1 Introduction

The development of computers, and specifically multi-users operating systems has quickly led to concerns about who should be able to access, create, modify or delete data and execute programs. There was a growing need to restrict access to some data, and that's why access control was created.

In 1985, the American Department of Defense (DoD) released the Trusted Computer System Evaluation Criteria (TCSEC) [oD85]. The TCSEC defines two types of access control policies: Discretionary Access Control and Mandatory Access Control.

Those are still widely used today, but they have also shown their limits in several scenarios. Since then, other models have been proposed, including Role-Based Access Control[SFK00], Organization-Based Access Control[ABB$^+$03], ...

This chapter describes some of those access control models, but focuses on Role-Based Access Control that will be used in the next chapters as well.

3.2 DAC: Discretionary Access Control

3.2.1 Overview

Discretionary Access Control has been defined in 1985 by the Trusted Computer System Evaluation Criteria as "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)." [oD85]

3.2.2 Implementations

The Unix system of users, groups and read/write/execute permissions is a typical example of DAC: each user can grant permissions to other users or groups of users on the files he owns, without restrictions.

On a Unix system, everything is a file. Files are files, but so are directories, pipes, devices,... A file's permissions are described like this :
`-rwxrw-r-- john staff 2048 May 18 2009 some_file`
where the first group of letters (`-rwxr--r--`) describes the permissions associated to the file, then the owner (*john*) is specified, then the group (*staff*), followed by the file size (*2048 bytes*), then the last modification date (*May 18 2009*), and finally the file name (*some_file*). In the permissions description, the first three letters indicate which permissions the owner, `john`, has on the file. The next three describe the permissions for the group, `staff`, and the last three ones describe the permissions for the other users. `r` is for the permission to *read* the file, `w` is for *writing* into it, and `x` is for executing it. Any `-` means that the permission is *not* granted. In this example, the user, *john*, can *read*, *write* and *execute* `some_file`, while the members of the *staff* group can only *read* and *execute* it. The other users can only *read* the file.

3.3 MAC: Mandatory Access Control

3.3.1 Overview

Mandatory Access Control has been defined by the Trusted Computer System Evaluation Criteria as "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorisation (i.e., clearance) of subjects to access information of such sensitivity" [oD85].

The difference between DAC and MAC is that MAC allows the system administrator to define restrictions that the users will not be able to override when setting permissions on the objects they own, while DAC allows them to do whatever they want with those objects.

For example, using MAC an administrator could forbid a user to give `write` permissions to the documents he owns, for some reason. That would be impossible to enforce with DAC.

3.3.2 MLS: Multi-Level Security

Multi-Level Security (MLS) defines the ability for the administrator to define different levels of authorisation. Each resource gets a security level, and every user gets a security level clearance allowing him to access resource of his clearance level, or any lower level.

It is a convenient way to hide sensitive data to users that do not have the right credentials to access it, as well as for higher level users to release sanitized versions of documents to lower-level users.

3.3.3 Implementations

SELinux has initially been developed by the NSA, and merged into the Linux kernel in 2003. It uses a feature of the Linux 2.6 kernel, LSM (Linux Security Modules) [WCS$^+$02]. LSM is an access control framework whose goal is to allow several access control policies to be developed as modules. Linux is used in a variety of different contexts, that have different security needs. Moreover, there is no agreement on one general access control solution that would fit everyone's needs. Therefore, a mechanism allowing people to use whatever access control mechanism they want to use without requiring too much changes in the kernel was needed. SELinux is used by several major Linux distributions, like Red Hat Enterprise Linux and Fedora.

AppArmor is another MAC implementation for the Linux kernel, and also uses LSM. It is included in distributions like Ubuntu and SUSE Linux. It is now supported by Novell. However, it is not included in the vanilla Linux kernel tree. This is an alternative to SELinux that has been developed with the idea of being easier to administrate and having a smaller impact on performances [Nov].

Windows Vista and Windows Server 2008 also implement Mandatory Access Control, using Mandatory Integrity Control.

Mac OS X also has its MAC mechanism, which is an implementation of the TrustedBSD [Wat] framework. TrustedBSD was originally developed for the FreeBSD operating system.

3.3.4 Example: SELinux

SELinux, which stands for Security-Enhanced Linux, has been developed by the NSA, and released as Open Source software in 2000. It has then been rewritten to use the LSM framework.

In order to enforce access control, SELinux defines three types of *contexts*, that contain access control information, like owner, role, type, or security level:

- contexts for *files*
- contexts for *processes*
- contexts for *users*

Let's take a closer look at these contexts, and how they are defined.

Contexts for files

Let's take a look at the permissions of a file on a GNU/Linux distribution with SELinux enabled:

```
ls -Z umlsec.tex
-rw-r--r--. john john unconfined_u:object_r:user_home_t:s0 umlsec.tex
```

The usual DAC permissions are still there, but there is also a SELinux *context*: `unconfined_u:object_r:user_home_t:s0`. The context contains the SELinux access control information for this file. However, the DAC permissions still have precedence, which means that if the DAC permissions don't allow an operation on the file, then the SELinux context won't be taken into account. In this example, we have a user (`unconfined_u`), a role (`object_u`), a type (`user_home_t`), and a level (`s0`).

The user is mapped to a system user. The SELinux user contains information about its set of roles, and its MLS (Multi-Level Security) range.

The role is an attribute of an RBAC model. Although SELinux is a MAC implementation, it also supports RBAC, that can be described using MAC principles [Kuh98].

The type of the file is used for type enforcement. When the type attribute is used to describe a process instead of a file, then it defines a domain instead of a type.

Finally, the level is the only optional attribute. It is an attribute of MLS and defines a range of levels using the following syntax: `lowestLevel-highestLevel` or `level` if the range only contains one level.

Contexts for processes

The process context is similar to the file context:

```
ps -eZ | grep passwd
unconfined_u:unconfined_r:passwd_t:s0-s0:c0.c1023 13212 pts/1 00:00:00 passwd
```

We still have a user (`unconfined_u`), a role (`unconfined_r`), a type (`passwd_t`) and a level range (`s0-s0:c0.c1023`). Remember that for a process, the type defines a domain.

Contexts for users

Finally, the users context appears as follows:

```
id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Every user has a mapping to a SELinux user (`unconfined_u`), who has a role (`unconfined_r`), a domain (`unconfined_t`) and a level range (`s0-s0:c0.c1023`).

3.4 DAC and MAC limitations

DAC is a widely-used access control system, probably because it is really easy to use: each file owner decides how he wants to protect the file from others. The major problem is that the administrator doesn't have any possibility to enforce access control restrictions on files created by users. If this is an acceptable solution for individual users or systems that do not contain any critical information, it is however unsuitable for most organisations, particularly when there is critical data involved.

On the other hand, MAC is a more powerful mechanism, that includes the ability for the administrator to define complex policies and to enforce restrictions on which permissions a user can give to its files. MAC is also strongly connected to MLS, allowing the administrator to define and use security levels to avoid leakage of critical information. However, a MAC model can quickly become complex and almost impossible to manage. Also, MAC models, at least in their first form as defined in [**?**], cannot handle some complex properties, like dynamic Separation of Duty (that will be discussed in section 3.5.3).

3.5 RBAC: Role-Based Access Control

Historically, RBAC (Role-Based Access Control) came after MAC and DAC. Before that, MAC and DAC were the only known access control models. RBAC can be used to simulate DAC and MAC, and MAC can sometimes be used to implement RBAC [NIS].

Here, we will concentrate on three major step towards the definition of an RBAC standard: the initial RBAC proposal, followed by an RBAC framework proposal, and finally, the RBAC standard proposal.

The initial RBAC model has been defined in 1992 by Ferraiolo and Kuhn [FK92], who gave a first formalisation of the model. However, their proposal has been discussed and improved a lot. In 1996, Sandhu, Coyne, Feinstein and Youman introduced a framework for RBAC model [SCFY96]. In 2000, Sandhu, Ferraiolo and Kuhn proposed a unified RBAC model as an RBAC standard [SFK00], that was adopted in 2004 by the American National Standards Institute, International Committee for Information Technology Standards (ANSI/INCITS) as an industry standard (*American National Standard 359-2004*).

The concept of RBAC models uses three basic components: users, roles and permissions. A user can have multiple roles, and each role is associated with a set of permissions. Roles are supposed to match the actual organizational roles a user has. For example, in a banking system, roles could be ***clerk***, ***supervisor***, ***director***, ***auditor***, ... ***supervisors*** could have the permission to accept any kind of credit to the customers, while ***clerks*** could only approve those under £1000. Figure 3.1 describes the relation between users, roles and permissions.

Using roles provides many advantages over the usual, DAC-like owner/group mechanism: since it matches "real life" organizational groups, it makes adding or removing permissions to a particular type of users really easy. Also, the permissions of a user whose status in the organization changes (like, for example, an employee being promoted to the supervisor status) can be updated simply by adapting the roles he is assigned to. Revoking a user's credentials is also as easy as removing all the roles he is assigned to. So is adding a new user.

Another key concept of RBAC is sessions. A session corresponds to the use of the system by a user at a particular time. Depending on the implementation used, users can chose which roles they want to activate in the set of roles they are assigned to, or all their assigned roles are activated when the session starts. When a role is activated, the user can use all the permissions associated with the role. Some implementations only allow one role to be activated at a time, but this restriction is now explicitly forbidden by the RBAC standard, as described in section 3.5.3.

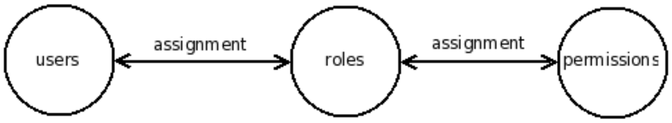


Figure 3.1: Relationships between roles, users and permissions

3.5.1 Ferraiolo and Kuhn's RBAC model

The first Role-Based Access Control proposal was Ferraiolo and Kuhn's RBAC model, proposed in 1992 [FK92]. This first proposal introduces the key RBAC concepts: users (called subjects), roles, permissions (called transactions, but it's essentially the same idea) and sessions.

Additionally, this proposal also describes properties such as Least Privilege, Separation of Duty (both static and dynamic) and role hierarchies, but those are not formally defined, leading to various possible interpretations.

3.5.2 Sandhu, Coyne, Feinstein and Youman's RBAC framework

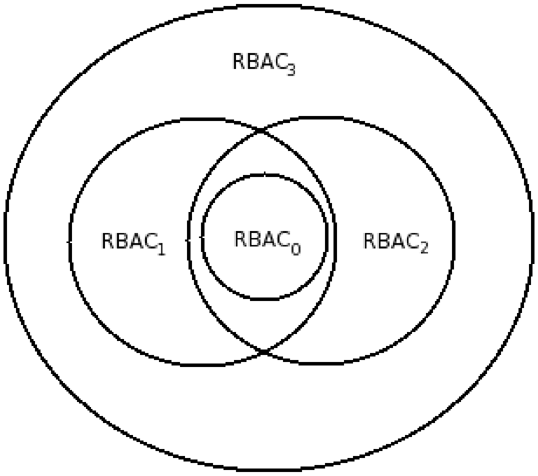


Figure 3.2: Level hierarchy for the 1996 RBAC framework proposal

The RBAC framework proposed in 1996 by Sandhu, Coyne, Feinstein and Youman in [SCFY96] introduces four levels in the RBAC model: $RBAC_0$, $RBAC_1$, $RBAC_2$ and $RBAC_3$. $RBAC_0$ describes the minimum requirements to support RBAC, while $RBAC_1$ adds support for role hierarchies on top of $RBAC_0$. $RBAC_2$ adds support for constraints on top of $RBAC_0$

too. Finally, $RBAC_3$ is the consolidated model, that include both $RBAC_1$ and $RBAC_2$. Figure 3.2 describes the relations between the different levels.

The four levels proposed in this paper will not be discussed in detail here, since the concept of levels has been adapted in the NIST (National Institute of Standards and Technology) [oC] RBAC standard that is described later, and where the different levels are explained in detail.

3.5.3 NIST RBAC model

Over the years, several different RBAC models have been proposed, including, but not limited to, those described in sections 3.5.1 and 3.5.2. Several implementations have also been developed.

Each of those models and implementations had their own interpretation of RBAC concepts, resulting in a confusing set of slightly different RBAC definitions. In order for the RBAC technology to develop and gain broader audience, a standard definition was needed.

In 2000, Sandhu, Ferraiolo and Kuhn proposed the definition of a unified RBAC model as such a standard [SFK00]. Their proposal was accepted in 2004 by the ANSI/INCITS (*American National Standard 359-2004*).

Overview

The NIST RBAC model is divided in 4 levels, similar to the ones described in section 3.5.2. Figure 3.3 summarises those levels and their functional capabilities. Those levels are cumulative, therefore each level includes the requirements from the previous ones.

Level 1: Flat RBAC

Flat RBAC describes the basic aspects of Role-Based Access Control. Users have roles, and those roles have permissions. Users acquire permissions by being members of roles.

The NIST RBAC model requires a many-to-many relation between users and roles, as well as between roles and permissions. That means that a user can have many roles, and that a role can have many users. Similarly, a role can hold many permissions, and a single permission can be held by many roles.

Another important requirement is that users should be able to use permissions from different roles they are assigned to simultaneously.

Figure 3.4 describes the relation between users, roles and permissions.

Level 2: Hierarchical RBAC

The second level, Hierarchical RBAC, simply adds support for role hierarchies. Those are formalised here as a partial order between parent roles and child roles, where child roles acquire their parent's permissions. Figure 3.5 describes the role hierarchy capability added to the flat RBAC model.

Level	**Name**	**Capabilities**
1	Flat RBAC	users acquire permissions through roles must support many-to-many user-role assignment must support many-to-many permission-role assignment must support user-role assignment review users can use permissions of multiple roles simultaneously
2	Hierarchical RBAC	Flat RBAC + must support role hierarchy (partial order) **level 2a** requires support for arbitrary hierarchies **level 2b** denotes support for limited hierarchies
3	Constrained RBAC	Hierarchical RBAC + must enforce Separation of Duty **level 3a** requires support for arbitrary hierarchies **level 3b** denotes support for limited hierarchies
4	Symmetric RBAC	Constrained RBAC + must support permission-role review with performance effectively comparable to user-role review **level 4a** requires support for arbitrary hierarchies **level 4b** denotes support for limited hierarchies

Figure 3.3: Summary of RBAC levels [SFK00]

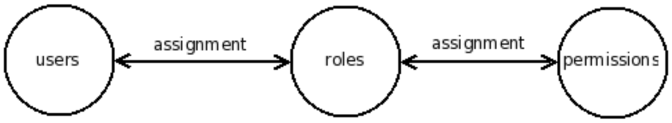


Figure 3.4: Flat RBAC

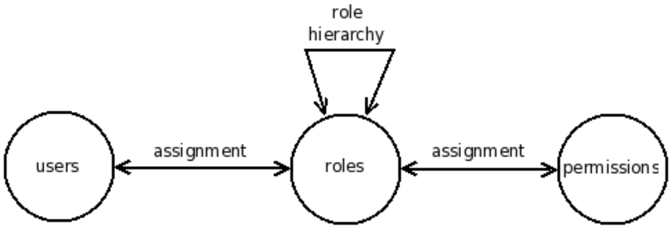


Figure 3.5: Hierarchical RBAC

The NIST standard defines two sub-levels: General Hierarchical RBAC and Restricted Hierarchical RBAC.

General Hierarchical RBAC where the role hierarchy is described by an arbitrary partial order.

Restricted Hierarchical RBAC where the role hierarchy can be restricted to simpler structures, like, for example, trees.

Level 3: Constrained RBAC

Constrained RBAC is an hierarchical RBAC model where the designer has the ability to add constraints, which can be either static or dynamic. Static constraints are associated with the user-role assignment, where dynamic constraints are associated with the activation of roles within new user sessions. A widely-used example of such a constraint is Separation of Duty.

Separation of Dutyis about making sure that a user cannot gain too much power, and that some specific operations can only be done by multiple users, each of them having permission to perform part, but not all, of the operation. In other words, Separation of Duty allows the administrator to specify that if a user is a member of a role ***roleA***, then he can not be a member of another role ***roleB***.

Separation of Duty constraints can be defined statically or dynamically. While a static Separation of Duty (SSD) constraint defines constraints on the assignment of roles, a dynamic Separation of Duty (DSD) constraint defines constraints on activated roles.

Static Separation of Duty (SSD) constraints stipulate that a user cannot be assigned two (or more) conflicting roles. Moreover, if there's an SSD constraint between two roles, then it also stands for roles that inherit from those roles. Imagine a SSD constraint between the role ***employee*** and the role ***supervisor***. If the role ***insurance employee*** inherits from ***employee***, then the constraint also holds between ***insurance employee*** and ***supervisor***.

Dynamic Separation of Duty constraints, on the other hand, are not defined regarding the *assignment* of roles, but instead, regarding the *activation* of roles. That means that a user could be assigned two potentially conflicting roles, as long as he doesn't activate both of them at the same time.

Level 4: Symmetric RBAC

The last NIST RBAC level strengthens the requirements for reviewing permission-roles assignments. It requires a review interface that can return two types of results.

The first type is "the complete set of objects that are associated with the permissions assigned to a particular user or role" [SFK00].

The second one is "the complete set of operation and object pairs that are associated with the permissions that are assigned to a particular user or role" [SFK00].

There are also a few optional requirements. A first option is “the ability to selectively define direct and indirect permission assignment” [SFK00], where direct assignments describe “the set of permissions that are assigned to the user and/or to the roles for which the user is assigned” [SFK00], and indirect assignments describe “the set of permissions that are included in the direct permission assignment in addition to the permissions that are assigned to the roles that are inherited by the roles assigned to the user” [SFK00].

A second optional requirement is “the ability to select the target systems for which the review will be conducted” [SFK00].

Uncovered attributes

There are a few RBAC attributes that are *not* discussed in the NIST RBAC proposal, some because their nature make them unsuitable for standardization, others because a consensus had not been reached when the proposal was written. Some of them are shortly explained here. This section is largely inspired from [SFK00].

Authentication While it is outside the scope of the RBAC model, how the users are authenticated in the system is also a crucial aspect of the system architecture

Constraints The NIST RBAC model recognises two types of Separation of Duty constraints (static and dynamic), but they are other ways of partitioning Separation of Duty. Also, obligation constraints aren’t addressed at all in the model.

Discretionary role activation It is the ability for a user to choose which roles should be activated during a particular session. This isn’t required by the proposal, but it is not forbidden either.

Nature of permissions The nature of permissions is not defined in the standard. They can be anything: primitives, abstract operations, customised, ...

Negative permissions The NIST RBAC model does not define negative permissions (*users assigned to role **A** can **not** perform operation **X***), but it does not forbid it either. Thus, vendors are welcome to implement it if they want.

RBAC administration There is no administration component in the NIST RBAC standard because of a lack of consensus on the subject. RBAC administration is about who can assign users or permissions to roles, and define role hierarchies.

Role engineering Role engineering consist of guidelines for designing roles and assigning users and permissions to roles.

Role revocation How (and when, especially in distributed environments) users or permissions should be revoked from a particular role is a complex issue, that is not addressed by the standard.

Scalability This is an important criteria when selecting an RBAC implementation. Scalability can be understood in terms of number of roles, number of permissions, size of role hierarchies, ...

3.5.4 Implementations

There are several implementations of the RBAC model, some of them are Active Directory and SELinux, that has already been described in section 3.3.4.

3.5.5 Limitations

While RBAC is a well-known and widely used access control model that allows one to easily solve problems that were difficult to address using MAC or DAC, it also has its weaknesses. Some limitations of the RBAC model are discussed here.

RBAC shows its limits when access control policies are not just static properties: context (like time, location, and a lot of other factors) can potentially affect a user's permissions, but it is really hard, or even impossible to model it within an RBAC model. For example, one might want to restrict the permissions of a user connected to a system using a wireless link, or from his home instead of within the company walls.

Although RBAC knows about users roles, there is no way one can be more specific, like differentiating ***senior developers*** and ***junior developers***. Of course, one could split the ***developers*** role in two different ones, or better, specialize it using role hierarchies, but this can lead to a fast-growing number of roles, making the access control policy harder and harder to manage. Sometimes, one might want to "tweak" permissions for a particular user instead of creating a new role. That's not possible with an RBAC model.

Finally, RBAC has been designed to define security policies for systems that are used by only **one** organisation. When a system has to be used by multiple organisations, RBAC does not provide any way to separate their respective roles, that would otherwise probably cause conflicts.

3.6 Other access control mechanisms

RBAC is not the ultimate access control model that solves every single problem in every possible situation, and therefore, other access control models have been developed, to address other needs.

There are some issues that are **not** addressed by RBAC, which doesn't make it the silver bullet solution for everyone's needs. First, RBAC doesn't support multiple organizations, as we discussed on section 3.5.5.

Another strong limitation of RBAC is its lack of support for contextual permissions, i.e. support for permissions who change according to time, physical location, access method,... For example, one might want to allow access to a patient's medical record only for the patient himself and his GP (General Practitioner), unless the GP is on holiday, in which case access would be granted to another GP, and finally, grant access to all the A&E (Accident and Emergency) doctors in case of an emergency. Or, one could want to give access to critical data to a user, unless he's connected to the system via an unsecure connection, in which case his set of permissions would be drastically reduced.

3.6.1 LBAC: Lattice-Based Access Control

Lattice-Based Access Control is another type of access control model. It has first been formalised by Denning in [Den76]. The general idea is to combine `objects` and `subjects`. A lattice describes the levels of security that an object has and that a subject can have access to. A subject can only access an object if the subject has a security level that is equal or greater than the object's security level.

3.6.2 OrBAC: Organization-Based Access Control

While the previous access control mechanisms can easily deal with static permissions, they show their limits when facing dynamic permissions that are build on contextual rules. Organization-Based Access Control was designed to address those specific issues. It brings a new abstraction layer to the usual user/role/permission entities, by introducing new concepts. This way, OrBAC is able to deal not only with contextual rules, but also with several organisations whose rules would conflict in a RBAC model.

3.6.3 And many more...

The list of access control mechanisms discussed in this chapter is not complete. Many other models have been proposed and used, but discussing all of them is out of the scope of this chapter. Some of those undiscussed mechanisms include Attribute-Based Access Control (ABAC) [YT05], Task-Role Based Access Control (T-RBAC) [OP00], . . .

Chapter 4

UMLsec

4.1 What is UMLsec

UMLsec [Jür05] is an extension of UML [OMG01] that allows one to define security properties and to formally check a model against those properties.

We will not describe all the UMLsec properties in this chapter, because we will not use all of them and because UMLsec is not a limited set of properties: anyone is welcome to define new properties to address new security concerns. Instead, we will focus only on the ones that we will use later.

The following properties have been extensively described in [Jür05], except the « permission » properties, that have been described in [JLW05]. The descriptions provided here is a summary of what can be found there, and are therefore largely inspired by those two publications, although some personal contributions have been added to extend some UMLsec properties.

UMLsec uses the standard UML extension mechanism, stereotypes and tagged values, to describe security properties. It is important to note that even though some stereotypes and tagged values have the same name, their purpose is different.

4.1.1 RBAC rules using UMLsec

UMLsec includes a way to define RBAC policies on activity diagrams. In order to do that, we need three tagged values: {role}, {right} and {protected}.

{role} is a list of pairs *(actor, role)* that assigns a *role* to an *actor* in the activity diagram.

{right} is also a list of pairs *(role, action)* that gives all the actors having a *role* activated the permission to perform an *action*.

Finally, {protected} is used to label states in the activity diagram that are protected. Only a user that has a role that allows him to perform the action will be able to perform it.

An activity diagram S meets the RBAC requirements if and only if, for every actor A in S and every activity a in the swim-lane of A in S, there exists a role R such that (A, R) is a value of {role} and (R, a) is a value of {right}.

Figure 4.1 gives an example of an activity diagram with RBAC requirements.

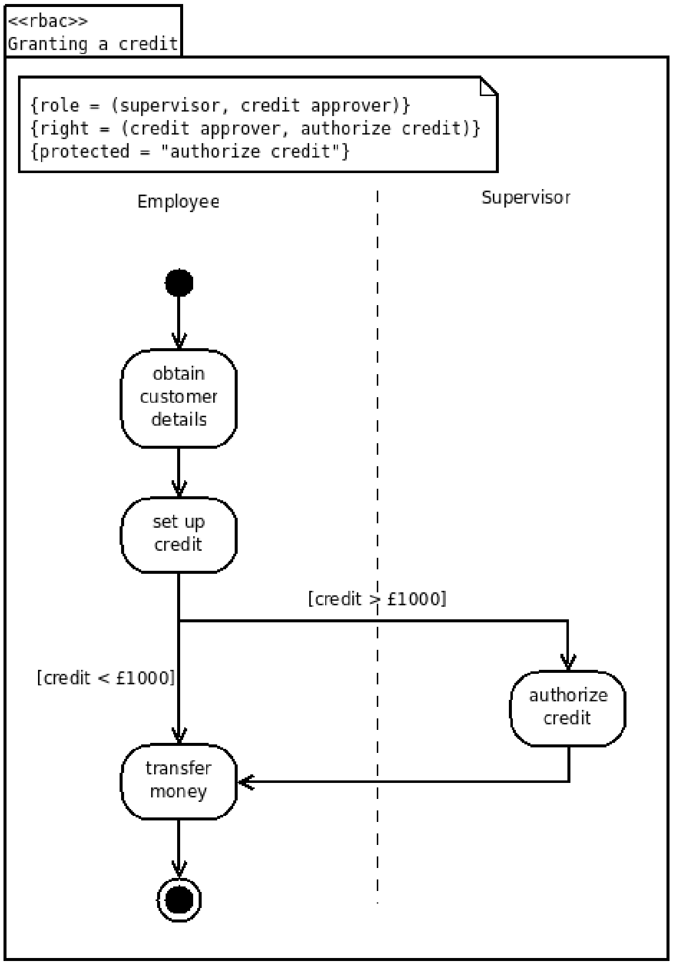


Figure 4.1: A simple activity diagram with the « rbac » property

Those requirements do not include some usual properties like Separation of Duty or role hierarchies. Those will be discussed in section 4.3, as an extension to the existing RBAC rules as defined in UMLsec.

4.1.2 « guarded » properties

The « guarded » requirements are used to label subsystems where access to some objects needs to be restricted. Any object that is labelled with the « guarded » stereotype can only be accessed through the objects specified by the tag {guard} attached to the « guarded » object.

The « guarded » requirements are inspired by the Java 2 Security Architecture framework (that is part of Java since Java 1.2).

Every object whose access needs to be restricted is stereotyped with « guarded », as well as the {guard} tagged value indicating the name of the statechart describing an object that takes care of the passing of references to the protected object. We assume that there is no other way to get a reference to the protected object.

The following example (inspired by the example in [Jür05, pp.65-67]) illustrates the « guarded » property. Figure 4.2 is a simple class diagram without any access restrictions. There are three classes: *Client*, *Account* and *ATM*. We want to restrict access to the *Client* and the *Account* classes.

Client
Account
ATM

Figure 4.2: A simple class diagram, without access restrictions

Figure 4.3 illustrates those restriction: Both the *Client* and the *Account* classes are stereotyped with « guarded ». They are also labelled with a {guard} tag, containing the name of the statechart describing the behaviour of the class that handles the passing of references: *CliAccess* for *Client*, and *AccAccess* for *Account*. There are also two new classes in the class diagram: *CliAccess* and *AccAccess*, as well as a *JavaSecArch* class (associated with its corresponding statechart) to describe the security architecture.

Now, every object of type *Client* or *Account* is protected, and the only way to get a reference to them is to use the associated *CliAccess* and *AccAccess* classes.

4.1.3 « permission » properties

The « permission » properties are a more complex and fine-grained way of restricting access than « guarded » since it is not limited to classes, but can also define permission requirements on operations.

« permission » properties are defined on both a class diagram and a sequence diagram. Consistency between those two diagrams is, of course, mandatory.

Every object can require permissions for another one to perform operations, and it can also have permission to perform operations on other objects.

Class diagram

Each object that requires permissions or that gets permissions to perform operations on other objects at instantiation time is stereotyped « permission-secured ».

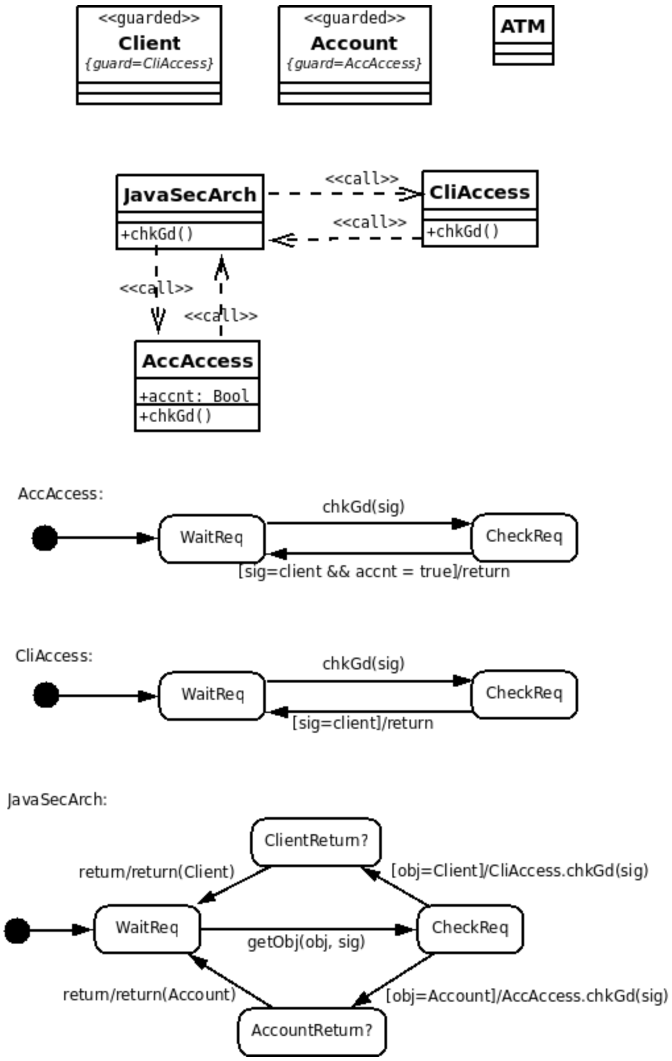


Figure 4.3: The same class diagram, with « guarded » properties

The {permission} tagged value, associated with a « permission-secured » class, is the set of permissions that any object of this type gets at instantiation time. A permission is a pair $(class, permission)$, where *class* is the class on which the permission *permission* is granted.

Each method or attribute that needs to be protected is stereotyped « permission-check », and the required permissions are listed in the {permission} tagged value. Note that when the {permission} tag contains multiple permissions, *all* of them are required.

In addition to the {permission} tag and the « permission-check » stereotype, one can add the {no_permission_needed} tag, that contains a list of classes whose objects do *not* need to have the required permissions to access the protected element.

Finally, permissions can also be delegated to another class, by using the {delegation} tagged value, which contains a set of delegation rules $(class, permission, role/class)$, where *class* and *permission* describe the permission that can be delegated, and $role/class$ describes the class to which the permission can be delegated. Of course, a class can only delegate permissions that it already has.

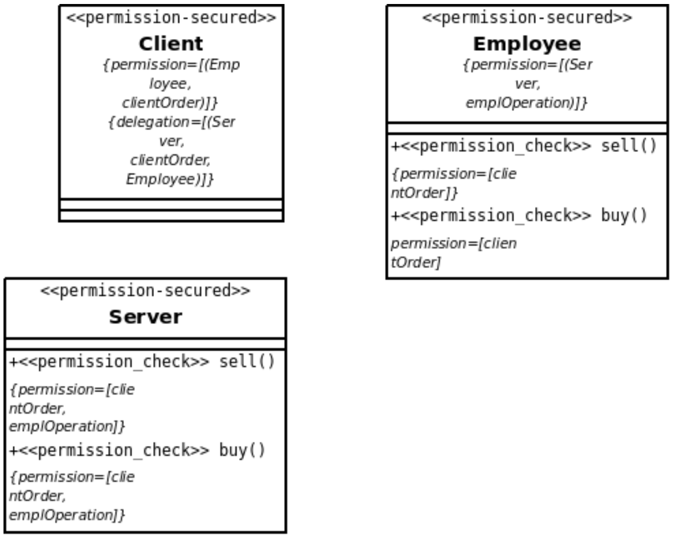

<<permission-secured>>
Client
{permission=[(Emp loyee, clientOrder)]}
{delegation=[(Ser ver, clientOrder, Employee)]}
<<permission-secured>>
Employee
{permission=[(Ser ver, emplOperation)]}
+<<permission_check>> sell()
{permission=[clie ntOrder]}
+<<permission_check>> buy()
permission=[clien tOrder]
<<permission-secured>>
Server
+<<permission_check>> sell()
{permission=[clie ntOrder, emplOperation]}
+<<permission_check>> buy()
{permission=[clie ntOrder, emplOperation]}

Figure 4.4: Class diagram with the « permission » property

Let's illustrate this with an example. Figure 4.4 is a class diagram with the « permission » stereotypes. There are three classes: *Client*, *Employee* and *Server*.

The *Client* class doesn't require any permission on any of its attributes or operations, but every object of type Client gets one permission at instantiation time, as listed by the {permission} tag: the *clientOrder* permission on the *Employee* class. The {delegation} tag also allows Client objects to delegate this permission to any object of type *Employee*.

The *Employee* class also gets a permission (*emplOperation* on the class *Server*). Moreover, it also has two operations that are protected. The first one, *sell()*, requires the *clientOrder* permission to be called, and the second one, *buy()*, also requires the *clientOrder* permission to be called.

Finally, the *Server* class doesn't get any permission, but has two operations that require the same set of permissions: *clientOrder* and *emplOperation*.

Sequence diagram

Like in the class diagram, each object that is protected or that gets permissions at instantiation time is stereotyped « permission-secured ».

If an object gets permissions at instantiation time, it is labelled with a {permission} tag listing those permissions. The {permission} tag's syntax is similar to the syntax of the {permission} tag used in the class diagram. If necessary, the {delegation} tag must also be specified, like in the class diagram.

Each call to a protected method needs to be stereotyped with « permission-check » and labelled with a {permission} tag that contains the list of the necessary permissions.

The delegation process is described using the « certification » stereotype, and an associated {certificate} tagged value. The delegation takes place during a method call on the object to which we want to delegate a permission. This method call is stereotyped with « certification », and labelled with the {certificate} tag. The syntax of the {certificate} tag is a 7-uple [JLW05]: $certificate = (e, d, c, o, p, x, s)$, where

- e is the emitting object
- d is the delegate object
- c is the delegate class
- o and p describe the permission that is being delegated. o being the object on which the permission can be used, and p the permission itself.
- x is a timer. Since sequence diagrams don't provide time management possibilities, this is actually the number of messages during which the certificate, and thus the delegated permission, will be valid. -1 means there's no validity limit.
- s is a sequence number that is used to make sure that the same certificate can not be used several times. -1 is used for certificates that can be used more than once.

Here, d and c cannot be used together. Exactly one of them has to be set to *null*. d will be used when the name of the object is known, and c will be used when it is not clear which object will get the permission delegation.

Figure 4.5 shows an example of a sequence diagram with « permission » properties. It is consistent with the example used for the class diagram.

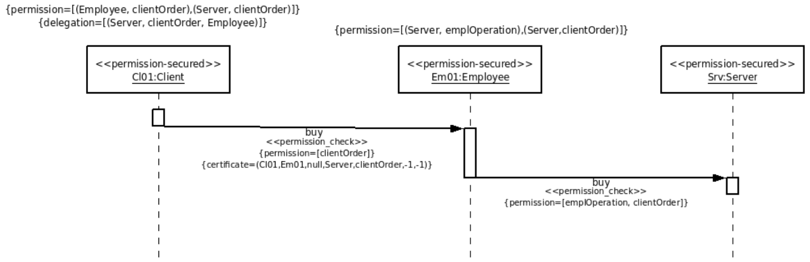

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation),(Server,clientOrder)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
{certificate=(Cl01,Em01,null,Server,clientOrder,-1,-1)}
buy
<<permission_check>>
{permission=[emplOperation, clientOrder]}

Figure 4.5: Sequence diagram with the « permission » property

We have three objects here: *Cl01*, which is a *Client*, *Em01*, which is an *Employee* and *Srv*, which is a *Server*.

Cl01 has the *clientOrder* permission that he can use on and *Employee* object, and can delegate this permission to an *Employee* object. *Em01* has the *emplOperation* on any *Server* object, and *Srv* does not have any permission.

The sequence is really simple: *Cl01* calls the *buy* operation on the *Em01* object, and then *Em01* calls the *buy* operation on the *Srv* object. Each call is stereotyped with « permission-check », and labelled with the necessary permissions.

Additionally, *Cl01* delegates a permission to *Em01* during the first operation call, using the {certificate} tag. The certificate specifies that the permission is delegated to an *Employee* object, without timer and without any sequence number, which means that the delegation can be used several times.

Consistency

Since the « permission » is defined over two different diagrams, we need to make sure that there is no contradiction between those. A few simple rules are described in [JLW05] to help make sure that the diagrams are consistent.

First, every object in a sequence diagram is labelled with the « permission-secured » stereotype if and only if it is the instance of a class that is also stereotyped « permission-secured » in the corresponding class diagram. And, of course, the associated {permission} and {delegation} tags have to be consistent: every permission or delegation in the sequence diagram has to refer to an object whose type is the corresponding class in the class diagram's {permission} tag.

Finally, the consistency of methods has to be checked: the needed permissions, as defined in the class diagram, have to match the required permissions described in the method calls in the sequence diagram.

4.1.4 « secure links » properties

« secure links » requirements can be defined on a deployment diagram. A deployment diagram has nodes, that are connected using links. nodes can contain components, and those components can be linked together using dependencies.

UMLsec defines several stereotypes for the deployment diagrams, that are used to describe the links and nodes, and to define requirements on the messages exchanged along dependencies.

Links can be described with one of the following stereotypes: « Internet », « encrypted », « LAN », or « wire ». nodes can be described with one of the following stereotypes: « smart card », « POS Device », or « issuer node ». For the « secure links » property, only the links stereotypes will be considered.

« secure links » requirements are always defined in association with an adversary. The « secure links » stereotype labels a deployment diagram, and has an {adversary} tagged value associated with it. The {adversary} tag defines an adversary, that can perform operations on nodes and on links. For each type of link, the adversary can perform a set of actions $LA \subseteq \{read, insert, delete\}$. And for each type of node, the adversary can perform a set of actions $NA \subseteq \{access\}$.

Finally, three stereotypes allow one to describe properties for the messages exchanged along dependencies on a deployment diagram S. Those properties are requirements regarding the set of actions an adversary A can perform on a link l:

- « secrecy »: $read \notin threats_A^S(l)$
- « high »: $threats_A^S(l) = \emptyset$
- « integrity »: $insert \notin threats_A^S(l)$

The « secure links »property is fulfilled if and only if, for every link l in a deployment diagram S, with an adversary A, $threats_A^S(l)$ does not contain any action forbidden by the stereotypes associated with l.

Figure 4.6 illustrates the « secure links » property defined on a deployment diagram. The default adversary used in the example is described in figure 4.7.

4.1.5 The « critical » stereotype

Before we can discuss the « secure dependency » property, we first need to introduce the « critical » stereotype, that we will use to label objects or subsystems containing data we want to protect. « critical » is used to label objects or subsystem instances containing critical data. The details of what “critical” exactly means depend on the associated tagged value, which can be either {authenticity}, {fresh}, {high}, {integrity} of {secrecy}. We will only discuss the {high}, {integrity} and {secrecy} tagged values here, since the other ones will not be needed for the « secure dependency » property.

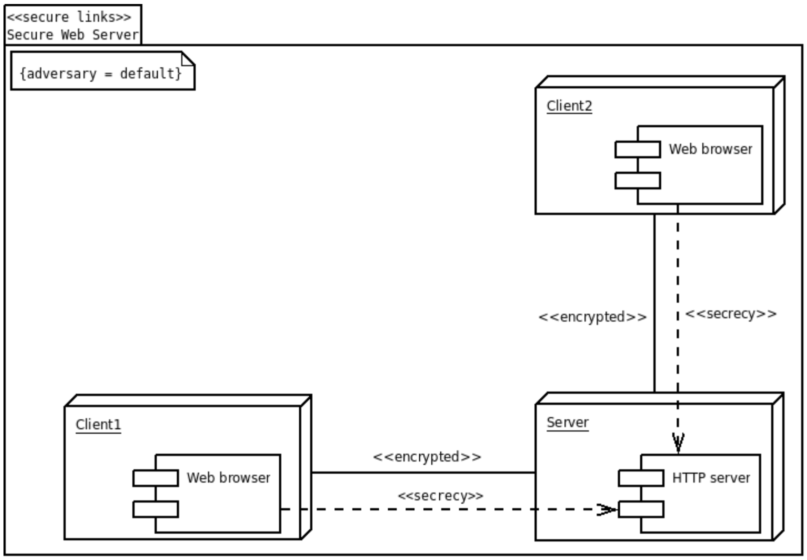


Figure 4.6: deployment diagram with « secure links » property

Link type	$threatsS_{default}$
Internet	$\{read, insert, delete\}$
Encrypted	$\emptyset$
LAN	$\emptyset$

Figure 4.7: Default adversary

{high}

The {high} tagged value contains the names of messages that are supposed to be protected by the stereotypes « no down-flow »and « no up-flow ».

{integrity}

{integrity} is a set of pairs (v, E) where v is the name of an object that needs to be protected, and E the set of expressions that can be assigned to v.

{secrecy}

Finally, {secrecy} contains a set of expressions, attributes or message argument variables of the current object (actually, their name) whose secrecy should be protected.

4.1.6 « secure dependency » properties

The « secure dependency » requirements are defined on any kind of static structure diagram. Its goal is to make sure that there will be no leakage of critical information occurring because of a communication between two objects.

Critical data is labelled with the « critical » stereotype, together with one of the following tagged values, that have been discussed in section 4.1.5: {high}, {integrity} or {secrecy}.

The formal definition of the « secure dependency » property is the following, as described in [Jür05].

If there is a « call » or « send » dependency from an object or subsystem C to an interface I of an object or subsystem D, then two conditions have to be fulfilled:

- For any message name n in I, n appears in the tag {high} (resp. {integrity} resp. {secrecy}) in C if and only if it does so in D.
- If a message name in I appears in the tag {high} (resp. {integrity} resp. {secrecy}) in C then the dependency is stereotyped « high » (resp. « integrity » resp. « high »).

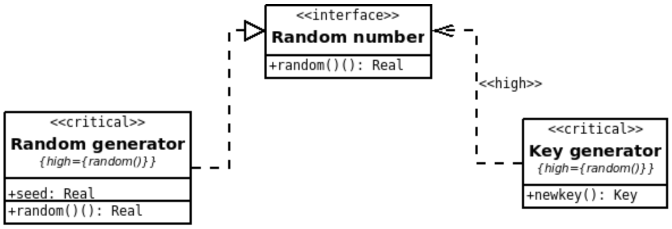


Figure 4.8: The « secure dependency » property

Let us illustrate this with an example, inspired from [Jür05]. Figure 4.8 describes a class diagram containing two classes and one interface. The interface provides a random number generator. This interface is implemented by a class, and used by the other one (through the « secrecy » stereotyped dependency).

The first condition is fulfilled: the *random()* method, provided by the interface, is critical in both classes, and appears in the {high} tagged value in both classes too.

The second condition is also fulfilled: the dependency between the interface and the class using it is stereotyped with « secrecy ».

4.2 Interactions between stereotypes

UMLsec is about modelling security properties on a UML model, but each property is independent from the others, and, most of the time, a property is defined on only one or two diagrams.

However, an UML model usually contains a lot of diagrams: class diagrams, sequence diagrams, activity diagrams, . . . All those diagrams describe the same system.

Since UMLsec properties are only defined on one or two diagrams, it might happen that a property that is defined on a particular diagram actually has an effect on another one, but without specifying it. We tried here to identify some of those properties, and to use other existing UMLsec properties to model their impact on other diagrams.

4.2.1 Generation of UMLsec stereotypes from a « rbac » stereotype

The « rbac » stereotype can be used to specify an access control policy in an activity diagram, as described in section 4.1.1. However, other diagrams, especially sequence diagrams and class diagrams, might contain information that will be impacted by access control rules: in a sequence diagram, calls to a protected method are protected by access control mechanisms, and in a class diagram, knowing which objects or methods are protected would be a really interesting feature for the developer.

For this purpose, we explored four different solutions, that we discuss here. The first one uses the « guarded » mechanism defined in UMLsec. The second one uses the « permission »mechanism, also defined in UMLsec. Finally, the third one uses the « permission » mechanism too, but differently, in order to make the sequence diagrams easier to read.

In order to compare the three solutions, we will use the same, simple model each time. The model describes a simplified version of a credit system for a bank. An employee can set up a credit for a customer, but large credits have to be approved by a supervisor. This example is a more detailed version of the « rbac »example from [Jür05, pp. 55-56].

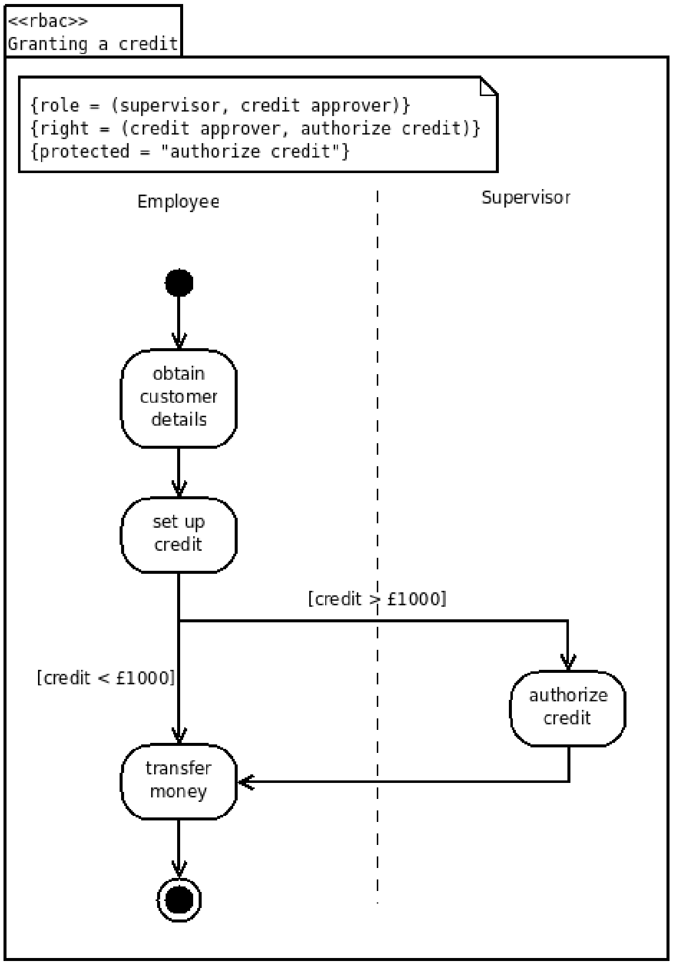


Figure 4.9: Activity Diagram describing RBAC rules for a credit system

Figure 4.9 is the original activity diagram describing the RBAC properties. Any employee can set up a credit after getting the customer's details, but only a supervisor can approve large credits of more than £1000.

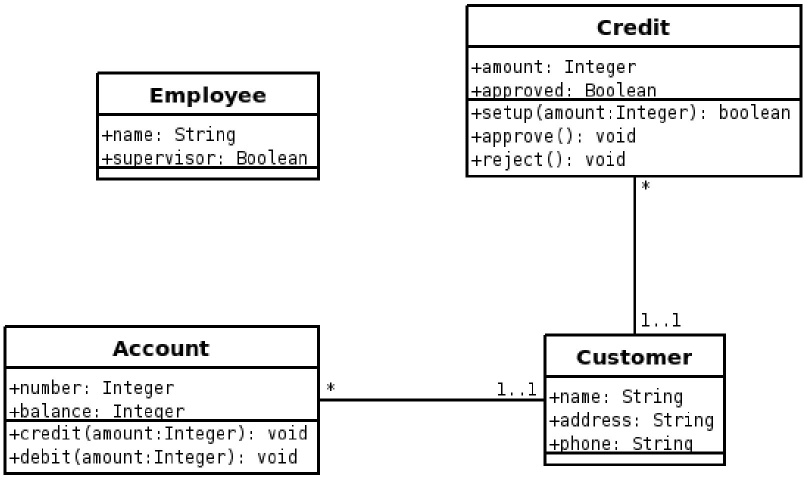


Figure 4.10: Class Diagram describing the credit system, without access control constraints

Figure 4.10 is the original class diagram, without access control information. Employees can be supervisors. There are customers, that can have several bank accounts. They can also get several credits.

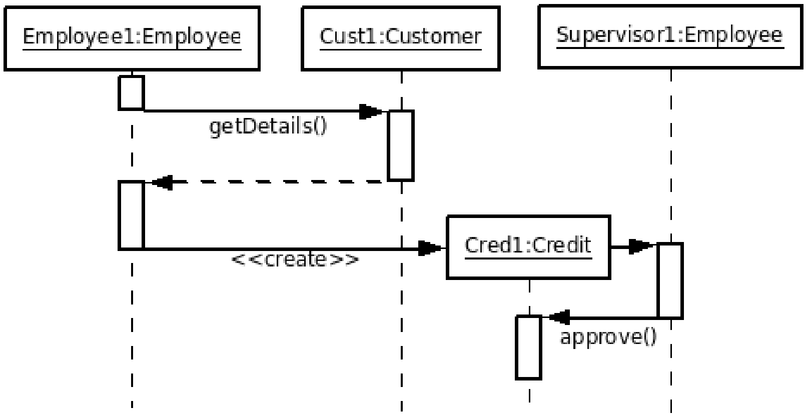


Figure 4.11: Sequence Diagram describing the setup and approval process for a large (>£1000) credit

Figure 4.11, the original sequence diagram, also without access control information, describes the sequence of actions for the creation by an employee that is **not** a supervisor (*Employee1*) of a large credit (*Cred1*) for a customer (*Cust1*), and the following approval of the credit by a supervisor employee (*Supervisor1*).

For each option we considered, we will show the class and sequence diagrams with the access control stereotypes added by the process described.

Option 1

The « guarded » mechanism was defined in section 4.1.2. It allows one to restrict access to protected objects. It is inspired by the Java 2 Security Architecture and its notion of Guarded Objects, which makes it very useful for anyone who wants to use the Java 2 Security Architecture to implement its access control policies: the model can be translated into code easily.

However, it suffers a **major restriction**: restrictions can only be defined on objects, and we would like to be able to be more specific, and chose which restrictions to apply on each method individually.

As we can see on figure 4.12, the « guarded » properties have been added to the *Credit* class in the original class diagram, and a few new statechart diagrams have been created. The sequence diagram, however, has not been modified: the « guarded » doesn't specify anything for sequence diagrams. Also, note that there is no way to tell which methods should be protected, and which ones should not.

Another problem with this option is that it is closely related to the Java 2 Security Architecture. If it makes implementation easier when using the Java 2 Security Architecture, it might get more complicated, or even confusing, if someone wants to use another security architecture (possibly using another programming language).

Option 2

The second option simply adds the access control restrictions defined in a activity diagram into the class diagrams and sequence diagrams, where needed. We only use the « permission » stereotypes here.

In order to represent the access control process itself, we chose to use a model of the JAAS architecture, which was included in Java since Java 2. Each call to a protected method is replaced by a call to the JAAS architecture, which checks that the caller has the needed permissions to perform the operation, performs the operation on the caller's behalf, and sends the return value back to the caller, as we can see on figure 4.13.

JAAS requires that each protected method should be encapsulated in an *action object*, that will check that the caller has the permission to call the method. Each time an object wants to call a protected method, it has to create a `PrivilegedAction` object, and ask it to run the protected method using the `doAsPrivilieged(...)` method. The `PrivilegedAction` object first checks that the caller has the necessary permissions for the required method call by calling the `AccessController` object, and then performs the method call on behalf of the caller. After that, the return value of the method is sent back to the caller. A more detailed description of the JAAS architecture can be found in section 5.3.1.

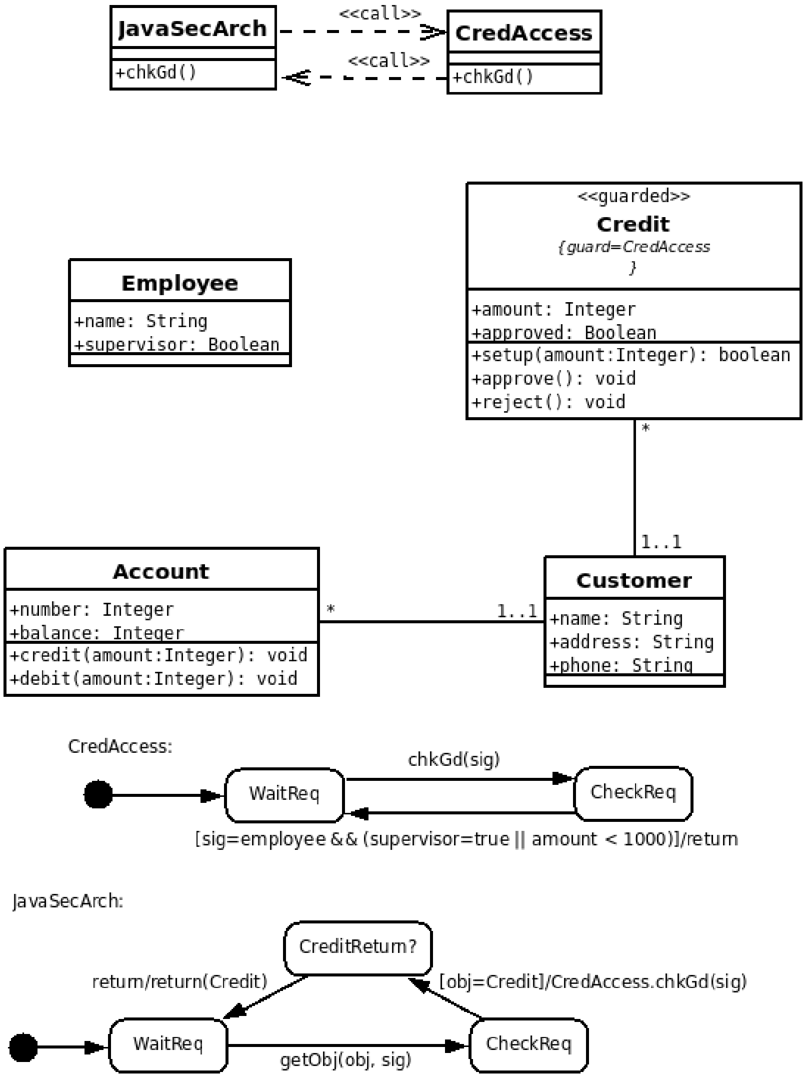


Figure 4.12: The class diagram with the « guarded » property, and the statechart diagrams describing its behaviour

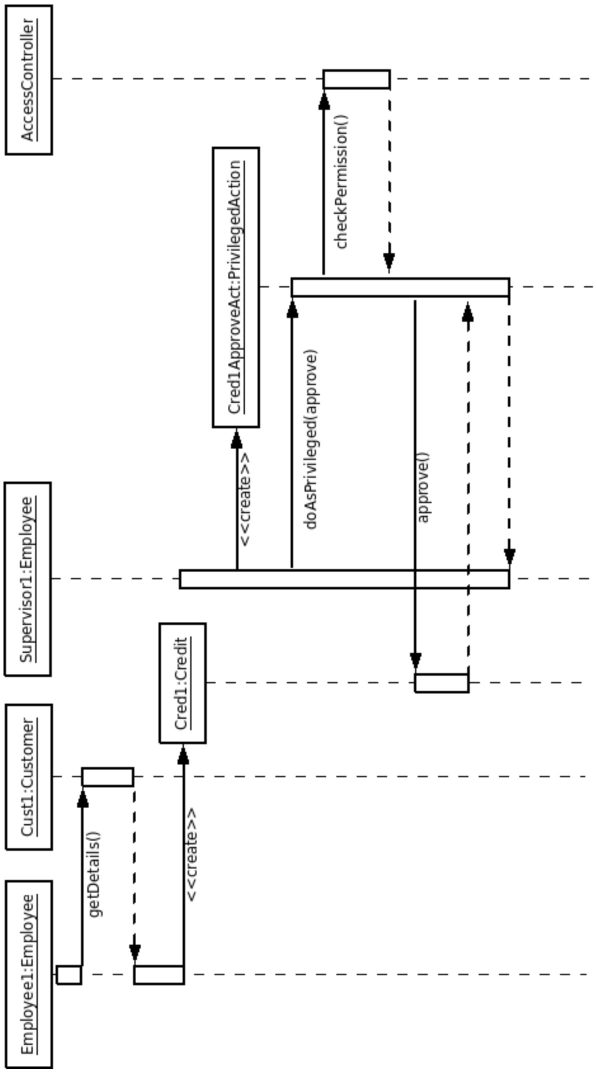


Figure 4.13: sequence diagram describing option 2

Figure 4.13 does not represent the authentication process, which has to be done before the authorisation process described.

The class diagram is also updated, as shown on figure 4.14, with the addition of « permission » stereotypes, and a few new classes for the JAAS architecture.

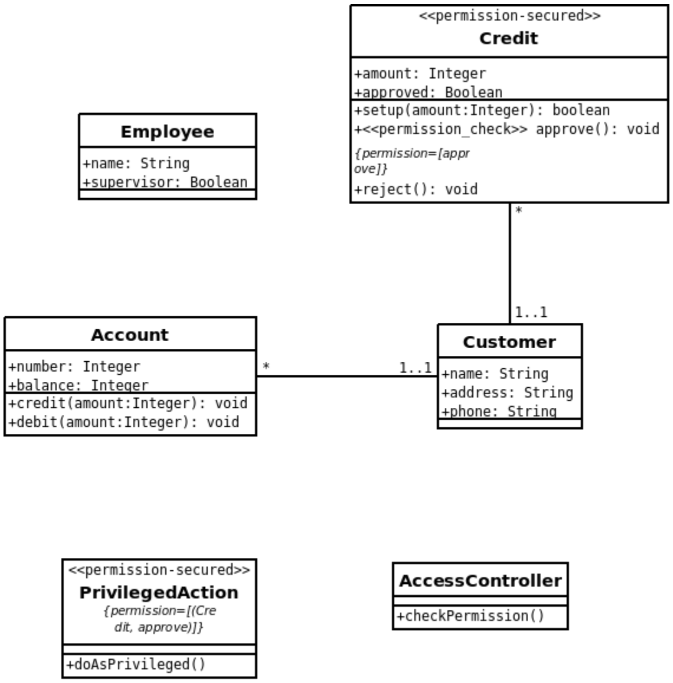


Figure 4.14: class diagram describing option 2

This solution has **two major problems**.
First, it adds quite a lot of messages exchange to the sequence diagrams, which quickly makes those almost impossible to read, even on a really simple example like figure 4.13. Real life examples can get really, really hard to understand, because of all this 'noise' added to the original diagrams, since an object has to be created for each call to a protected method.

Second, it suffers the same limitation as the previous option: it is great for anyone who wants to implement the solution using the JAAS architecture, but it can get confusing for someone who would want to use another authentication and authorisation system.

Option 3

The third and last option is derived from the previous one, and is designed to address the limits we just discussed. The idea is to define a new stereotype, « auth », to label an authentication and authorisation object that would replace the complex JAAS architecture. Instead of multiple calls to multiple objects for each protected method call, we just need to call this special object, as shown on figure 4.15.
The details of the authentication and authorisation processes can then be described in separate

sequence diagrams. The authorisation au authentication object will also be labelled with the {authentication} tag and the {authorization} tag, that contain a reference to the sequence diagram describing the authentication process (for the {authentication} tag) and a reference to the sequence diagram describing the authorisation process (for the {authorization} tag). Those sequence diagrams are not shown in this example, since there is no way to derive it from the original RBAC-enabled activity diagram. This has to be added manually by the developer.

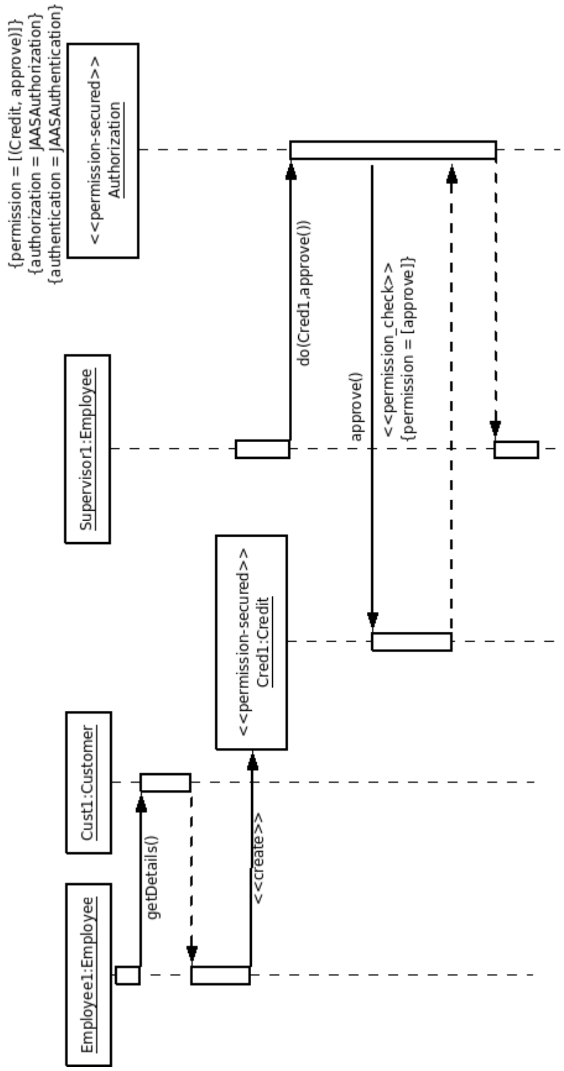


Figure 4.15: Sequence diagram describing option 3

The class diagram has to be updated accordingly, in order to fulfil the « permission » requirements on consistency between the class and the sequence diagrams. As we can see on figure 4.16, a new class has been added for the authorisation and authentication mechanisms. That class is tagged with the two newly introduced tags, {authentication} and {authorization}. The class also gets permissions to access all the protected methods at instantiation time through the {permission} tag, and it is therefore stereotyped with « permission-secured ».

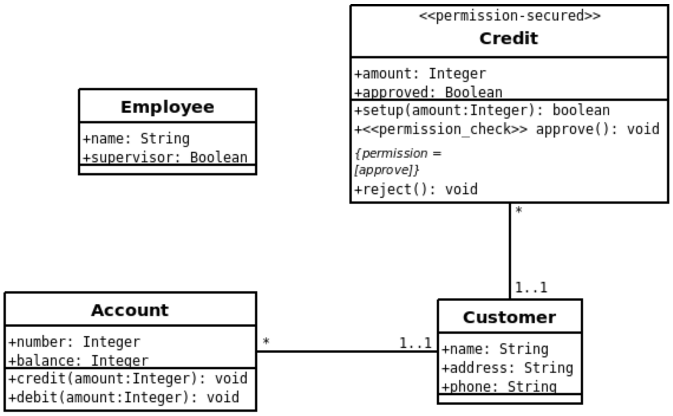


Figure 4.16: Class diagram describing option 3

Finally, all the operations whose access need to be restricted (here, the *approve()* operation only) are stereotyped with « permission-check », and labelled with a new permission. Every class that contains such operations is also stereotyped with « permission-secured »(here, the *Credit* class only).

The solution has two advantages over the previous ones: it makes the diagrams really easier to read, but it also allows one to choose which authentication and authorisation method to use: JAAS or any other one will do.

Conflicts with existing « permission » stereotypes

When generating « permission » stereotypes on a sequence diagram or a class from an activity diagram with « rbac » stereotypes, conflicts can arise with existing « permission » stereotypes on the sequence diagram. See section 4.4.1 for detecting and solving them before the generation of the new stereotypes.

4.3 Extending the « rbac » requirements

In order to model some more advanced features of Role-Based Access Control, the « rbac » property has to be extended. Some of the features we add to the « rbac » property exist in the NIST RBAC standard (see section 3.5.3, and others (like negative permissions) don't.

4.3.1 Supporting hierarchical roles

We can extend the Role-Based Access Control stereotype to support hierarchical roles too. We can do that by simply adding a new tagged value, like : {hierarchy = (parent_role, child_role)} . This means that the parent_role role is a parent of the child_role role.

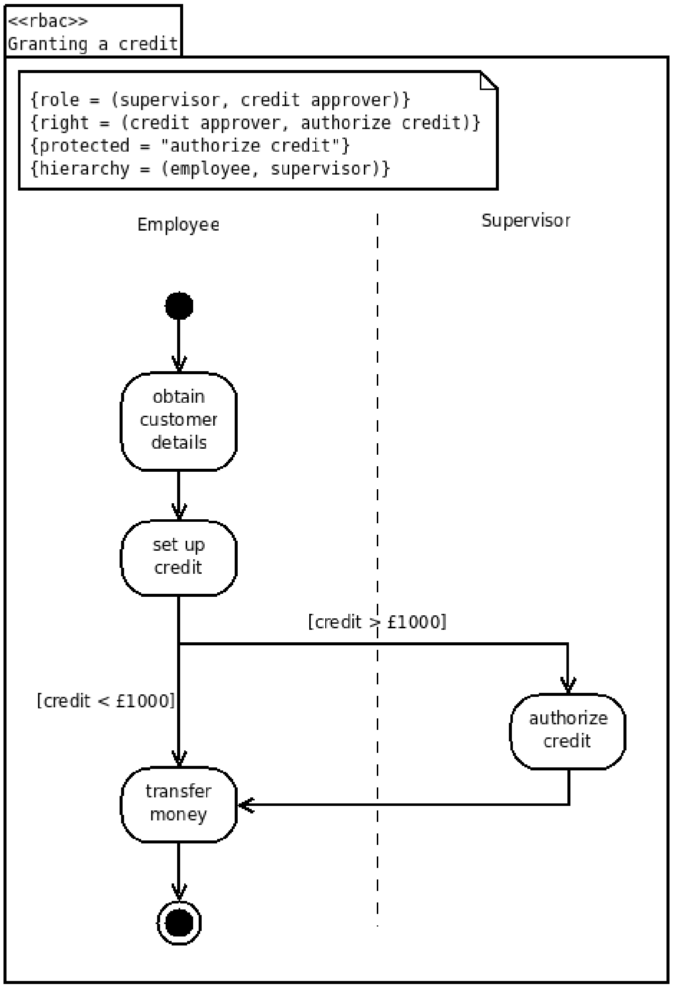

<<rbac>>
Granting a credit
{role = (supervisor, credit approver)}
{right = (credit approver, authorize credit)}
{protected = "authorize credit"}
{hierarchy = (employee, supervisor)}
Employee
Supervisor
obtain customer details
set up credit
[credit > £1000]
[credit < £1000]
authorize credit
transfer money

Figure 4.17: Activity diagram with a hierarchy relation between two roles

This way, we can extend the activity diagram used in section 4.2.1 to express the fact that a supervisor is an employee that can accept or deny large credits to customers. Figure 4.17 is the updated version of the activity diagram. We just added a new tagged value describing the relationship between Employee and Supervisor.

4.3.2 Separation of Duty

Separation of Duty is another key feature in Access Control models. This can simply be added to the existing « rbac »specification by adding a new tagged value, {sod}, with the following syntax : {sod = (action_1, action_2)}, which means that a user cannot be granted permissions for both action_1 and action_2.

4.3.3 Negative permissions

Negative permissions have been described in section 3.5.3. Those have **not** been standardised in the NIST RBAC standard. Adding negative permissions support to the « rbac » specification is just about adding a new tagged value to the set of existing ones: {neg permission}. Its syntax is : {neg permissions = (role, action)}, which means that a user with the role ***role*** activated can not perform action ***action***.

Using negative permissions can lead to conflicts with the usual way of defining permissions. One should be very careful when defining negative permissions, and the following rules will apply for an « rbac » property to be defined correctly in UMLsec:

- if a role ***roleA*** contains a permission to perform an operation *operationA*, then it cannot contain a negative permission that would deny the right to perform *operationA*

- if a user is granted a role ***roleA*** containing a permission to perform an operation *operationA*, he can't be granted another role ***roleB*** that contains a negative permission for the same operation *operationA*

Extra care must be taken when using negative permissions together with role hierarchies: not only should one deal with the permissions given by a role, but also with the inherited permissions:

- if a role definition ***roleA*** grants a permission to perform operation *operationA* to its users, than there cannot be another role ***roleB*** inheriting (directly or indirectly) from ***roleA***'s permissions that defines or inherits of a negative permission on operation*operationA*

- if a user is granted a role ***roleA*** containing or inheriting a permission to perform an operation *operationA*, then the user can't be granted another role ***roleB*** containing or inheriting a negative permission for the same operation *operationA*

Since the UMLsec « rbac » property only support the activation of *all* the roles at the same time, there is no way to activate only a subset of a user's roles, or to deactivate and/or reactivate roles at any time. Therefore, there is no need to handle the possibility for a user to have two conflicting roles that would never be activated at the same time.

4.4 Conflicts between UMLsec properties

Usually, UMLsec properties are defined separately. Often, they are even defined on different diagrams that are part of the same model. Thus, it can be hard to identify and solve potential conflicts between two different properties.

When a model is made of several diagrams, we first have to make sure that there is no conflict between them before adding the UMLsec properties. This is out of the scope of this thesis, but research in that field can be found in [ELF08].

4.4.1 Conflicts between « permission » and « rbac »

Multiple conflicts can occur between an activity diagram with « rbac » stereotype and a class diagram and/or a sequence diagram with « permission » stereotype, because of those properties' nature: they both aim at restricting access to authorised users or objects only. Fortunately, those potential conflicts are relatively easy to spot, since we defined a way to generate « permission » stereotypes in sequence and class diagrams from a activity diagram with « rbac » stereotypes in section 4.2.1, options 2 and 3.

Detecting conflicts can be done when generating « permission » properties in class and sequence diagrams from « rbac » properties in activity diagrams. A conflict occurs when an action that is authorised by the activity diagram turns out to be impossible when looking at the sequence diagram, or when an action permitted by the sequence diagram is forbidden by the activity diagram. Since the sequence diagram and the class diagram are supposed not to conflict (this should already have been checked when defining the « permission » properties), we can safely ignore the latter, and focus on the former, which provides more information about which methods can or cannot be called, and by which objects.

4.4.2 Conflicts between « guarded » and « rbac »

Just like the conflict resolution strategy between « permission » and « rbac » properties, conflicts between the « guarded »and « rbac » properties can benefit from the generation of « guarded » properties from « rbac »properties, described in section 4.2.1, option 1. We use a similar approach to detect and solve conflicts.

4.4.3 Conflicts between « permission » and any sequence diagram

If the sequence diagram is tagged with « permission » stereotypes, one just has to make sure that both the sequence diagram and the class diagram do not conflict. If the sequence diagram is not tagged with « permission » stereotypes but some roles involved in the diagram have a type whose class is marked with a « permission-secured » stereotype, then « permission » stereotypes should be added to the sequence diagram. However, if the sequence diagram is not tagged with « permission » stereotypes, and none of the roles involved it it have a type whose class is tagged with a « permission-secured » stereotype, then there is no conflict.

4.5 Automated Security Hardening for UMLsec Models

4.5.1 Overview

UMLsec allows one to define security properties on a UML model, and to check that those properties are actually enforced by the model. UMLsec can not only detect that a model does not meet some security requirements, it can also pinpoint where the problem lies.

Once the developer knows that his model does not meet all the security requirements he wants it to meet, we can either let him change his model manually, which can be a long and painful process, or we can try to automatically suggest improvements that will make the model meet the needed security requirements.

Most of the time, it will not be possible to modify the model in a completely automated way: we will need some additional input from the developer, or we will have several possible changes that may all solve the problem we identified, but that will change the model's behaviour in different way: we will then show the possible solutions to the developer, and ask him to pick up the one he wants to apply.

The following hardening strategies have been described in [MJY+09]. The examples illustrating those strategies also come from [MJY+09], although some minor changes have been done here.

4.5.2 « secure links »

The « secure links » property has been described in section 4.1.4. As we already know, we need two things to define a « secure links » property: an adversary profile, and a deployment diagram with the appropriate UMLsec stereotypes. Dependencies between components need to be labelled with the security we want to enforce for the messages carried along them, and links have to be labelled according to their types.

This property has to be checked against an adversaries, that we define by its ability to perform actions (`read, insert and/or delete`) on some link or node types. The three adversaries used in this section are common types of adversaries, but it is possible to check the « secure links » property against any type of user-defined adversary.

Default adversary

Link type	$Threats_{default}$
Internet	$\{read, insert, delete\}$
Encrypted	$\{delete\}$
LAN	$\emptyset$
wire	$\emptyset$

Figure 4.18: Default Adversary

Figure 4.18 describes the default adversary we will use in the following example. The default adversary comes from outside the organisation, and therefore has access to everything that travels unencrypted over the Internet. He can delete data that goes through an encrypted link, but he cannot read or modify it. He doesn't have access to the organisation's local network, smart-cards and POS devices.

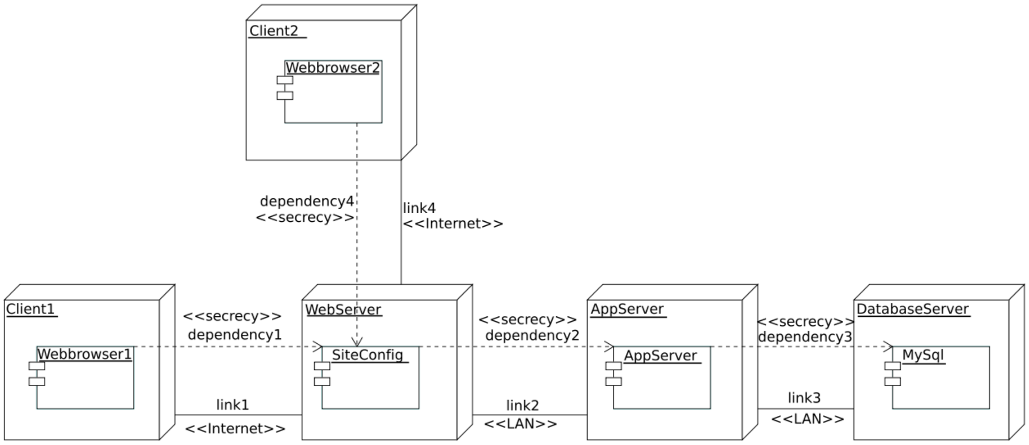


Figure 4.19: Simple web application deployment diagram

Our example is a simple web application deployment diagram, as shown on figure 4.19. Clients connect to the web server over the Internet. The organisation's local network has three nodes: the web server, that connects to the application server, itself being connected to the database server.
We want all communications between the components to be secret (we don't want an adversary to be able to read the data), which is why they all have the « secrecy » stereotype.

This model does not meet the « secure links » requirements for the default adversary we defined. Since the default adversary can read messages over an Internet link, he can read the messages exchanged between Client1 and WebServer, and between Client2 and WebServer. However, the two corresponding dependencies are labelled with the « secrecy » stereotype. Thus, the « secure links » property is violated.

There are two possible solutions: we can either lower the security requirements, and drop the « secrecy » stereotype on the two dependencies that cause problems, or we can make the two links more secure by adding encryption to the communication.

The first solution is not an acceptable one, especially when automatically modifying the model, because it will result in a new model that is less secure than what the developer expected in the first place.
The second one enforces the « secure links » property as defined by the developer by replacing the « Internet » stereotype on the links between the clients and the web server by a « encrypted » stereotype. This way, the default adversary can not read the messages anymore,

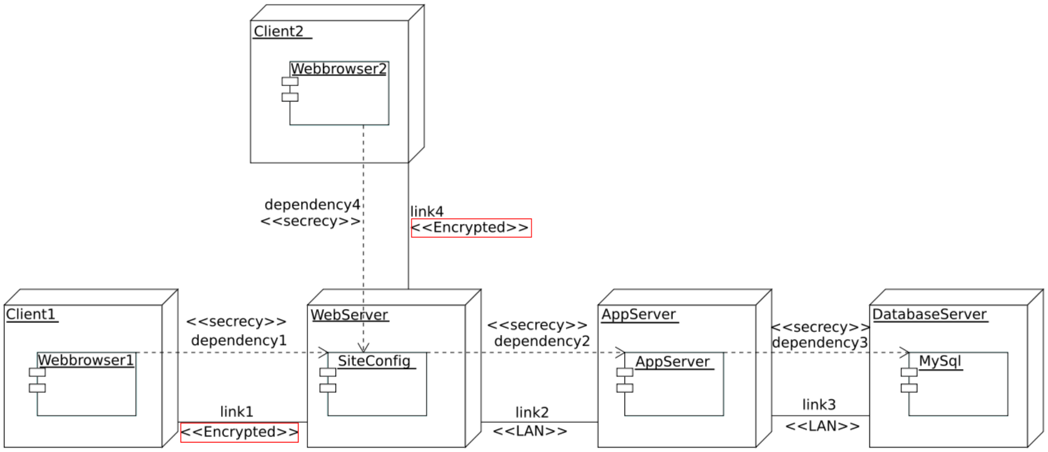


Figure 4.20: Deployment diagram meeting the « secure links » property with a default adversary

and the « secrecy » properties are fulfilled by the model. This solution is described in figure 4.20.

A stronger adversary

Our first adversary could not access a lot of information. It could only capture data over the Internet, but had no access to the organisation's internal network. Let's now define another adversary (figure 4.21) that can access information exchanged within the organisation's LAN, and see if our example still meets the « secure links » requirements.

Link type	$Threats_{insider}$
Internet	$\{read, insert, delete\}$
Encrypted	$\{delete\}$
LAN	$\{read, insert, delete\}$
wire	$\emptyset$

Figure 4.21: Insider Adversary

The difference between the new adversary and the previous one is that the new one has full access to what is exchanged over the LAN links. He can read or delete messages, and even insert forged messages. However, he still can't read encrypted messages, nor can he insert encrypted data in a communication.

The original model does not meet the « secure links » requirements, for the same reason it did not meet the « secure links » requirements with the default adversary. However, the fix proposed for the default adversary doesn't meet the « secure links » requirements either with the inside adversary: since the new adversary is now capable of reading messages exchanged on the « LAN » network, the model violates the « secrecy » property on the « LAN » links.

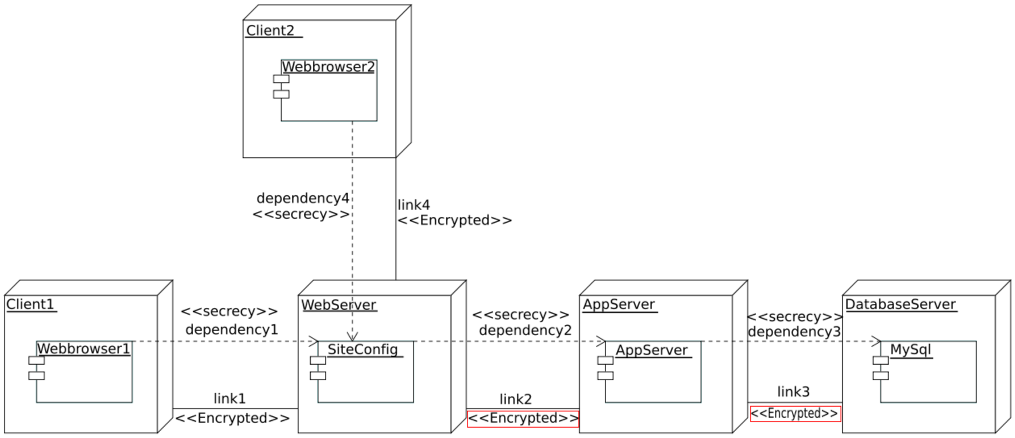


Figure 4.22: Deployment diagram meeting the « secure links » requirements with an inside adversary

The solution is similar to the previous one: we just need to make sure the adversary can not read the messages exchanged on the « LAN » links. This can be done by changing the « LAN » links into « encrypted » links, since the adversary can't read messages exchanged on an encrypted link. Figure 4.22 shows the transformed model that now meets the « secure links » requirements with an inside adversary.

An even stronger adversary : the inside adversary

Let us now define a last adversary (figure 4.23). He has now access to the encryption keys, which means that he can read, insert and delete messages even on « encrypted » links.

Link type	$Threats_{insider}$
Internet	$\{read, insert, delete\}$
Encrypted	$\{read, insert, delete\}$
LAN	$\{read, insert, delete\}$
wire	$\emptyset$

Figure 4.23: Insider Adversary with access to encryption keys

The original model doesn't meet the « secure links » requirements, nor does any of the two fixes proposed earlier, since this adversary can also read messages exchanged on an encrypted link.

There is no other solution here than lowering the « secure links » requirements, which is probably not a good option, or heavily modifying the model, which can probably not be done automatically. This is a limitation of the proposed approach, that is not capable of converting any model to a model that satisfies the « secure links » requirements regarding any kind of adversary.

4.5.3 « secure dependency »

« secure dependency » has been described in section 4.1.6. A structure diagram needs to comply with two rules in order to fulfil the « secure dependency » requirements [Jür05]: Given an interface I, a class C that implements I, and a class D that has a dependency link with I, those rules are:

- For any message name n in I, n appears in the tag {high} (resp. {integrity} resp. {secrecy}) in C if and only if it does so in D.
- If a message name in I appears in the tag {high} (resp. {integrity} resp. {secrecy}) in C then the dependency is stereotyped « high » (resp. « integrity » resp. « high »).

Thus, an unsecured model contains a structure diagram that doesn't satisfy one, or both of these two conditions.

If the first condition is violated, then one of the two classes (either the one implementing the interface, or the one using it) has an object in a tagged value associated with the « critical » stereotype (either {high}, {secrecy} or {integrity}), and that object is also provided by the interface, but the other class does not have this object in the same tagged value. The solution is simply to add the object to the tagged value of the second class. Another solution would be to remove the « critical » stereotype and the associated tagged value, but that would lower the security requirements, and is probably not a good idea.

If the second condition is violated, then there is a critical message in the calling class (the one that is linked to the interface with a dependency stereotyped with « call » or « send ») that is provided by the interface, but the dependency is not stereotyped « secrecy ». The solution is simply to add the « secrecy » stereotype to the dependency. Again, we do not see the second solution (removing the « critical » stereotype and its associated tagged value) as a suitable option, since it would lower the security requirements.

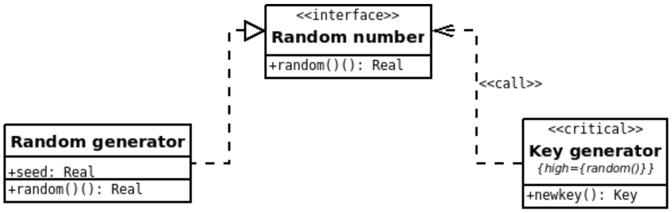


Figure 4.24: « secure dependency »property: a key generation system

Let's illustrate this with a small example (taken from [Jür05]): figure 4.24 is a class diagram describing a key generator that uses a random number generator. We have one interface, *Random number*, a class that implements this interface (*Random generator*), and another class *Key generator*, that calls the interface. The critical data is the output of the *random()* method, and the tagged value used here is {high}.

This example is incorrect regarding the « secure dependency » property. It violates the first condition since the *random()* message in the interface appears in the {high} tag in *Key generator*, but not in *Random generator*. This can easily be fixed by adding the *random()* message in the {high} tag in *Random generator*, as illustrated on figure 4.25.

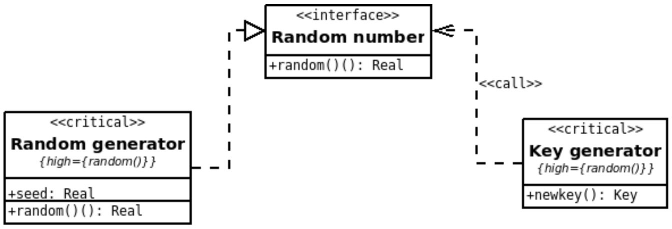


Figure 4.25: The key generation system has been fixed to meet the first condition of the « secure dependency » property

It is better, but now, the second condition is violated: *random()* appears in the {high} tag in *Key generator*, but the dependency is **not** labelled with the « high » stereotype. The solution is to add this stereotype to the dependency, and we now have a subsystem that satisfies the « secure dependency » property, as shown on figure 4.26.

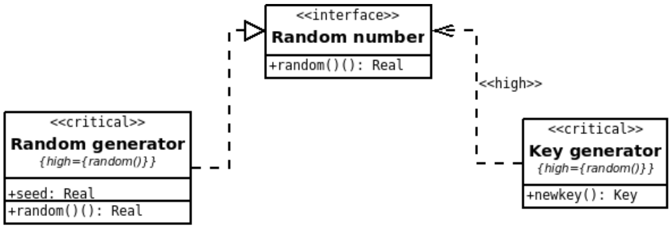


Figure 4.26: The key generation system now fulfils the « secure dependency » requirements

4.5.4 « permission »

The « permission » property is a complex one, since it is spread over two different UML diagrams. Therefore, there are four groups of possible violations of the property: errors in the class diagram, static errors in the sequence diagram, consistency problems between those two, and finally, dynamic errors in the sequence diagram, when a sequence of operations can never be completed due to lack of permissions.

The order in which the possible mistakes and how to fix them are described is the order in which the « permission » checks should be performed: first, make sure that each diagram is correct (which means that the first two steps can be swapped), then make sure that the diagrams are consistent with each other, and finally, make sure that each sequence of operation can be completed. Any other ordering would not make sense: checking for consistency before knowing that the diagrams are syntactically correct would be useless, since any subsequent modification of any diagram would require another consistency check. Similarly, if we check that all the sequences of operations can be completed before checking the consistency of the diagrams is a waste of effort, since any modification to a sequence diagram would require to re-check for its possible completion.

Errors in the class diagram

The easiest mistakes to detect and to fix are when a class (resp. an operation) is stereotyped with « permission » (resp. « permission-check ») but doesn't have any {permission} tag associated. Similarly, any class (resp. operation) having a {permission} tag, but no « permission » (resp. « permission-check ») stereotype violates the « permission » property. The fix is straightforward: one just has to add the missing tag or stereotype. However, while adding the stereotype can be automated, adding the tag requires a manual operation by the user: it is impossible to guess what he wanted to put in the tag in the first place. The user might also chose to remove the existing stereotype or tag instead of adding the missing information.

Permissions that are granted for a class that doesn't appear in the diagram are obvious mistakes. The solution is either to correct the permission (by referring to an existing class) or to remove it.

Another potential error arise when an operation requires a permission that is not granted to any object. The consequence would be that no object would be able to call the operation, and therefore the operation would be completely useless. There are two possible workarounds for this problem: either remove the required permission, or grant it to at least one object.

Delegation of permissions can also lead to property violation when a class tries to delegate a permission it doesn't have. In this case, the workaround is either to remove the delegation capability, or add the delegated permission to the list of permissions granted to the class.

Static errors in the sequence diagram

Static errors in the sequence diagram are similar to those that one can find in the class diagram. This section does **not** deal with mistakes that lead to the impossibility to perform a sequence of operations completely.

As in the class diagram, the easiest type of error arises when an object has a {permission} tag but no « permission-secured » stereotype. The solution is also to either add the « permission-secured » stereotype, or remove the {permission} tag, as well as the {delegation} tag if it exists. However, an object stereotyped with « permission-secured » but without any {permission} tag attached can be valid: an object that doesn't get any permission at instantiation time can still require permissions for one (or more) of its operations.

Again, like in a class diagram, any permission granted to an object that doesn't exist is a mistake. The solution is also either to change the permission to match an existing object in the diagram, or to remove it.

The {certificate} tag can lead to more complex issues. First, a method call stereotyped with « certification » but without any {certificate} tag attached is not valid, and a method call with a {certificate} tag attached but without the « certification » stereotype is not valid either. Two possible fixes are to remove the tag (resp. the stereotype), or to ad the missing stereotype (resp. tag). While adding the stereotype can be done automatically, adding the certificate needs to be done manually.

The {certificate} tag also has to be well-formed. Several rules apply here, for a certificate $certificate = (e, d, c, o, p, x, s)$:

- e, the emittent, must be an existing object.
- e must name the emittent object
- at least one of d (the delegate object) or c (the delegate class) must be not null. If both are not null, d must be an instance of c.
- d and/or c (the one that is not null, or both) must name the object (or its class) who receives the certificate
- o, the object on which the permission is delegated, must exist somewhere in the diagram
- p, the permission delegated, has to be required by at least one operation in o
- x, as well as s, must be an integer > 0 or -1

If any of those rules are violated, the {certificate} tag has to be adapted by the user.

Consistency errors

Consistency errors happen when the class diagram and the sequence diagram are conflicting. The rule of thumb is that there can't be more permissions in the sequence diagram than in the class diagram: an object doesn't necessarily need to hold or delegate all the permissions allowed by its class, but it certainly can't use permissions it doesn't hold at instantiation time or get through the delegation mechanism.

We will not discuss here the potential consistency issues between a "regular" (without UMLsec extensions) class diagram and a "regular" sequence diagram. Whether all the objects in the sequence diagram are instances of a class that actually exists in the class diagram, or every operation used in the sequence diagram is actually described in the corresponding class in the class diagram is out of the scope of this section. We assume that the diagrams are consistent, and will focus on consistency of the « permission » UMLsec property.

The first, easy check to perform is making sure that every permission or delegation possibility described in the {permission} and {delegation} tags of any object in the sequence diagram exists in the {permission} and {delegation} tags of the corresponding class in the class diagram. Note that if a permission (resp. a delegation) cannot exist in the object's {permission} (resp. {delegation}) tag without also appearing in the corresponding class's {permission} (resp. {delegation}) tag, the opposite can happen: it is possible for an object not to get all the permissions that its class can get. When an object gets more permissions at instantiation time than its class allows, two strategies are available: either add the missing permission to the class's {permission} tag in the class diagram, or remove the permission from the object's set of permissions. The first approach can't be done without the user's agreement, since it could weaken the security policy. However, the second approach requires to check the sequence diagram for static errors again.

Finally, consistency issues can also arise on the operation level: every call to an operation in the sequence diagram must come with the exact same set of permissions than required in the class diagram. No more (that would be useless), no less (the operation could not be performed). If there are too much permissions on the call, the solution is simply to remove the surplus. However, if there are missing permissions, they need to be added. If the caller object holds the missing permissions right before the call (because it got it either at instantiation time or by delegation), then adding the permissions to the call is enough. If it is not the case, then several strategies can be used, and the user has to chose one:

- add the permission to the set of permissions the object gets at instantiation time, but only if this is allowed by the corresponding class in the class diagram
- get the permission through a delegation, if another object can delegate the missing permission

If none of those strategies are possible, then the model needs to be changed manually by the user. Possible solutions include adding a permission at instantiation time for both the object and its corresponding class, or adding delegation capabilities so that another object can delegate the missing permission.

Dynamic errors in the sequence diagram

Finally, one needs to make sure that every sequence of operation can be completed. In other words, all the necessary permissions have been granted before being used, and all the necessary permissions need to be used. Permissions granted at instantiation time have already been discussed, but dynamically granted permissions (permissions granted using a certificate) are a more complex issue.

First, a certificate used by an object to call a protected operation must have been received by the object **before** being used.

Example

We will use a slightly modified version of an example described in [MJY$^+$09]. We have both a class diagram (figure 4.27) and a sequence diagram (figure 4.28).

The described system is simple: a *Client*, an *Employee* and a *Server* interact to buy something. The *Client* calls the method *buy()* on the *Employee*, that then calls the *buy()* method on the *Server*.

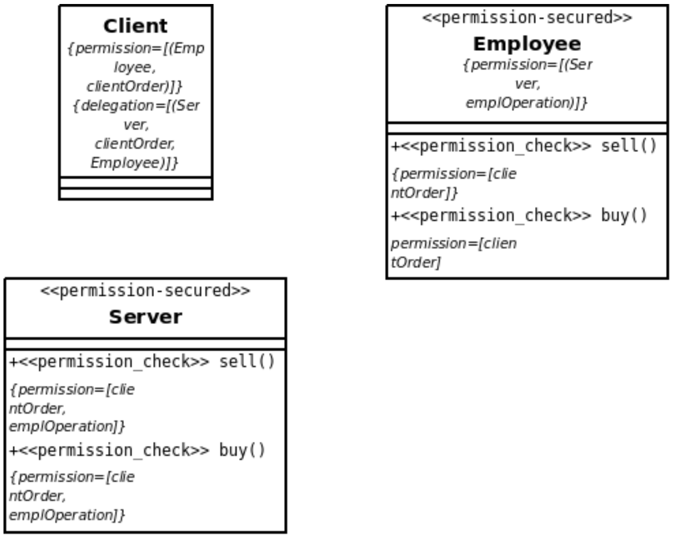

Client
{permission=[(Employee, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
<<permission-secured>>
Employee
{permission=[(Server, emplOperation)]}
+<<permission_check>> sell()
{permission=[clientOrder]}
+<<permission_check>> buy()
permission=[clientOrder]
<<permission-secured>>
Server
+<<permission_check>> sell()
{permission=[clientOrder, emplOperation]}
+<<permission_check>> buy()
{permission=[clientOrder, emplOperation]}

Figure 4.27: A simple Class diagram with the « permission » property

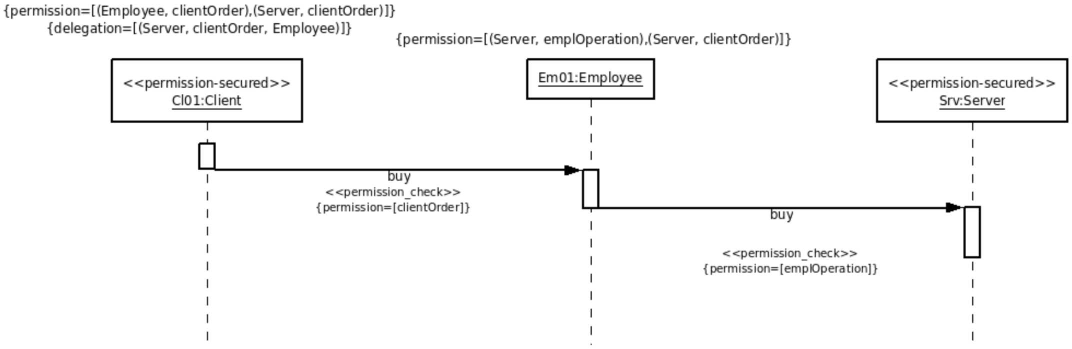

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation),(Server, clientOrder)]}
<<permission-secured>>
Cl01:Client
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
buy
<<permission_check>>
{permission=[emplOperation]}

Figure 4.28: A simple sequence diagram with the « permission » property

Let's look at the class diagram first: one of the classes, *Client*, doesn't meet the « permission » requirements: while it gets permissions and delegation capabilities at instantiation time, the « permission-secured » stereotype is missing. The solution is either to remove the {permission} and the {delegation} tags, or to add the missing stereotype. The two other classes are fine: they are both stereotyped with « permission-secured », and all the required tags and stereotypes are included. The corrected version of the class diagram is on figure 4.29, where the missing « permission-secured » stereotype has been added.

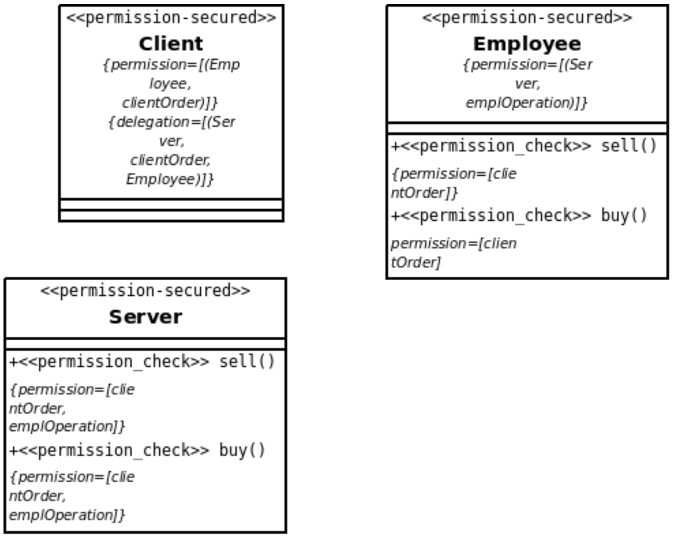

<<permission-secured>>
Client
{permission=[(Employee, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
<<permission-secured>>
Employee
{permission=[(Server, emplOperation)]}
+<<permission_check>> sell()
{permission=[clientOrder]}
+<<permission_check>> buy()
permission=[clientOrder]
<<permission-secured>>
Server
+<<permission_check>> sell()
{permission=[clientOrder, emplOperation]}
+<<permission_check>> buy()
{permission=[clientOrder, emplOperation]}

Figure 4.29: The correct class diagram, with the missing « permission-secured » stereotype

The sequence diagram also contains a mistake: the *Em01* object, of type *Employee*, gets permissions at instantiation time and requires permissions for its *buy()* operation, but the « permission-secured » stereotype is missing. Like in the class diagram, the solution is either to add the missing stereotype, or to remove the permissions the object gets at instantiation time, as well as the required permission for the *buy()* method.

There is a second mistake in the sequence diagram: the call to the *buy()* operation performed by *Em01* on *Srv* is protected, which means that the permissions that *Em01* will use need to appear in the {permission} tag labelling the message. Here, the {permission} is *{permission = [emplOperation]}*. However, the *buy()* operation requires two permissions: *emplOperation* and *clientOrder*. This is a simple syntax error that can be easily fixed by adding the missing permission to the « permission » tag: *{permission = [emplOperation, clientOrder]}*. Dynamic checking of the sequence diagram will later make sure that those two permissions can actually be used there. Figure 4.30 is the correct version of the sequence diagram, with the « permission-secured » stereotype and the modified « permission » tag on the second message.

There is also a second option to solve this problem: the user can chose to drop the need for the *clientOrder* permission. In this case, both the class diagram and the sequence diagram need to be updated, as shown on figures 4.31 and 4.32. However, the user should be aware that

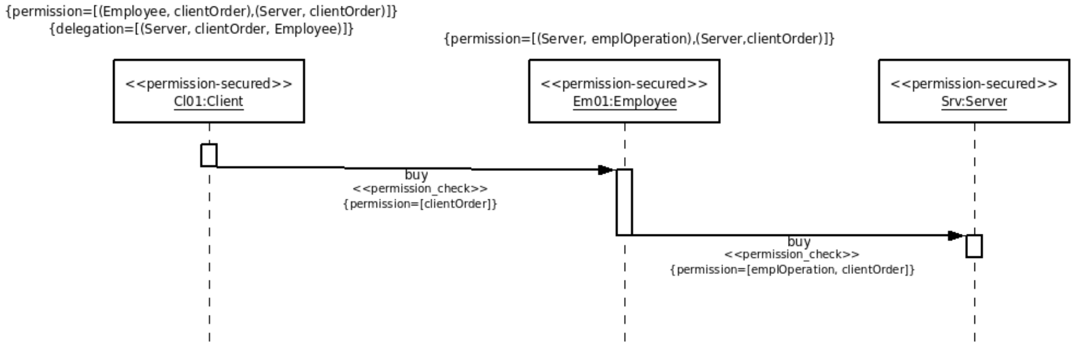

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation),(Server,clientOrder)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
buy
<<permission_check>>
{permission=[emplOperation, clientOrder]}

Figure 4.30: The correct sequence diagram, with the missing « permission-secured » stereotype and the updated « permission » tag

this modification will actually make his model **less** secure, since less permissions are needed to perform the *buy()* operation on the *Server* objects.

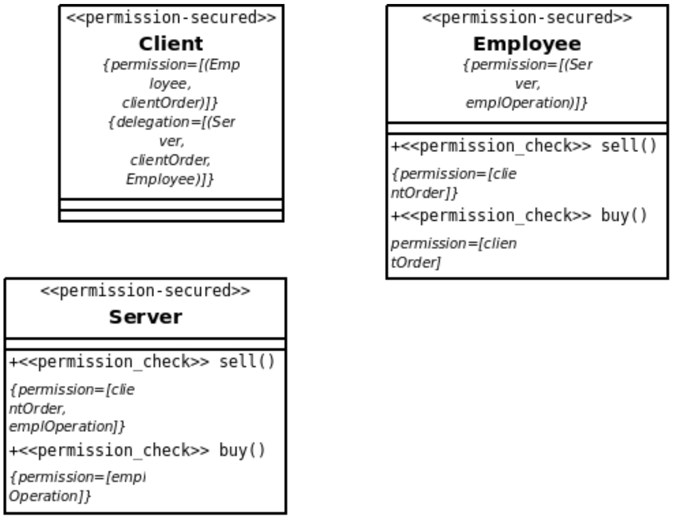

<<permission-secured>>
Client
{permission=[(Employee, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
<<permission-secured>>
Employee
{permission=[(Server, emplOperation)]}
+<<permission_check>> sell()
{permission=[clientOrder]}
+<<permission_check>> buy()
permission=[clientOrder]
<<permission-secured>>
Server
+<<permission_check>> sell()
{permission=[clientOrder, emplOperation]}
+<<permission_check>> buy()
{permission=[emplOperation]}

Figure 4.31: A less secure, but valid, version of the class diagram

Because both diagrams are individually correct doesn't mean there are no inconsistencies when they are put together. Let's go back to figures 4.29 and 4.30. As we can see here, the *Em01* object, whose type is *Employee*, gets two permissions at instantiation time: the first allows *Em01* to use the *emplOperation* permission on the *Server* class, and the second allows it to use the *clientOrder* permission on the *Server*. However, the class diagram only grants the permission to use the *emplOperation* permission on the *Server* class to the *Employee* class at instantiation time. Since there cannot be more permissions granted at instantiation time in the sequence diagram than in the class diagram, the « permission » property is violated by the model. There are two candidate solutions: the first one is to add the missing permission to the *Employee* class in the class diagram, and the second is to remove the problematic permission

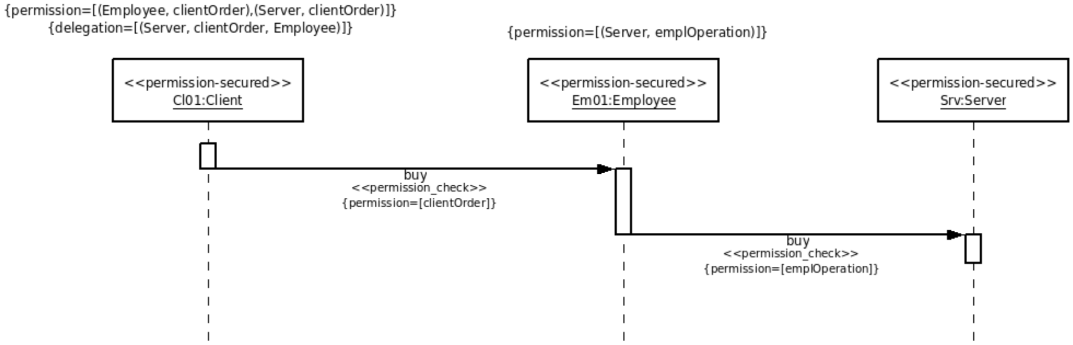

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
buy
<<permission_check>>
{permission=[emplOperation]}

Figure 4.32: A less secure, but valid, version of the sequence diagram

from the sequence diagram. Figure 4.33 shows the updated sequence diagram according to the second option. There is no need to modify the class diagram now, since there are no more inconsistencies.

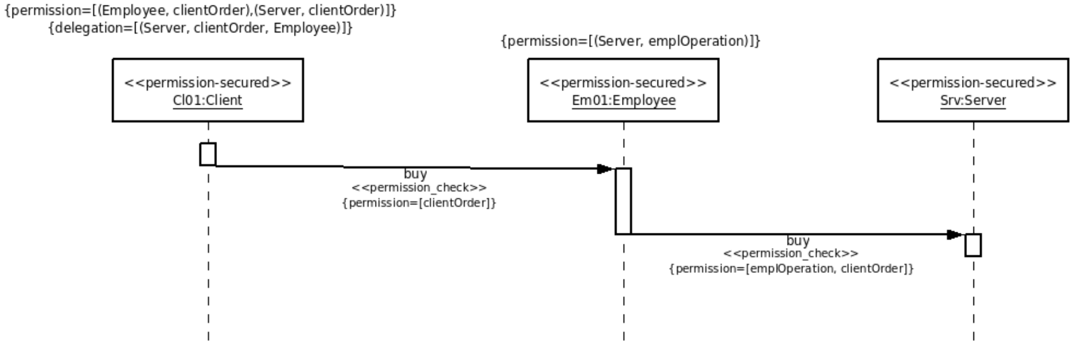

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
buy
<<permission_check>>
{permission=[emplOperation, clientOrder]}

Figure 4.33: The updated sequence diagram, with the second permission removed

Finally, one need to make sure that every sequence of actions can be completed. In order to do this, we have to "play" each sequence of actions, and make sure that each time an object tries to use a permission, it actually holds it, either because it got the permission at instantiation time, or because it has a valid certificate to use it.

Here, this is not the case: the call to the *buy()* method made by the *Em01* object on the *Srv* object uses two permissions: *clientOrder* and *emplOperation*. However, at this time in the sequence diagram, the *Em01* object holds the *EmplOperation* permission (that has been granted at instantiation time), but **not** the *clientOrder* permission. The only way to grant this permission is through the *Cl01* object, that can delegate the missing *clientOrder* permission to the *Em01* object, using the call to the *buy()* method on *Em01*. Figure 4.34 shows the updated version of the sequence diagram. Since the class diagram already allows *Client* classes to delegate the *clientOrder, Server* permission, there is no need to modify the class diagram.

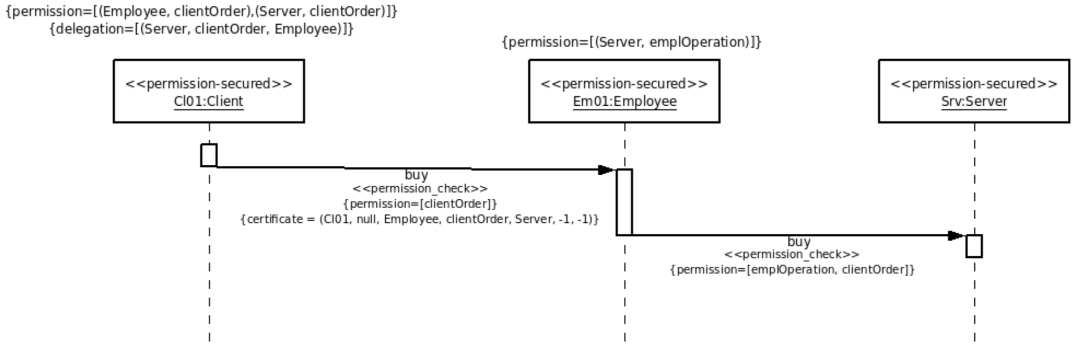

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
{certificate = (Cl01, null, Employee, clientOrder, Server, -1, -1)}
buy
<<permission_check>>
{permission=[emplOperation, clientOrder]}

Figure 4.34: The final version of the sequence diagram

Another solution would have been to drop the need for the missing permission. This possibility has already been discussed, and is pictured on figure 4.32.

4.5.5 Possible side effects

When a model is modified automatically using one of the strategies described above, we concentrate on getting the model to meet only one property. However, most of the time, a complete model contains several UMLsec properties. It would be unfortunate that the changes made to fulfil one of them break another one.

Since different diagrams can strongly depend on each other (for example, a sequence diagram uses objects that are instances of classes described in a class diagram), it is also important to make sure that any modification to a diagram is spread correctly in all the other ones when necessary. This problem, however, is not specific to UMLsec, since it is about the "basic" UML model. However, we try not to modify the UML model when automatically changing a UMLsec model. Instead, we focus on adapting the UMLsec tagged values and stereotypes.

Chapter 5

Producing verified code

Having a set of UML diagrams describing the expected architecture and behaviour is great. Defining security properties in UMLsec and making sure that the UML model meets them is, of course, even better. But because the model meets those properties, doesn't necessarily means that the implementation will also meet them: there's almost no such thing as a bug-free implementation. As soon as a developer, as careful as he might be, writes code that is supposed to match the behaviour described by the UML diagrams, bugs *will* most certainly be introduced. And, of course, some of these almost unavoidable bugs might expose the actual application to a security threat that the developer thought was addressed, since the UMLsec model was correct.

Automatically generating code from a UML model with UMLsec properties would be very helpful. However, the developers will want to modify the code once it has been generated. The main reason is that probably no one will model every single detail of a piece of software using UML diagrams. But also, developers will want to optimise their code, to modify it, to extend it, add features, use libraries, ...

Therefore, a more interesting way of dealing with automatic code generation is not only to generate code that fulfils the UMLsec properties expressed in the UML model, but also to determine in which circumstances modifications of the produced code will or will not impact the validity of the software regarding those security properties. This way, the developers will not only be confident that the produced code fulfils the UMLsec properties, but they will also know which changes will affect those properties, and which changes won't.

It this chapter, we will *not* cover code generation for *every* UMLsec property: instead, we will focus on some of them which are related to access control. We will discuss two different techniques for generating code: the first one, described in section 5.1.1, generates pure-Object-Oriented code, while the second one, described in section 5.1.2, generates Object-Oriented code and Aspect-Oriented code.

5.1 Code generation techniques

5.1.1 Generating Object-Oriented code

The first technique proposed is to generate only Object-Oriented code from UML diagrams with UMLsec properties. The UML diagrams have to be translated into code, and in the same time, the UMLsec properties have to be included in the code. Usually, adding the UMLsec property is just a matter of adding some calls to security APIs.

5.1.2 Generating Aspect-Oriented code

The second technique is to generate Object-Oriented code for the UML diagrams without taking the UMLsec properties into account, and then to generate one aspect for each UMLsec property.

Security is a crosscutting concern, and therefore it makes sense to write aspects in order to enforce UMLsec properties in the generated code. UMLsec properties can also be defined independently, which makes using Aspect-Oriented code generation even easier: we can generate one aspect for every UMLsec property we want to enforce, and activate or deactivate one when needed, without having an influence on the other ones and on the fulfilment of the other properties.

5.2 Generating code from a « permission » property

The « permission » property is defined on two different types of diagrams: class diagrams and sequence diagrams, as described in section 4.1.3. Therefore, the code generation for this property will be twofold: first, we generate the code from the class diagram, and then code from the sequence diagram is added.

5.2.1 Authorization API

The « permission » property in UMLsec allows one to restrict access to operations and attributes, but it does not model how the authorization is performed. Therefore, the code generator can not generate code for the authorization mechanism itself. Instead, we will assume that an authorization API is available, and the generated code will call the API when necessary.

In this section, we will assume that the following methods are available through the API. Those are written in pseudo-code.

- `public addPermission(source,permission, target)`: grant to the *source* object permission *permission* on object *target*.

- `public checkPermission(source, permission, target)`: check that object *source* has the permission *permission* on object *target* at the moment the check is performed.

- `public delegate(source, permission, target, recipient, validity_nbr, validity_time)`: the *source* object delegates permission *permission* on the *target* object to the *recipient* object. The permission can be used *validity_ nbr* times, and its validity period is *validity_ time*.

- `public addRestriction(element, permission)`: permission *permission* is needed to access the method or attribute *element*.

- `public addNoPermNeeded(element, source)` class *source* doesn't need any permission to access the method or element *element*.

- `public addDelegation(source, permission, target)`: allows the *object* to delegate permission *permission* on object *target* to any other object.

The authorization code will also have to deal with certificates lifespan:

- Some certificates can't be used more than a fixed number of times. The authorization code will make sure that the usage counter is decremented at every usage, and that the permissions granted by the certificate will be removed when the certificate becomes invalid.

- Other certificates do no have this limit. Therefore, their counter should never be decremented, and the permissions granted by the certificate should never be removed

- Some certificates have a validity limited in time. They should be removed, as well as all the permissions that come with them, as soon as they are no longer valid.

- Other certificates have an unlimited validity period, and should therefore never be removed.

5.2.2 Object-Oriented solution

We first describe the plain Object-Oriented solution. We will have to generate the code in two steps: first, we will generate code from the class diagram, and then code from the sequence diagram. The order is important: we first generate the classes with their attributes and method signatures from the class diagram, and then we can fill the methods with the sequences of operations described in the sequence diagram.

Class diagram

Generating Object-Oriented code from a Class diagram, without taking an UMLsec property into account, is quite easy: a class is a class, an attribute is an attribute, and an operation is a method. Things, however, get slightly more complicated when we want to introduce access control on the methods.

In a class diagram with the « permission » property, the classes that have operations or attributes whose access needs to be restricted are labelled with the « permission-secured » stereotype. Therefore, any class that is *not* stereotyped with « permission-secured » can have its code generated like a "plain" UML class: a class in UML becomes a class in the target language, an attribute becomes an attribute, and an operation becomes a method. All the other classes require a little bit more work when the code is generated.

When code for a class stereotyped with « permission » is generated, permissions and delegation capabilities have to be granted to the objects at instantiation time. Thus, calls to the API need to be added as the first lines of the constructor's body. For every permission in the {permission} tag, the following call will be added: `addPermission(source, permission, target)`. And for every delegation in the {delegation} tag, the following call will be added: `addDelegation(source, permission, target)`.

Each time code for an operation protected with the « permission-check » stereotype is generated, there is an API call to add as the first thing inside the method body: `addRestriction(element, permission)`, where *element* is the method, and *permission* is the permission required to call the operation. If multiple permissions are needed to call the operation, then there will be a similar API call for each of them.

If the operation contains a {no_permission_needed} tag, another API call has to be added: `addNoPermNeeded(element, source)`, with *element* being the operation, and *source* the object that doesn't need permissions to call the operation.

For an attribute, it gets more complicated: the attribute should be made `private`, and a method should be created to access it. Inside the method body, the first lines will be calls to the API, similar to those described above.

Sequence diagram

Once the class diagram has been processed, it is time to process the sequence diagram. The sequence diagram tells us which method calls will be performed during a method execution. However, it does not model the entire method body, so the code will have to be completed by hand, or generated using another diagram, like an activity diagram. Still, when playing the execution sequence of the sequence diagram, every call to an operation can be translated as a method call inside the current method. Return values are return, conditional blocks translate into if/then/else structures, and loop blocks translate into while structures.

Every time a method call is stereotyped with « certification », a certificate has to be transmitted to the callee. A call to the API should be added right before the method call: `delegate(source, permission, target, recipient, validity_nbr, validity_time)`, that will delegate the permission to the callee.

Example

We will use a small example to illustrate the code generation. Let's go back to the example we used in section 4.5.4. The class diagram (figure 5.1) and the sequence diagram (figure 5.2) describe an interaction between a *Client*, an *Employee* and a *Server*. The *Client* calls the operation *buy()* on the *Employee*, which then calls the operation *buy()* on the *Server*.

For this example, we will use the Java programming language as an example of an Object-Oriented language. We assume that the authentication API we use matches to one described above, and can be accessed through a static object called `AuthAPI`.

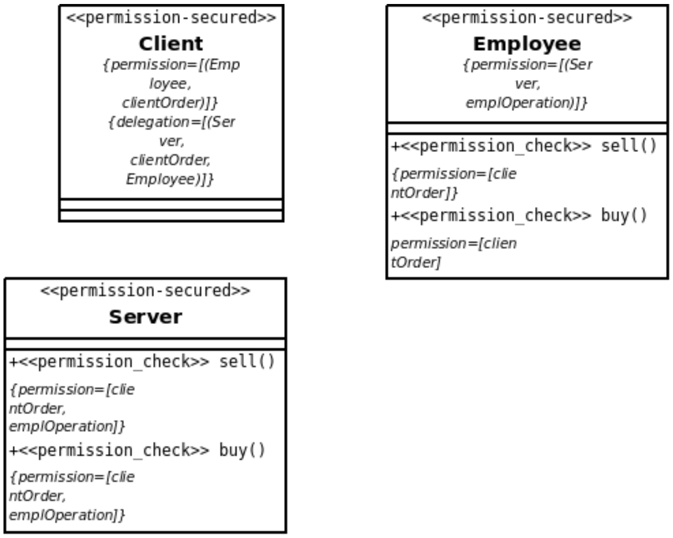

<<permission-secured>>
Client
{permission=[(Employee, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
<<permission-secured>>
Employee
{permission=[(Server, emplOperation)]}
+<<permission_check>> sell()
{permission=[clientOrder]}
+<<permission_check>> buy()
permission=[clientOrder]
<<permission-secured>>
Server
+<<permission_check>> sell()
{permission=[clientOrder, emplOperation]}
+<<permission_check>> buy()
{permission=[clientOrder, emplOperation]}

Figure 5.1: The class diagram describing the system

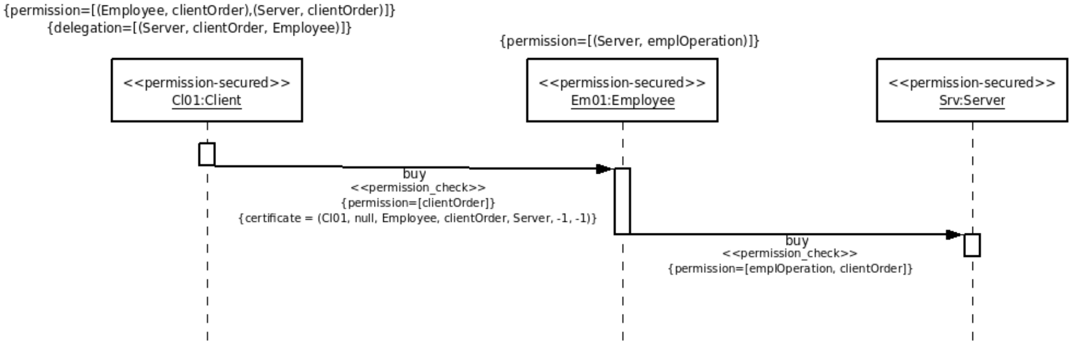

{permission=[(Employee, clientOrder),(Server, clientOrder)]}
{delegation=[(Server, clientOrder, Employee)]}
{permission=[(Server, emplOperation)]}
<<permission-secured>>
Cl01:Client
<<permission-secured>>
Em01:Employee
<<permission-secured>>
Srv:Server
buy
<<permission_check>>
{permission=[clientOrder]}
{certificate = (Cl01, null, Employee, clientOrder, Server, -1, -1)}
buy
<<permission_check>>
{permission=[emplOperation, clientOrder]}

Figure 5.2: The sequence diagram

Let's start with the class diagram. Figures 5.3, 5.4 and 5.5 contain the generated code for each of the three classes.

```
public Client{
    public Client() {
        AuthAPI.addPermission(this,Employee,ClientOrder);
        AuthAPI.addDelegation(this,clientOrder,Server);
    }
}
```

Figure 5.3: client.java

The Client class, on figure 5.3, doesn't contain any method. It gets one permission at instantiation time, and the ability to delegate it.

```
public Employee {
    public Employee() {
        AuthAPI.addPermission(this,emplOperation,Server);
        AuthAPI.addRestriction(this.sell, clientOrder);
        AuthAPI.addRestriction(this.buy, clientOrder);
    }

    public sell() {
        AuthAPI.checkPermission(caller, clientOrder, this.sell);
    }

    public buy() {
        AuthAPI.addRestriction(caller, clientOrder, this.buy);
    }
}
```

Figure 5.4: employee.java

The Employee class, on figure 5.4, gets one permission at instantiation time, and has two methods, each requiring one permission to be called.

And finally, the Server class, on figure 5.5, doesn't get any permission at instantiation time, but has two methods, each requiring two permissions to be called.

Now, we process the sequence diagram. Figures 5.6 and 5.7 show the updated code.

An *Employee* object, named *em01*, has been added to the *Client* class, so the the *buy()* method could be called. Before calling the *buy* method, the delegation is performed through the *delegate(...)* call to the API, so that the permission can be delegated to *em01*, as seen on figure 5.6.

```
public Server {
    public Server() {
        AuthAPI.addRestriction(this.sell, clientOrder);
        AuthAPI.addRestriction(this.sell, emplOperation);
        AuthAPI.addRestriction(this.buy, clientOrder);
        AuthAPI.addRestriction(this.buy, emplOperation);
    }

    public sell() {
        AuthAPI.checkPermission(caller, clientOrder, this.sell);
        AuthAPI.checkPermission(caller, emplOperation, this.sell);
    }

    public buy() {
        AuthAPI.checkPermission(caller, clientOrder, this.buy);
        AuthAPI.checkPermission(caller, emplOperation, this.buy);
    }
}
```

Figure 5.5: server.java

```
public Client{
    private Employee em01;

    public Client() {
        AuthAPI.addPermission(this,Employee,ClientOrder);
        AuthAPI.addDelegation(this,clientOrder,Server);

        Auth.delegate(this, permission, target, recipient, validity_nbr,
            validity_time);
        em01.buy();
    }
}
```

Figure 5.6: updated client.java

```
public Employee {

    Server srv;

    public Employee() {
        AuthAPI.addPermission(this,emplOperation,Server);
        AuthAPI.addRestriction(this.sell, clientOrder);
        AuthAPI.addRestriction(this.buy, clientOrder);
    }

    public sell() {
        AuthAPI.checkPermission(caller, clientOrder, this.sell);
    }

    public buy() {
        AuthAPI.addRestriction(caller, clientOrder, this.buy);

        srv.buy();
    }
}
```

Figure 5.7: updated employee.java

As we can see on the sequence diagram, during the execution of the `buy()` method on *em01*, a call to the `buy()` method of the *srv* object is performed. Therefore, a reference to the *srv* object has been added to the *Employee* class, and the call to the `buy()` method on *srv* has been added to the body of the `buy()` method, as illustrated on figure 5.7.

Finally, The *Server* class hasn't been updated, since it wasn't necessary.

5.2.3 Aspect-Oriented solution

In the Aspect-Oriented solution, we will first generate code from both the class diagram and the sequence diagram without taking the UMLsec properties into account, and then we will create an aspect that will add the « permission » restrictions to the code.

Class diagram

Once the code has been generated from the class diagram, it is time to take care of the UMLsec stereotypes. In the class diagram, every class that is stereotyped with « permission-secured » has operations and/or attributes that need to be protected. Therefore, every call to a protected operation (which is stereotyped with « permission-check ») and every access to a protected attribute (which is also stereotyped with « permission-check ») needs to be captured by a pointcut that will make sure that the caller has the necessary permissions to access the operation or the attribute. Otherwise, it will throw an exception.

Permissions also need to be granted to objects at instantiation time. All the classes stereotyped with the « permission-secured » stereotype and labelled with the {permission} tag get one or more permissions at instantiation time. For each of them, we add a pointcut to any of

the class' constructors, as well as an advice that performs the following calls to the Permission Management Module:
`addPermission(this, permission, target)` Where *permission* and *target* describe which permission is granted to be used on which object. Of course, if there are several permissions granted, there will also be several calls to the Permission Management Module (one for every permission granted).

Within the same advice, we add another call to the Permission Management API if and only if the class is also labelled with the {delegation} tag:
`addDelegation(this, permission, target)` Where *permission* and *target* describe which permission can be delegated. Of course, there will be one API call for each delegation possibility in the {delegation} tag.

Sequence diagram

Once the code from the sequence diagram has been generated, as well as the aspect from the class diagram, it is time to use the stereotypes in the sequence diagram to enforce the « permission » property. There is no need to create a new aspect, since we can use the one that has been created while processing the class diagram. However, creating another one would also be possible.

The delegation of a certificate has also to be done using a pointcut: we capture the method call corresponding to the message in the sequence diagram where the certificate is delegated, and, using a `before` advice, add a call to the permission manager, that will make sure that:

- the caller has the right to delegate the permission
- the certificate is well formed
- the callee's permissions are updated

Finally, the validity of the certificates will have to be handled by the permission manager, which will work, since a call to the permission manager is already performed each time an object tries to access a protected method or attribute.

Example

Let's reuse the example we used for the generation of Object-Oriented code, and see what happens when we generate Aspect-Oriented code. We will also use Java as our Object-Oriented language, and we will add AspectJ as our Aspect-Oriented extension.

We start with the generation of the code from the class diagram, that we can see on figures 5.8, 5.9 and 5.10.

Now, we need to generate the aspect from the class diagram. It is shown on figure 5.11

```java
public Client {
    public Client() {
    }
}
```

Figure 5.8: client.java

```java
public Employee {
    public Employee() {
    }

    public buy() {
    }

    public sell() {
    }
}
```

Figure 5.9: employee.java

```java
public Server {
    public Server() {
    }

    public sell() {
    }

    public buy() {
    }
}
```

Figure 5.10: server.java

```
public aspect PermissionAspect {

    pointcut employeeProtection()
        : call (* Employee.sell()) || call (* Employee.buy());

    pointcut serverProtection()
        : call(* Server.sell()) || call (* Server.buy());

    pointcut clientConstr()
        : call(* Client.new(..));

    pointcut employeeConstr()
        : call(* Employee.new(..));

    before() : employeeProtection() {
        AuthAPI.checkPermission(this, clientOrder, Employee);
    }

    before() : serverProtection() {
        AuthAPI.checkPermission(this, clientOrder, Server);
        AuthAPI.checkPermission(this, emplOperation, Server);
    }

    before() : clientConstr() {
        AuthAPI.addPermission(this, Employee, clientOrder);
        AuthAPI.addDelegation(this, clientOrder, Server);
    }

    before() : employeeConstr() {
        AuthAPI.addPermission(this, emplOperation, Server);
    }
}
```

Figure 5.11: Aspect generated from the class diagram only

The first two pointcut capture calls to the « permission-secured » methods, and the last two capture calls to any of the constructors of classes *Client* or *Employee*. Since the *Server* class doesn't get permissions at instantiation time, there is no need to capture its constructor calls.

The first advice adds a call to the `checkPermission` method on the Permission Management API, in order to make sure that the caller has the permission to call the protected methods on the *Employee* objects. The second advice is similar, but for the protected methods of the *Server* objects.

The third one grants permission to every object of type *Client* when it is created, as well as the possibility to delegate a permission, through the corresponding API calls. The last one is similar, but for the objects of type *Employee*, and without the delegation capability.

Now, we update the Java code using the information provided by the sequence diagram. The resulting code is on figures 5.12 and 5.13. The *Server* class has not been updated, since the sequence diagram doesn't provide any extra information about it.

```
public Client {

    private Employee em01;

    public Client() {
        em01.buy();
    }
}
```

Figure 5.12: updated client.java

```
public Employee {

    private Server srv;

    public Employee() {
    }

    public buy() {
        srv.buy();
    }

    public sell() {
    }
}
```

Figure 5.13: updated employee.java

And finally, we update the aspect with the information found in the sequence diagram, as we can see on figure 5.14.

```
public aspect PermissionAspect {

    pointcut employeeProtection()
        : call (* Employee.sell()) || call (* Employee.buy());

    pointcut serverProtection()
        : call(* Server.sell()) || call (* Server.buy());

    pointcut clientConstr()
        : call(* Client.new(..));

    pointcut employeeConstr()
        : call(* Employee.new(..));

    pointcut certFromClientToEmployee()
        : call(* Employee.buy());

    before() : employeeProtection() {
        AuthAPI.checkPermission(this, clientOrder, Employee);
    }

    before() : serverProtection() {
        AuthAPI.checkPermission(this, clientOrder, Server);
        AuthAPI.checkPermission(this, emplOperation, Server);
    }

    before() : clientConstr() {
        AuthAPI.addPermission(this, Employee, clientOrder);
        AuthAPI.addDelegation(this, clientOrder, Server);
    }

    before() : employeeConstr() {
        AuthAPI.addPermission(this, emplOperation, Server);
    }

    before() : certFromClientToEmployee() {
        AuthAPI.delegate(this, clientOrder, Server, Employee, -1, -1);
    }
}
```

Figure 5.14: Aspect updated with the information found in the sequence diagram

.

Another pointcut has been added, that will capture every call from the *Cl01 Client* to the `buy()` method on the *EM01 Employee*. A new advice adds a call to the `delegate(..)` method of the Permission Management API, that will perform the permission delegation.

5.3 Generating code from a « rbac » stereotype

The second property we will focus on is the « rbac » property, that allows one to define Role-Based Access Control rules on an activity diagram. Like for the code generation from the « permission » stereotype, we will use an external module to perform authentication and authorization. Here, we will use the JAAS framework, that is described in details below. Of course, any other authentication and authorization framework could be used with little change.

5.3.1 The JAAS framework

JAAS, Java Authentication and Authorization Service, is part of the Java Security Framework since Java 1.4. It is very extensible, since each of its modules' default implementation can be replaced by custom code, in order to deal with any possible situation. It is divided in two parts: authentication and authorization. The following description is inspired by [Mic01].

Authentication

The authentication process using JAAS has been implemented in a completely pluggable way [Mic01], which means that every part of the authentication process can be replaced with custom code. That makes JAAS independent from any underlying authentication technology [Mic01].

We will cover here the basic principles of authentication using JAAS: we will describe the concepts it introduces, and see how they work together to provide a complete authentication mechanism.

The first thing that needs to be done for using JAAS-based authentication is creating a `LoginContext` (`javax.security.auth.login.LoginContext`). The `LoginContext` takes two arguments: an entry name, and a `CallBackHandler`.

The entry name is the name of an entry in the JAAS login configuration file. This file contains entry names describing which `LoginModule` should be used to perform authentication. Each entry contains the name of a class implementing the `LoginModule` interface. That is where the authorization mechanisms are implemented.

The second argument, the `CallBackHandler`, is used to provide the `LoginModule` with a way to interact with the user, eg. to ask for a username or a password.

One the `LoginContext` has be instantiated, then the *login()* method has to be called. It will perform the authentication, and create a new `Subject` (`javax.security.auth.Subject`) object. The `Subject` represents the user, and is populated by the `login()` method with a *Principal* if the authentication is successful. The *Principal* also describes the user, but it is different from the `Subject`: while the `Subject` describes the source of a request, there might be several *Principal*s for a single `Subject`, each *Principal* being an identity of the user that will be used to determine whether he has access to protected resources [Mic01].

Authorization

Once the user is authenticated, it is possible to make sure that he has the required permissions to access a protected method. This is the authorization part of the JAAS framework.

Calls to protected methods can not be performed directly. Instead, each protected method should be encapsulated in a *action object*, who must implement either `PrivilegedAction` (`java.security.PrivilegedAction`) or `PrivilegedExceptionAction` (`java.security.PrivilegedExceptionAction`). There must be a `run()` method that implements the protect method.

To execute the *action object*, the static method `doAsPrivileged(Subject, PrivilegedAction, AccessControlContext)` of the `Subject` class has to be called. This method will try to perform the protected operation *on behalf* of the `Subject` in its first argument.

Every method that needs to have its access protected need to call the static `checkPermission()` method on the `AccessController` class (`java.security.AccessController`), that will throw an `Exception` if the `Subject`'s permissions are not sufficient.

Then, the security policy must be configured, in order to define which access rights are given to which users, according to their *Principal*s. This configuration is done in a `Policy` file.

The `Policy` file *grants* permissions to *Principal*s. The syntax for a *grant* statement is described in figure 5.15.

```
grant <signer(s) field>, <codeBase URL>
      <Principal field(s)> {
                 permission perm_class_name "target_name", "action";
                 ....
                 permission perm_class_name "target_name", "action";
      };
```

Figure 5.15: grant statement syntax (from [Mic01])

The `signer field`, the `codeBase URL` and the `Principal field` are all optional. Here, we will only focus on the `Principal field`, sine we won't use the other ones.

The `Principal field`'s syntax is simple:

Principal *Principal_class* *“principal_name”*.

Principal_class is the fully qualified name of the *Principal* class, that implements the java.security.Principal interface, and the second field is the *Principal* name.

If a *grant* statement has several *principal* fields, then **all** of them are required for a Subject to be granted access to the protected resource.

An example of a JAAS program using authentication and authorization will be given in the next sections, both using Object-Oriented programming and Aspect-Oriented programming.

5.3.2 Authentication

Similarly to the « permission » property, the « rbac » property does **not** model how the authentication process is handled. When generating code, we will then have to assume that the authentication has already been done. It will be the developer’s responsibility to create an authentication module, or to use an existing one.

5.3.3 Authorization: Object-Oriented solution

First, we need to create a Permission for every operation that needs to be protected. Since we only need the name of the Permission, we can use the BasicPermission class, which is a simple subclass of Permission that does everything we need. In order to make things easier to read, we can even subclass the BasicPermission class, for every class that has protected methods. This will also avoid any name conflict problem. Figure 5.16 shows a sample BasicPermission subclass.

```java
import java.security.*;

public final class MyNewClassPermission extends BasicPermission {
    public MyNewClassPermission(String name) {
        super(name);
    }

    public MyNewClassPermission(String name, String actions) {
        super(name, actions);
    }
}
```

Figure 5.16: A sample subclass of the BasicPermission class

Once each BasicPermission subclass has been created, we can start generating the code. A single activity diagram, like required for the « rbac » stereotype definition, is not sufficient to generate the code. We need at least a class diagram, but it doesn’t have to come with any UMLsec stereotype or tagged value.

The generation of the classes from the class diagram has already been described in section 5.2. Once the classes have been generated, let's take a look at the activity diagram. Each protected action has to have a corresponding operation that will allow us to link the action to an operation in the class diagram. For every protected action, we add to the method implementing the corresponding operation a call to the `AccessController` class at the very beginning of the method body, as we can see on figure 5.17.

```
AccessController.checkPermission(
                    new MyNewClassPermission("methodName"));
```

Figure 5.17: Code snippet to add at the beginning of every protected method

Once the access restricted methods have been protected, it is time to take care of the calls to the protected methods. Provided that the authentication process has been made, we need to:

- get the `Subject` from the `LoginContext`
- Create a `PrivilegedAction` object to run the protected method
- Call the static `doAsPrivileged(...)` method on the `Subject` class, with the `Subject` and the `PrivilegedAction` as parameters.

Therefore, every call to a protected method will be replaced by those steps. Figure 5.18 shows a simple call to a protected method. The code snippet uses the `lc` attribute, which is of type `LoginContext`. It tries to run the protected `credit(int amount)` method on the `account` object.

```
Subject mySubject = lc.getSubject();

Subject.doAsPrivileged(authenticatedSubject,
                       new PrivilegedAction() {
                           public Object run() {
                               account.credit(1000);
                               return null;
                           }}, null);
```

Figure 5.18: Code snippet that will call a protected method

Example

We illustrate this solution with a simple example that produces Java code, inspired by an example from [Lad03]. Figure 5.19 is a simple activity diagram with « rbac » properties included.

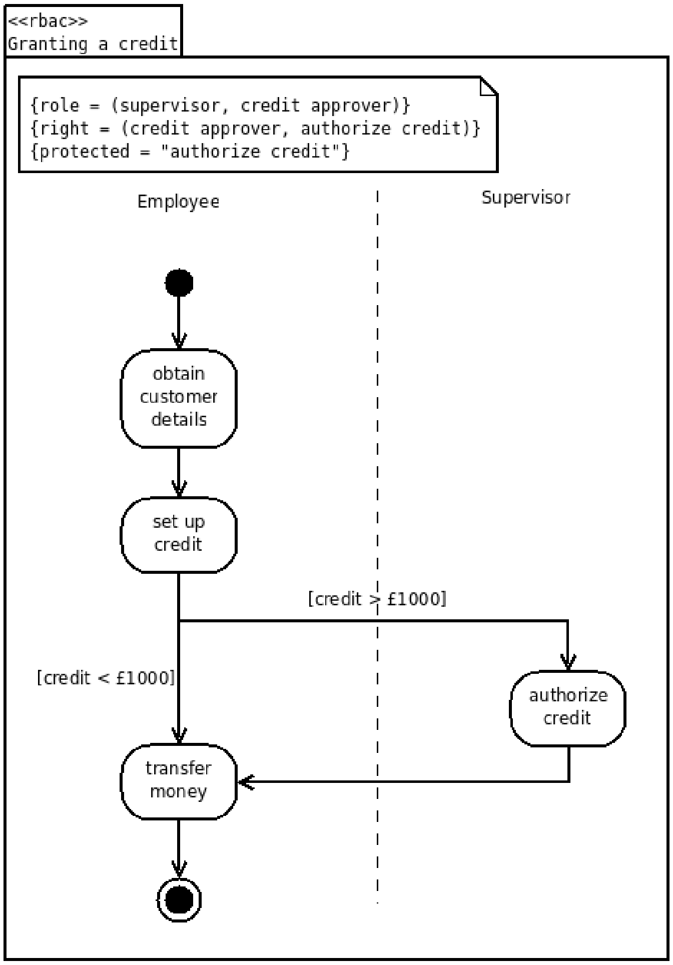


Figure 5.19: A simple activity diagram with « rbac » properties

The *authorize credit* activity is protected. The role ***credit approver*** has the right to perform it, and the user *Supervisor* has that role.

But we also need a class diagram to generate the code. We can find it on figure 5.20.

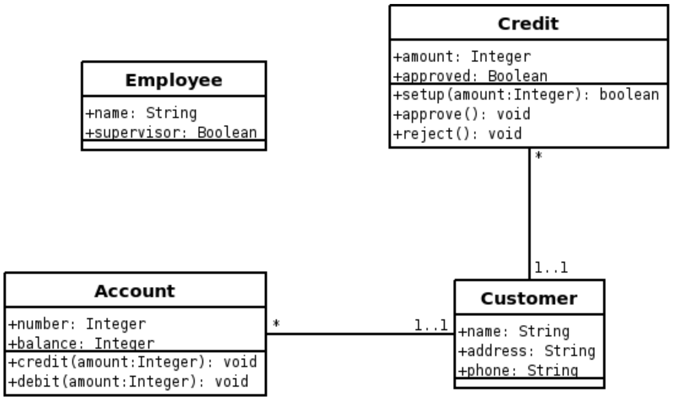


Figure 5.20: The class diagram corresponding to the activity diagram on figure 5.19

The first thing we do is processing the class diagram in order to produce Java classes. The result of this operation can be seen on figures 5.21, 5.22, 5.23 and 5.24.

```
public class Employee {

    private String name;
    private Boolean supervisor;

    public Employee() {
    }
}
```

Figure 5.21: Employee.java

Once the classes have been generated, it is now time to add access control information that we will extract from the activity diagram. The activity diagram shows that the `authorize credit` action state is protected. Since it contains in its actions list the `approve()` operation of the *Credit* class, we deduce that the `approve()` operation needs to be protected.

Figure 5.25 shows the updated *Credit* class, with the necessary JAAS access control calls. We also had to create a new class that extends `BasicPermission`, as we can see on figure 5.26.

Now that all the methods requiring access control have been protected, it is time to make sure that all the calls to those methods include the necessary code for being authorized to

```java
public class Account {
    private Integer number;
    private Integer balance;

    public Account() {
    }

    public void credit(Integer amount) {
    }

    public void debit(Integer amount) {
    }
}
```

Figure 5.22: Account.java

```java
public class Credit {
    private Integer amount;
    private Boolean approved;

    public Credit() {
    }

    public boolean setup(Integer amount) {
    }

    public void approve() {
    }

    public void reject() {
    }
}
```

Figure 5.23: Credit.java

```java
public class Customer {
    private String name;
    private String address;
    private String phone;

    public Customer() {
    }
}
```

Figure 5.24: Customer.java

```
import permissions.CreditPermission;

public class Credit {
    private Integer amount;
    private Boolean approved;

    public Credit() {
    }

    public boolean setup(Integer amount) {
    }

    public void approve() {
        AccessController.checkPermission(new CreditPermission("approve");
    }

    public void reject() {
    }
}
```

Figure 5.25: Credit.java - now with the `approve()` method protected

```
import java.security.*;

public final class CreditPermission extends BasicPermission {
    public CreditPermission(String name) {
        super(name);
    }

    public CreditPermission(String name, String actions) {
        super(name, actions);
    }
}
```

Figure 5.26: CreditPermission.java - the Permission that will handle permissions for the *Credit* class

perform the code. Unfortunately, the current model does not provide any sequence diagram that would allow us to generate method calls.

Let's add a sequence diagram to our model so that we can illustrate the addition of code for calls to protected methods. Figure 5.27 shows a simple sequence diagram that describes a sequence of actions performed by an `Employee` on a *Credit*.

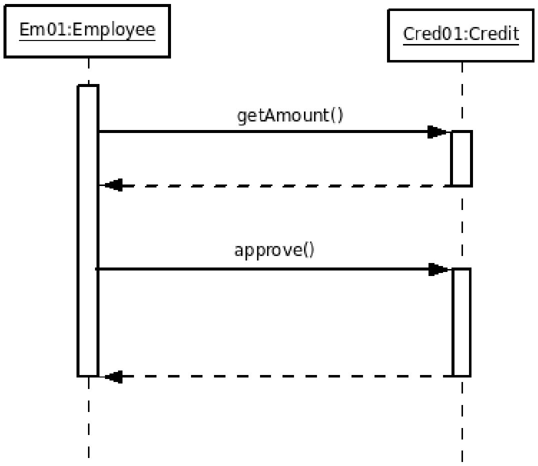


Figure 5.27: A sequence diagram that includes a call to a protected operation

When we generate the code from that diagram, it is the *Employee* class that is updated, as we can see on figure 5.28.

This, of course, will have to be repeated for *every* call to a protected method. Now, if for some reason, once we already have a large codebase, someone wants to add access restrictions to a method or wants to remove access restrictions from another one, changes will have to be made through the whole code, and it is very likely that some errors will arise.

5.3.4 Authorization: Aspect-Oriented solution

As we can see in the previous section, the Object-Oriented solution produces a lot of additional code: for **each** call to a protected method, we need to create a new `PrivilegedAction` object, call the `doAsPrivileged` method with the `Subject`, and also handle the exceptions that could be thrown. Also, every method that needs to be protected also requires some code to be added at the beginning of its body. It is always the same code that is reused everywhere, leading to typical tangling and scattering problems. Using Aspect-Oriented programming will allow us to regroup the access control related code in one place only.

The code we need to add can be divided in two parts: the code that protects a method, and the one that allows to call a protected method. Therefore, we will have to create two advices in our aspect, each one adding code to a different pointcut.

```
import java.security.*;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;

public class Employee {

    private String name;
    private Boolean supervisor;
    private Credit Cred01;
    LoginContext lc;

    Subject authenticatedSubject = lc.getSubject();

    public Employee() {
        Cred01.getAmount();
        Subject.doAsPrivileged(authenticatedSubject,
                              new PrivilegedAction() {
                                  public Object run() {
                                      Cred01.approve();
                                      return null;
                                  }}, null);
    }
}
```

Figure 5.28: Update Employee.java - now with JAAS authorization support

We start with the protection of methods: each time a method needs to be protected, we want to add a call to
`AccessController.checkPermission(Permission perm)`
for each permission needed to access the method. This means that we will have to capture those method's execution, and then write a `before()` advice that will add the calls to the `AccessController`.

Then, each call to any of those protected method will have to be captured in another pointcut. The corresponding advice will be an `around()` advice, that will call the necessary methods, as shown on figure 5.29, where `authenticatedSubject` is the `Subject` on behalf of which the method will be called, and `authOperation()` the pointcut that captures the call to the protected method.

Defining an abstract aspect

We can make this solution even more elegant if we use an Aspect-Oriented Programming language that supports abstract aspects. Actually, we can reuse the same advices every time, with abstract pointcuts that we will re-implement each time. Figure 5.30 shows such an abstract aspect written in AspectJ, following the solution proposed by [Lad03].

As we can see, the two advices are already implemented, and can be reused. Only the pointcuts need to be generated, as well as the `getPermission` method. We only have to

```
Object around()
    : authOperation() && !cflowbelow(authOperation()) {
        try {
            return Subject.doAsPrivileged(authenticatedSubject,
                                                new PrivilegedExceptionAction() {
                                                    public Object run() throws
                                                        Exception {
                                                        return proceed();
                                                    }}, null);
        } catch (PrivilegedActionException ex) {
            throw new AuthorisationException(ex.getException());
        }
}
```

Figure 5.29: A sample `around()` advice that performs a call to a protected method

generate one aspect for each class, and most of the code will be inherited from the abstract aspect.

Example

Finally, we illustrate the Aspect-Oriented Programming approach using an example. Let us reuse the example we used for the Object-Oriented Programming approach: it will make it easier to compare the two proposed solutions.

We have figure 5.31, an activity diagram with UMLsec properties. Figure 5.32 is the corresponding class diagram, and figure 5.33 is the dummy Sequence diagram we introduced later in the Object-Oriented Programming approach example.

When processing those diagrams without taking the UMLsec properties into account, we get four classes, as we can see on figures 5.34, 5.35, 5.36 and 5.37.

It is now time to process the UMLsec properties in order to create the aspect that will enforce the authorization properties. We will reuse the abstract aspect defined above on figure 5.30.

We then create the pointcuts for this particular example in an aspect that extends the abstract aspect, as we can see on figure 5.38.

We just had to implement the `authOperations()` pointcut and the `getPermission` method to add JAAS authorization support to the code. We also had to create a new class extending *Permission*, like in the Object-Oriented Programming example. This class is shown on figure 5.39.

```
import org.aspectj.long.JoinPoint;

import java.security.*;
import javax.security.auth.Subject;
import javax.security.auth.login.*;

import com.sun.security.auth.callback.TextCallbackHandler;

public abstract aspect AbstractAuthAspect {
    private Subject _authenticatedSubject;

    public abstract pointcut authOperations();

    public abstract Permission getPermission(JoinPoint.StaticPart
        joinPointStaticPart);

    Object around()
        : authOperations() && !cflowbelow(authOperations()) {
        try {
            return Subject.doAsPrivileged(_authenticatedSubject,
                                          new PrivilegedExceptionAction() {
                                              public Object run() throws
                                                  Exception {
                                                  return proceed();
                                              }}, null);
        } catch (PrivilegedActionException ex) {
            throw new AuthorizationException(ex.getException());
        }
    }

    before() : authOperations() {
        AccessController.checkPermission(getPermission(thisJoinPointStaticPart)
            );
    }
}
```

Figure 5.30: AbstractAuthAspect.aj - Abstract Authentication Aspect [Lad03]

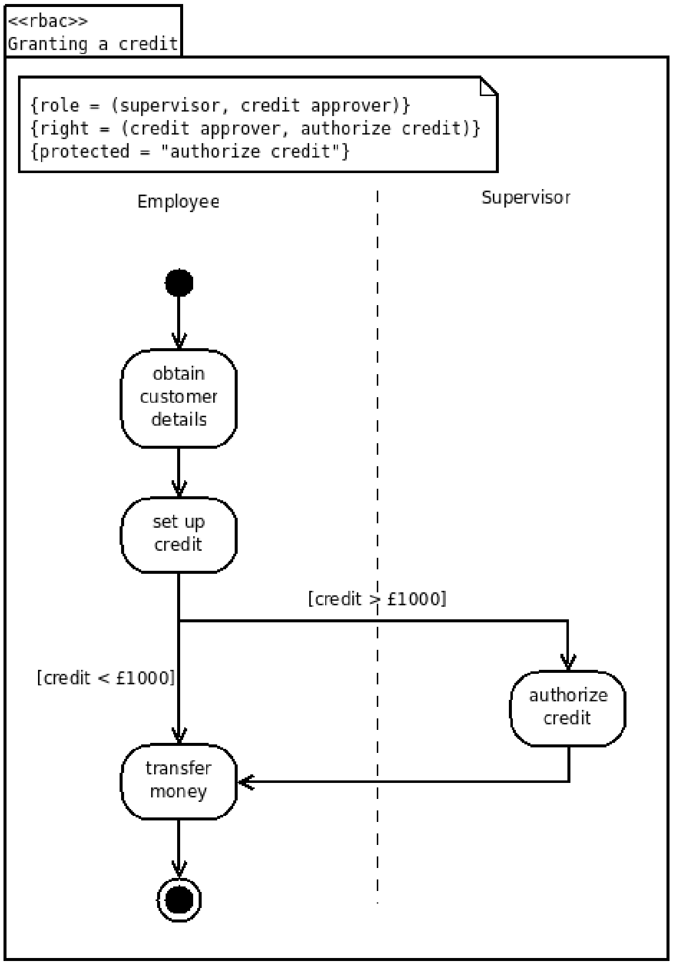


Figure 5.31: A simple activity diagram with the « rbac » UMLsec property

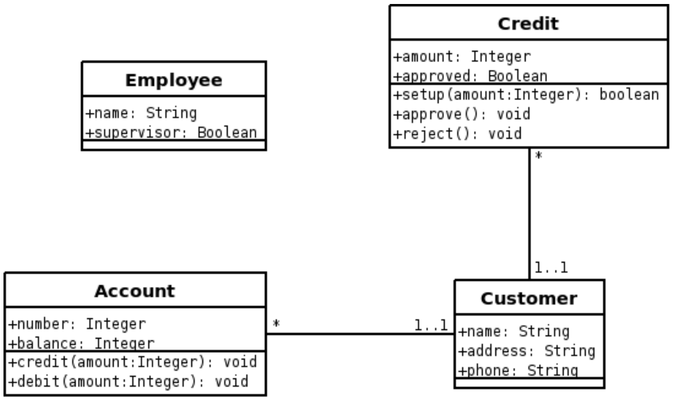


Figure 5.32: The Class diagram corresponding to the activity diagram on figure 5.31

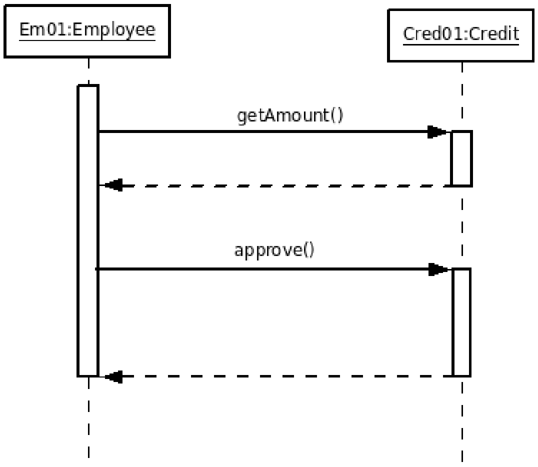


Figure 5.33: A sequence diagram that performs a call to a protected operation

```java
opublic class Employee {

    private String name;
    private Boolean supervisor;
    private Credit Cred01;

    public Employee() {
        Cred01.getAmount();
        Cred01.approve();
    }
}
```

Figure 5.34: Employee.java - generated without UMLsec properties

```java
public class Account {
    private Integer number;
    private Integer balance;

    public Account() {
    }

    public void credit(Integer amount) {
    }

    public void debit(Integer amount) {
    }
}
```

Figure 5.35: Account.java - generated without UMLsec properties

```java
public class Credit {
    private Integer amount;
    private Boolean approved;

    public Credit() {
    }

    public boolean setup(Integer amount) {
    }

    public void approve() {
    }

    public void reject() {
    }
}
```

Figure 5.36: Credit.java - generated without UMLsec properties

```java
public class Customer {
    private String name;
    private String address;
    private String phone;

    public Customer() {
    }
}
```

Figure 5.37: Customer.java - generated without UMLsec properties

```java
import org.aspectj.lang.JoinPoint;

import java.security.Permission;

import auth.AbstractAuthAspect;

public aspect CreditAuthAspect extends AbstractAuthAspect {
    public pointcut authOperations()
        : execution(public void Credit.approve());

    public Permission getPermission(
                                    JoinPoint.StaticPart joinPointStaticPart) {
        return new CreditPermission(
                                    joinPointStaticPart.getSignature().getName
                                        ());
    }
}
```

Figure 5.38: CreditAuthAspect.aj - the authorization aspect for the *Credit* class

```java
import java.security.*;

public final class CreditPermission extends BasicPermission {
    public CreditPermission(String name) {
        super(name);
    }

    public CreditPermission(String name, String actions) {
        super(name, actions);
    }
}
```

Figure 5.39: CreditPermission.java - Extends the *Permission* class for the *Credit* class access control

This solution has a lot of advantages over the Object-Oriented approach. If someone wants to change which methods are protected, then changes only have to be done in one place: the aspect. Moreover, every call to a protected method is handled without having to add any additional code, which makes evolution of the code much, much easier.

Chapter 6

The UMLsec tool

6.1 Overview

The UMLsec tool helps the UMLsec developer designing secure models, by allowing him to check that the desired UMLsec properties are actually enforced by the model. When the tool detects a flaw in the model regarding a particular property, it can tell the developer where the problem is. The UMLsec tool can also automatically suggest corrections for models that do not meet some UMLsec properties. Finally, it can generate Java code with AspectJ aspects from an UMLsec model.

The UMLsec tool is *not* an UML modelling tool, and it is not capable of displaying a graphic representation of UML models. Instead, it uses models produced by ArgoUML, which is an Open Source, GPL-licenced tool, written in Java and available on Windows, GNU/Linux and Mac OS X.

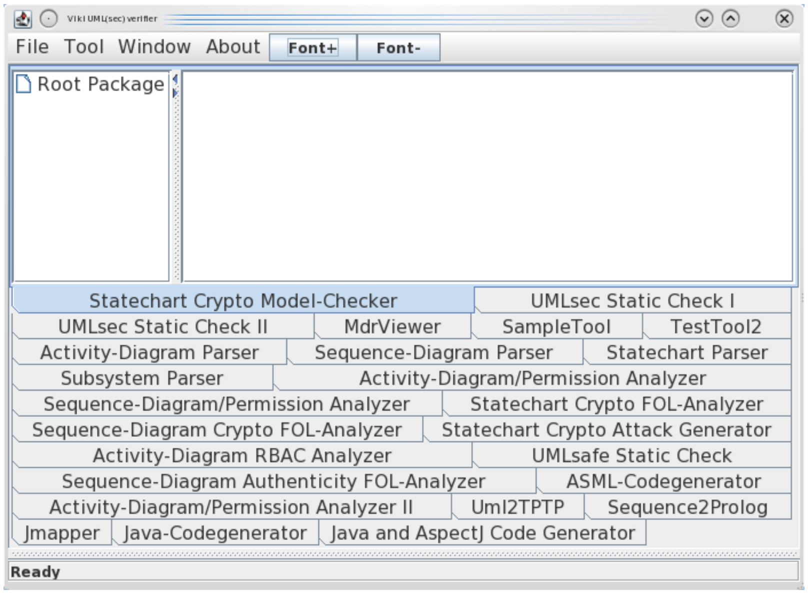

Viki UML(sec) verifier
File
Tool
Window
About
Font+
Font-
Root Package
Statechart Crypto Model-Checker
UMLsec Static Check I
UMLsec Static Check II
MdrViewer
SampleTool
TestTool2
Activity-Diagram Parser
Sequence-Diagram Parser
Statechart Parser
Subsystem Parser
Activity-Diagram/Permission Analyzer
Sequence-Diagram/Permission Analyzer
Statechart Crypto FOL-Analyzer
Sequence-Diagram Crypto FOL-Analyzer
Statechart Crypto Attack Generator
Activity-Diagram RBAC Analyzer
UMLsafe Static Check
Sequence-Diagram Authenticity FOL-Analyzer
ASML-Codegenerator
Activity-Diagram/Permission Analyzer II
Uml2TPTP
Sequence2Prolog
Jmapper
Java-Codegenerator
Java and AspectJ Code Generator
Ready

Figure 6.1: The UMLsec tool's GUI

The UMLsec tool can be used in three different ways : as an interactive console application, as a batch console application, or using a GUI, as shown on figure 6.1. A web interface also used to be available, but this feature is no longer maintained.

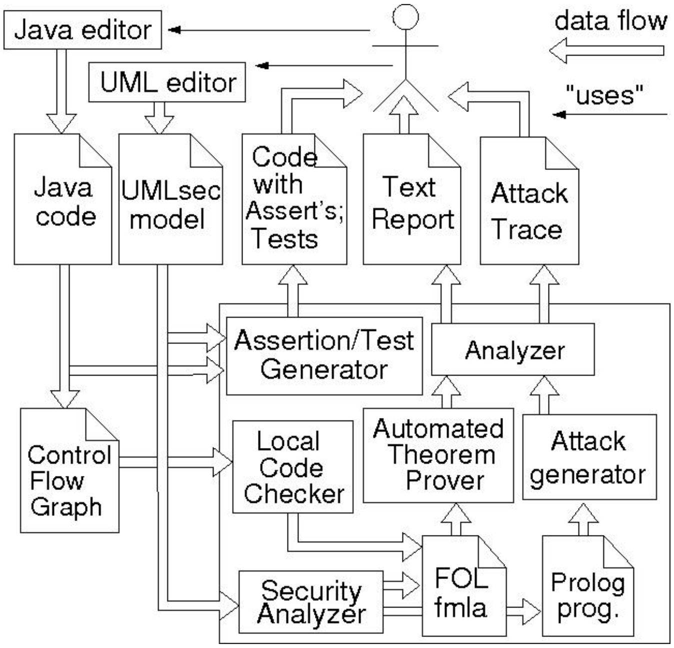


Figure 6.2: The UMLsec tool general architecture

Figure 6.2 depicts the general architecture of the tool. The tool receives UMLsec models that are produced by ArgoUML as an input, and produces code, text reports, UMLsec models or attack traces, depending on the property that is being checked. It is written in Java and AspectJ and works on all major platforms: Windows, GNU/Linux and Mac OS X. It can use Automated Theorem Provers and Prolog to detect and generate possible attack traces.

6.2 UMLsec properties checking

The main feature of the UMLsec tool is to check whether or not a model satisfies a particular UMLsec property. Doing it is really simple: first, a model has to be loaded in the tool. Using the GUI, this is done by selecting `File` then `Load` in the menu. The text zone at the bottom of the tool GUI indicates whether the model has been loaded successfully after a moment. Then, the user can select a check to perform from the list. The check is then performed, sometimes using external tools like a Prolog engine or an Automated Theorem Prover. Finally, when the user clicks on the tab corresponding to the check he just run, he can see the output, that will tell him if the model violates the UMLsec property or not.

6.3 Automatic correction of unsecure models

Automatic or semi-automatic correction of unsecure UMLsec models has been described in details in section 4.5. When a model is proven insecure, instead of letting the developer adapt it manually, we would like to correct it automatically, or at least partially automatically, using developer input when needed.

The UMLsec tool implements such mechanism for a few UMLsec properties. It is, of course, possible to extend it in the future in order to handle more UMLsec properties and more problem resolution scenarios.

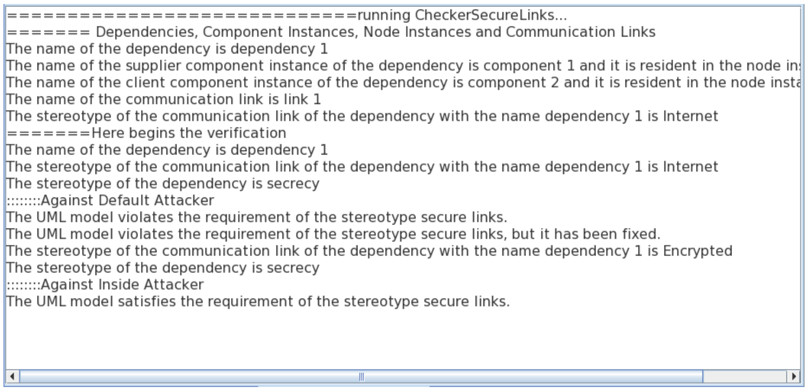
6.3.1 « secure links » property

The « secure links » property is the first one for which automated hardening has been implemented in the UMLsec tool. Whenever a « secure links » check is performed on a model using the tool, if it appears that the model doesn't fulfil the « secure links » property for a given attacker, then it is automatically modified, according to the hardening strategy described in section 4.5.2.

Once the model has been modified, a new version of the model can be exported by the user in a ArgoUML-readable format (a .zargo file, which is simply a zip archive containing an .xmi file describing the model, as well as a file describing the picture of every diagram).

For this particular property, the model can be modified in a completely automated way: there is no need for user input, and since there is only one possible solution that leads to a model fulfilling the « secure links » property *without* lowering the security mechanisms in place, there is also no need to ask the user to chose a solution in a set of possible ones.

However, some models can never fulfil the « secure links » property for a given attacker. There is no way to modify the model to make it secure with regard to the « secure links » property. In this case, the user is warned that the model is insecure and that the tool couldn't find any way to correct it.



```
==================================running CheckerSecureLinks...
======== Dependencies, Component Instances, Node Instances and Communication Links
The name of the dependency is dependency 1
The name of the supplier component instance of the dependency is component 1 and it is resident in the node in:
The name of the client component instance of the dependency is component 2 and it is resident in the node insta
The name of the communication link is link 1
The stereotype of the communication link of the dependency with the name dependency 1 is Internet
========Here begins the verification
The name of the dependency is dependency 1
The stereotype of the communication link of the dependency with the name dependency 1 is Internet
The stereotype of the dependency is secrecy
::::::::Against Default Attacker
The UML model violates the requirement of the stereotype secure links.
The UML model violates the requirement of the stereotype secure links, but it has been fixed.
The stereotype of the communication link of the dependency with the name dependency 1 is Encrypted
The stereotype of the dependency is secrecy
::::::::Against Inside Attacker
The UML model satisfies the requirement of the stereotype secure links.
```

Figure 6.3: UMLsec tool output for an unsecure model regarding the « secure links » property

Figure 6.3 shows the output produced by the UMLsec tool for a model that doesn't fulfil the « secure links » requirements with regard to the default attacker.

6.3.2 « secure dependency » property

Models that do not satisfy the « secure dependency » requirements can also be automatically modified, as explained in section 4.5.3. Exactly like for the « secure links » property, any model that is proven incorrect is modified. But here, the good thing is that there is *always* a way to get the model to be valid regarding the « secure dependency » property, without lowering any security mechanism already in place.

Like the « secure links » property, modification of a model in order to fulfil the « secure dependency » is completely automatic, and doesn't require any user choice or additional input.

6.3.3 « permission » property

Finally, implementation of automated hardening of the « permission » property is almost completed, following the transformation strategies described in 4.5.4. Some problems still arise due to the fact that ArgoUML's support of sequence diagrams is incomplete.

Here, user input might be requested when a model is found to be incorrect regarding the « permission » property. Most of the time, several choices will be available, and the user will be prompted to choose one of them.

6.4 Code generation

Code generation from models with selected UMLsec properties has also been implemented in the UMLsec tools. Code generation from UMLsec enabled models has been discussed in chapter 5, where two possible approaches were described: the first was a strict Object-Oriented Programming approach, and the second was using Aspect-Oriented Programming on top of Object-Oriented Programming.

The chosen approach is Aspect-Oriented Programming, because of its ability to put the security-related code out of the way of the developer. Since the UML diagrams without the UMLsec extensions are first processed in order to produce plain Object-Oriented code, and the UMLsec properties are enforced using aspects, the developer has the opportunity to extend its business logic code without worrying about the security code. And since every UMLsec property is implemented in one aspect, he can easily chose which ones he wants to activate or not. That is a great advantage for code readability, and to avoid code tangling and security code scattering.

Keeping a traceability link between the model and the code is also a lot easier with this Aspect-Oriented approach: since every UMLsec property is implemented in one aspect, it is possible to derive the property from the code just by examining the aspect.

The Object-Oriented language that has been chosen is Java, together with is widely-used Aspect-Oriented extension AspectJ.

But we also chose to use Aspect-Oriented Programming for the implementation of the code generator itself, where it provides several advantages over the classical Object-Oriented Programming approach.

Using Aspect-Oriented Programming allows to write the code that generates code from UML diagrams (without the UMLsec extensions) only once, and then to plug one aspects for every UMLsec property we want to support. That is a huge advantage over the Object-Oriented Programming approach, that requires a more complicated and less modular way of generating the code, since the generation of code from pure UML diagrams and UMLsec extensions were mixed. This way, when we want to add support for generating code for a new UMLsec property, we only have to write an aspect that will enforce it by generating... an aspect.

This solution even allows us to provide several implementations of the code generator for one specific UMLsec property. Let's take the « permission » property, for example. Section 5.2 described the code generation mechanism extensively, using an authorisation API. But one might want to use another authorisation API for some reason. Well, he would just have to write an aspect for that, and replace the existing one with his own, without having to touch the Object-Oriented code.

Currently, only the code generation from a « rbac » property is implemented completely. Support for the « permission » property is being implemented, but again, the poor support of sequence diagrams by ArgoUML makes it hard to complete. A previous implementation of code generation from a « permission » property already exists, but it seens that the switch from Poseidon to ArgoUML (that will be discussed in section 6.5.1) made it obsolete and buggy. It has been decided to rewrite this code completely, this time using Aspect-Oriented Programming.

6.5 UML tool migration

6.5.1 From Poseidon to ArgoUML

Initially, the UMLsec tool was designed to accept models created by Poseidon as an input. Poseidon [pos09] was a commercial fork of ArgoUML, an open source (BSD licence) UML modelling tool. Although Poseidon was a commercial tool, its editor, Gentleware [?], was giving away a *Community Edition* of Poseidon free of charge. This was of great interest since Poseidon was more complete than ArgoUML, especially for the implementation of the sequence diagrams.

However, someday Gentleware decided to stop giving away the *Community Edition* of Poseidon for free, and it was decided to switch to ArgoUML as the modelling tool used to write models that would then be loaded into the UMLsec tool. The problem is that, even though both Poseidon and ArgoUML come from the same codebase, they both evolved in different directions. So switching from one tool to another lead to quite a lot of bugs. As today, some of them have already been solved, but some are unfortunately still there.

6.5.2 Exporting models to ArgoUML

Since the automated hardening of some UMLsec properties has been implemented in the UMLsec tool, the model, once modified by the tool, needed to be readable by the developer. Therefore, a function allowing one to export the loaded model back to a ArgoUML-readable format was needed. This function has been implemented, and it is now possible to save the current model either in xmi format or in .zargo format, both of them being readable by ArgoUML.

In order to do this, the version of MDR used by the UMLsec tool had to be updated so that it would match the one used by ArgoUML. MDR (MetaData Repository) [Mic] is an implementation of the OMG's MOF [OMG] that used to be part of the Netbeans IDE. It has now been abandoned and it isn't supported anymore. ArgoUML developers are slowly considering the possibility of moving to another MOF implementation. If this happens, then maybe a similar move will have to be considered for the UMLsec tool.

6.6 Development process

Improvements have also been accomplished not in the tool itself, but in the software engineering approach for it development.

First, Unit tests have been introduced, using the JUnit 4 framework [jun]. This allow the UMLsec tool developers to be more confident in the quality of the code, and helps to make sure that no regressions arise from code modification.

Logs have also been introduced. The log4j [Foua] logging framework has been used in order to provide traces when a bug is spotted, or when a problem arises. Using log4j allowed the UMLsec tool developers to work more efficiently on solving problems.

Finally, a continuous integration server has been set up, that automatically makes sure that the last svn version of the tool compiles and passes all the tests.

6.7 Future works

The UMLsec tool is a great way of working with UMLsec, allowing one to automatically ensure that the models he writes are correct regarding the UMLsec properties he wants to enforce. However, there is more work to be done in order to make it even better.

A first thing would of course be to extend the number of UMLsec properties that can benefit from automated hardening. Also, the properties that already have automated hardening support could see it getting even better with the addition of new problem resolution strategies.

It would be even better if the user could define its own resolution strategies using a specialized high-level programming language. This would allow him to easily extend the possibilities of the UMLsec tool without having to modify its code, and to share and reuse his own strategies.

Automated code generation could also be spread to the remaining UMLsec properties, and special care should be taken in order to make sure that properties do not cause security problems when interacting. Since aspects are used to implement the UMLsec properties, this problem can be seen as an aspect composition problem.

There are still some bugs left from the switch from Poseidon to ArgoUML that need to be fixed. Also, generalisation of log4j and JUnit test cases would help a lot in making the code more reliable.

Also, efforts have already been made in order to ease the installation and compilation of the tool, by writing `ant` scripts that automatically build and run the tool, and are able to detect the necessary third-party software, like Prolog for example. Those efforts should be continued in order to make the build and run processes even easier.

Finally, one of the weaknesses of the UMLsec tool is its lack of possibility to visualise the UMLsec models. People need to see the UML diagrams. This is especially true for the automated modification of models: when the user is asked to chose one solution in a set of possible ones, not being able to see what the result of each solution would be is a big handicap.

Chapter 7

Conclusion

UMLsec is a strong, formally-defined way of dealing with security requirements early during the development cycle, since it is used during the modelling phase. Moreover, the fact that it is defined using the UML extension mechanism makes it easy to use and to learn.

The automated correction of incorrect UMLsec models, although still incomplete, seems very promising in helping the developer to design secure software and have a formal proof of the security requirements it actually enforces.

Automatic code generation addresses one of the weakest points of this approach: the likelihood of human error during the translation of the UMLsec model into code. By generating the code automatically, the UMLsec tool produces a secure code base that can then be adapted to fit the developer's needs. Of course, bugs and problems could still arise during that adaptation phase, but having a secure code in which the developer can be confident at the start is an interesting first step.

Finally, Aspect-Oriented Programming allowed us to completely separate security concerns from the functional code, and even to separate the security concerns themselves, achieving a far better modularisation than with a classical Object-Oriented Programming approach, and avoiding the code scattering and code tangling problems. Of course, Aspect-Oriented Programming brings its own potential problems, especially when it comes to aspects composition, but we believe that the gain in clarity an adaptability outweighs those drawbacks.

The original contributions submitted in this thesis include work on extending UMLsec, on better integration of UMLsec properties into all the UML diagrams of a model, and on conflicts identification and resolution between UMLsec properties.

Another important contribution is the automated correction of unsecure UMLsec models. Also, the production of verified code can lead to interesting developments, especially since the Aspect-Oriented approach has been introduced in order to facilitate manual evolution of the generated code as well as traceability links.

But the orginial contributions discussed here and not only theorical, but also more concrete, with the development of the UMLsec tool, as well as the definition and set up of several development processes, like unit tests, continuous integration, versionning, ...

There is still lots of work to be done in the field of UMLsec, automated correction of models and Aspect-Oriented Programming-based code generation, but hopefully the elements described in this document are a first useful step.

The automated hardening of UMLsec models could be improved a lot by defining new solutions to common problems and by extending the set of UMLsec properties for whose a model can be automatically modified. But also, one should take care of which transformations will alter other UMLsec properties, and which won't. This way, the user could be warned that using a specific resolution strategy might lead to the violation of another UMLsec property he wants to enforce on the model.

UMLsec in itself can always be extended to match everyone's needs. However, the access control related stereotypes could be improved, for example by extending the RBAC stereotype. It has already been extended in this document, but other useful extensions, like the ability to model the activation and deactivation of roles during the execution, would be interesting.

The potential conflicts between UMLsec properties should also be studied more extensively. So far, UMLsec properties can be checked independently, but in practice, one will want to enforce several of them at the same time. We need to make sure that the addition of one property won't violate the existing ones.

This is also true when generating code from the UMLsec models using Aspect-Oriented Programming. When multiple UMLsec properties need to be enforced, the proposed generation strategy will generate at least one aspect for each property. Those could potentially interact in an undesired way, producing side effects that would jeopardise the safety of the resulting software regarding the UMLsec properties it is supposed to enforce. Therefore, questions like aspects precedence and aspects composition should be further discussed.

Bibliography

[ABB$^+$03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control, June 2003.

[Cen02] Palo Alto Research Center. Frequently asked questions about aspectj, 2002. http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/doc/faq.html (Accessed August 2009).

[Den76] Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19 (5):236–243, 1976.

[DMN] Ole-Johan Dahl, Bjørm Myhrhaug, and Kristen Nygaard. The simula programming language. http://www.edelweb.fr/Simula/ (Accessed August 2009).

[ELF08] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in uml design models. In *ASE*, pages 99–108. IEEE, 2008.

[FK92] D.F. Ferraiolo and D.R. Kuhn. Role based access control. 1992.

[Foua] Apache Software Foundation. Log4j logging framework, http://logging.apache.org/log4j/1.2/index.html. (Accessed Sept. 2008).

[Foub] The Apache Foundation. Apache tomcat. http://tomcat.apache.org/ (Accessed August 2009).

[Fouc] The Eclipse Foundation. Eclipse public licence - v1.0. http://www.eclipse.org/org/documents/epl-v10.php (Accessed August 2009).

[Foud] The Eclipse Foundation. Eclipse.org. http://www.eclipse.org (Accessed August 2009).

[IKL$^+$97] John Irwin, Gregor Kiczales, John Lamping, Jean-Marc Loingtier, Chris Maeda, Anurag Mendhekar, and Cristina Videira Lopes. Aspect-oriented programming. *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[JLW05] Jan Jürjens, Markus Lehrhuber, and Guido Wimmel. Model-based design and analysis of permission-based security. In *ICECCS*, pages 224–233. IEEE Computer Society, 2005.

[jun] Junit framework, http://www.junit.org/. (Accessed Sept. 2008).

[Jür05] Jan Jürjens. *Secure Systems Development with UML.* Springer-Verlag, 2005.

[Jü04] Jan Jürjens. Umlsec tool, 2004. Published at http://mcs.open.ac.uk/jj2924/umlsectool/index.html (Accessed Sept. 2008).

[Kay] Alan C. Kay. The early history of smalltalk. http://gagne.homedns.org/ tgagne/contrib/EarlyHistoryST.html (Access August 2009).

[Kuh98] D.R. Kuhn. Role based access control on mls systems without kernel changes. In *Third ACM Workshop on Role Based Access Control*, pages 25–32, 1998.

[Lad03] Ramnivas Laddad. *AspectJ in action: Practical Aspect-Oriented Programming.* Manning, 2003.

[Mic] Sun Microsystems. Metadata repository - mdr. http://mdr.netbeans.org/ (Accessed September 2008).

[Mic01] SUN Microsystems. Jaas tutorials, 2001. http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/tut (Accessed December 2008).

[MJY^{+}09] Lionel Montrieux, Jan Jürjens, Yijun Yu, Jörg Schreck, and Pierre-Yves Schobbens. Automated security hardening for umlsec models, January 2009. Software Engineering Workshop, Trento.

[NIS] NIST. Role based access control - frequently asked questions. http://csrc.nist.gov/groups/SNS/rbac/faq.html (Accessed October 2008).

[Nov] Novell. Apparmor. http://forge.novell.com/modules/xfmod/project/?apparmor (Accessed August 2009).

[oC] U.S. Department of Commerce. National institute of standards and technology - nist. http://www.nist.gov/index.html (Accessed November 2008).

[oD85] Department of Defense. Trusted computer system evaluation criteria, December 1985. http://nsi.org/Library/Compsec/orangebo.txt.

[OMG] Object Management Group OMG. Mof 2.0 formal specification. http://www.omg.org/spec/MOF/2.0/ (Accessed August 2009).

[OMG01] Object Management Group OMG. Uml 1.4 formal specification, 2001. http://www.omg.org/spec/UML/1.4/ (Accessed August 2009).

[OP00] S. Oh and S. Park. Task-role based access control (T-RBAC): An improved access control model for enterprise environment. *Lecture Notes in Computer Science*, 1873:264–273, 2000.

[pos09] Poseidon website, February 2009. http://www.gentleware.com/.

[SCFY96] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SFK00] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.

[Sou] Spring Source. Spring j2ee framework. http://www.springsource.org/about (Accessed August 2009).

[Wat] Robert Watson. The trustedbsd project. http://www.trustedbsd.org/ (Accessed August 2009).

[WCS$^+$02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In Dan Boneh, editor, *USENIX Security Symposium*, pages 17–31. USENIX, 2002.

[xer] Xerox parc (now parc). http://www.parc.com (Accessed August 2009).

[YT05] E. Yuan and J. Tong. Attribute based access control (ABAC): a new access control approach for service oriented architectures. In *Ottawa New Challenges for Access Control Workshop*, volume 27, 2005.

Index