SECURE OBJECT-ORIENTED DATABASES

by

MARTIN STEPHANUS OLIVIER

THESIS

submitted in fulfilment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

FACULTY OF NATURAL SCIENCES

of the

RAND AFRIKAANS UNIVERSITY

PROMOTER: PROF S.H. VON SOLMS

DECEMBER 1991

Contents

Prologue

Many people contribute to one's work. Those who have contributed to this work on a technical level are acknowledged in the bibliography at the back of this thesis. Others have contributed on a personal level, especially with their interest and encouragement. In the opinion of this author such personal participation has more value in itself than any technical contribution made by this work, and is therefore greatly appreciated by the author.

Firstly, my thanks to prof Von Solms for suggesting the topic of this research and for his advice throughout the project.

Thanks also to my parents, family and friends. I am indebted to you for the support and encouragement that you demonstrated in so many ways.

In the final instance, thanks and praise to God Almighty, by whose grace all that we achieve is accomplished.

If the LORD *does not build the house, the work of the builders is useless*

Psalm 127:1
Good News Bible

Summary

The need for security in a database is obvious. Object-orientation enables databases to be used in applications where other database models are not adequate. It is thus clear that security of object-oriented databases must be investigated.

Overview

This work investigates secure object-oriented databases. Firstly, we propose a new model, SECDB, for such databases. SECDB differs substantially from other proposals for such models. Secondly, we propose a taxonomy for secure object-oriented databases. The taxonomy identifies a number of design parameters—aspects that may differ from one such model to the next. It also indicates implications that specific choices for one design parameter have on the choice of other design parameters and on other aspects of the model. Thirdly, we propose an initial model for discretionary security in object-oriented databases, DISCO. DISCO illustrates how results from the taxonomy may be applied when a new security model is developed. A brief description of the work covered in each of these cases follows.

This work focuses on the secrecy aspect of security; integrity remains a major and essentially unsolved problem in secure databases.

SECDB

The first model proposed by us (SECDB) extends object-oriented databases to enable individual objects to take responsibility for security—ie to protect themselves. This extension is based on the concept of 'baggage'—baggage is collected from all components involved in any request; this baggage may then be verified by the object against its personal security profile before a method is executed or a variable is accessed. Note that the profile has the complete access path of such a request available to base its decision on.

The taxonomy

Models for secure object-oriented databases differ in many respects, because they focus on different aspects of the security problem, because they make different assumptions about what constitutes a secure database or because they make different assumptions about the object-oriented model. The taxonomy we propose may be used to compare the various models: Models that focus on specific issues may be positioned in the broader context with the aid of the taxonomy. The taxonomy also identifies eight major aspects where security models may differ and indicates some alternatives available to the system designer for each such parameter. We also indicate implications of using specific alternatives.

Since differences between models for secure object-oriented databases are often subtle, a formal notation is necessary for a proper comparison. Such a formal notation also facilitates the formal derivation of restrictions that apply under specific conditions. The formal approach further gives a clear indication about the assumptions made by us—given as axioms—and the consequences of those assumptions (and of design choices made by the model designer)—given as theorems.

DISCO

Lastly, we propose a discretionary security model for object-oriented databases (DISCO). Entities in the database are protected by capabilities. A subject that possesses a capability is authorised to access the corresponding entity. Additionally, under certain conditions, a subject may pass the capability on to another subject, authorising this other subject to access the protected entity. Passing the capability on is done at the first subject's discretion, hence the term *discretionary* security.

The object-oriented model has a rich variety of entities with relationships between such entities. A subject that passes a capability on to another subject may (inadvertently) authorise the second subject to access more entities than intended. We describe the restrictions that apply to the transfer of capabilities to safeguard against such an unintended disclosure of information. Similarly, we consider the restrictions that apply when capabilities are revoked.

Afrikaanse Oorsig

Daar bestaan 'n verskeidenheid redes waarom dit nodig mag wees om inligting te beskerm: in 'n besigheid is inligting 'n bate wat die besigheid 'n finansiële voorsprong bo mededingers kan gee; in die militêre omgewing is geheimhouding van inligting dikwels van lewensbelang; voorts mag daar sosiale, etiese en selfs regsredes wees waarom inligting nie aan ongemagtigde persone bekendgemaak behoort te word nie. Namate rekenaars meer gebruik word om inligting te verwerk, raak dit des te belangriker dat kontroles gevind word om te verseker dat inligting nie kwaadwilliglik of per abuis openbaar word aan ongemagtigde persone nie.

Navorsing aangaande veilige databasisse is dus van wesenlike belang en het, inderdaad, reeds heelwat navorsingsaandag ontvang. Sodanige navorsing het egter grootliks op relasionele databasisse gekonsentreer. Die objekgeoriënteerde databasismodel is egter geskik vir toepassings waar die relasionele model tekortskiet. Dit is dus nodig om ook veiligheidsmeganismes vir sulke databasisse te vind. 'n Aantal modelle vir veilige objekgeoriënteerde databasisse is reeds voorgestel. Dit is duidelik van hierdie voorstelle dat 'n veilige objekgeoriënteerde databasis 'n hele verskeidenheid aspekte het wat aandag verg. Die primêre bydrae van hierdie projek is die identifikasie van sulke aspekte, asook 'n aantal aanduidings watter invloed 'n keuse ten opsigte van 'n gegewe aspek op ander aspekte van die model het. Hierdie resultate word as 'n taksonomie vir veilige objekgeoriënteerde databasisse gegee.

Benewens die taksonomie stel ons ook twee modelle vir veilige objekgeoriënteerde databasisse voor. Die eerste model, SECDB, beperk die vloei van inligting binne (en uit) die databasis sodat sulke inligting nie na plekke vloei waar 'n ongemagtigde gebruiker dit kan bereik nie. Die tweede model, DISCO, illustreer diskresionêre sekerheidsmodelle in objekgeoriënteerde databasisse. In 'n diskresionêre model beskik gebruikers oor regte ten opsigte van sommige entiteite; sulke regte kan, na die 'eienaar' van die entiteit se diskresie, ook aan ander gebruikers oorgedra word. DISCO maak van die taksonomie gebruik om 'n aanduiding te gee van die beperkings wat geld ten opsigte van sodanige oordrag van regte. Ons lig die twee

modelle en die taksonomie vervolgens kortliks toe.

SECDB

SECDB is 'n model vir 'n veilige objekgeoriënteerde databasis. 'n Versoek in 'n objekgeoriënteerde databasis bestaan uit 'n reeks boodskappe wat tussen objekte gestuur word. SECDB hou boek van die roete wat so 'n versoek deur die databasis volg: Inligting word versamel aangaande alle objekte en metodes, asook apparatuur wat by enige versoek betrokke is; sulke inligting staan as bagasie bekend. Vir elke entiteit wat beskerm word, word 'n profiel geskep wat in besonderhede beskryf wie toegang tot die entiteit mag verkry en watter roetes deur so 'n gemagtigde subjek gebruik mag word en watter nie. Voor 'n versoek toegang tot enige entiteit verkry, ondersoek die profiel die bagasie wat met die versoek geassosieer is, en bepaal of aan al die toegangsvereistes voldoen is. Hierdie toegangsbeheer is op die pad-konteksmodel van Boshoff en Von Solms gebaseer; ons toon in hierdie proefskrif hoe hierdie model aangepas kan word vir die objekgeoriënteerde geval.

SECDB hou, bo en behalwe die bagasie van 'n versoek, rekord van die sensitiwiteit van die inligting waaroor enige metode beskik. Dit word gedoen deur die profiel(e) van enige inligting wat (potensieel) deur 'n metode bekom word, met die metode te assosieer—sien die volledige beskrywing van SECDB in die proefskrif vir besonderhede. Telkens wanneer 'n metode inligting na 'n veranderlike skryf, word die profiele wat met die metode geassosieer is, ook met die veranderlike geassosieer; dit verseker dat inligting nie na 'n plek geskryf word waar 'n ongemagtigde gebruiker toegang daartoe kan verkry nie.

SECDB verskil aansienlik van ander modelle vir veilige objekgeoriënteerde databasisse: beide die konsep van bagasie en die aanwending (en veral die 'rondstuur') van profiele kom nie by vergelykbare modelle voor nie. SECDB verbreed dus die spektrum modelle wat gebruik word om die taksonomie van veilige objekgeoriënteerde databasisse voor te stel.

Die taksonomie

'n Aantal modelle vir veilige objekgeoriënteerde databasisse is reeds in die literatuur voorgestel. Hierdie modelle openbaar 'n verbasende diversiteit, deels omdat die modelle verskillende aspekte van sulke databasisse in gedagte het. Die taksonomie beskryf agt sulke aspekte sistematies; dit toon ook hoe spesifieke keuses vir sommige aspekte ander aspekte van die databasis kan beïnvloed.

Die eerste groep aspekte (of parameters) het te doene met die semantiek van die beskerming van entiteite: Parameter X1.1 beskryf die onderliggende model; dit kan byvoorbeeld die militêre model van inligtingklassifikasie wees, of toegangsbeheerlyste, of vermoëns. Parameter X1.2 spesifiseer die betekenis wat geheg word aan die feit dat 'n entiteit beskerm is; dit kan in een model byvoorbeeld beteken dat 'n ongemagtigde gebruiker nie toegang tot so 'n entiteit kan verkry nie, terwyl 'n ander model kan vereis dat 'n ongemagtigde gebruiker nie van die bestaan van so 'n beskermde entititeit bewus mag wees nie.

Die tweede groep parameters handel oor sekerheidsaspekte wat voortvloei uit die (objekgeoriënteerde) struktuur van die databasis. Parameter X2.1 spesifiseer watter entiteite beskerm kan word in 'n gegewe model; dit kan byvoorbeeld objekte, veranderlikes, metodes en/of klasse wees. Parameter X2.2 beskryf hoe die aanvanklike 'beskermingsvlak' van 'n entiteit bepaal word; dit kan byvoorbeeld geërf word van die klas waar die entiteit gedefinieer is. Parameter X2.3 beskryf die verwantskappe wat tussen die 'beskermingsvlakke' van verwante entiteite kan (en moet) bestaan; daar word byvoorbeeld aangetoon dat 'n voorkoms van 'n klas minstens so beskermd as die klas self moet wees.

Die derde (en finale) groep parameters beskryf aspekte wat na vore tree tydens die (dinamiese) werking van die databasis. Parameter X3.1 toon 'n aantal wyses waarop die magtiging (klaring) van 'n aktiewe metode bepaal kan word; in die eenvoudigste geval sal dit slegs afhang van die magtiging van die gebruiker van wie die oorspronklike versoek kom. Parameter X3.2 toon hoe die sensitiwiteit van inligting waaroor 'n aktiewe metode beskik bepaal kan word; sien die bespreking van SECDB hierbo ter illustrasie. Parameter X3.3 beskryf maatreëls wat getref kan word om te verseker dat inligting nie blootgestel word aan ongemagtigde toegang wanneer dit na 'n veranderlike geskryf word nie.

DISCO

Die tweede model wat ons voorstel is vir diskresionêre sekerheid in 'n objekgeoriënteerde databasis. Ons stel hierdie model, DISCO, voor om te illustreer hoe die taksonomie gebruik kan word wanneer nuwe modelle ontwerp word. DISCO is dus nie 'n omvattende model van diskresionêre sekerheid in objekgeoriënteerde databasisse nie; dit is egter uitgebreid genoeg om beide die aanwending van die taksonomie en die implikasies van diskresionêre sekerheid in objekgeoriënteerde databasisse te illustreer.

DISCO maak gebruik van vermoëns om entiteite te beskerm: 'n Subjek wat in besit is van 'n vermoë vir 'n spesifieke entiteit is gemagtig om toegang tot daardie entiteit te verkry. Voorts mag die 'eienaar' van die entiteit vermoëns vir die entiteit aan ander subjekte verskaf of van ander subjekte terugtrek. DISCO toon dat die eienaar nie willekeurig vermoëns kan uitdeel en terugtrek nie: die objekgeoriënteerde struktuur veroorsaak dat 'n eienaar (onopsetlik) meer regte mag toestaan as bedoel wanneer 'n vermoë veskaf word; soortgelyk is dit moontlik dat nie alle bedoelde regte teruggetrek word wanneer 'n vermoë teruggetrek word nie. DISCO toon watter beperkings geld vir die spesifieke omstandighede wat in DISCO geld; eenvoudiger (strenger) beperkings word dan voorgestel.

Slotsom

Ons hoop dat die werk in hierdie proefskrif die huidige werk aangaande veilige objekgeoriënteerde databasisse in perspektief plaas. Voorts behoort die werk wat in die taksonomie beskryf is, 'n positiewe bydrae tot toekomstige modelle vir sulke databasisse te maak. Laastens raak die twee modelle, SECDB en DISCO, aan 'n aantal buitengewone aspekte; ons hoop dat die melding van hierdie aspekte verdere navorsing sal stimuleer.

Om jou één eie taal tot nut te gebruik is 'n groter kuns as om sewe vreemde by te leer.

C. J. Langenhoven

Chapter 1

Aims and Scope

Chapter 1 serves as a road map for the remainder of the thesis: it delineates the research area and describes how the chapters of the thesis are organised to address the research questions.

'Where shall I begin, please your Majesty?'
he asked.
'Begin at the beginning,' the King said,
gravely, 'and go on till you come to the end:
then stop.'

Lewis Carroll
Alice's Adventures in Wonderland, ch.11

1.1 Introduction

This work deals with secure object-oriented databases. The primary contributions to the field contained herein are:

- A proposed theoretical model for secure object-oriented databases (called SECDB);
- A taxonomy that may be used to compare various models for secure object-oriented databases; and
- An initial theoretical model for discretionary security in an object-oriented database (called DISCO).

Note that the focus is on secrecy in secure databases; integrity remains a major problem we do not address in this work.

The next section explains the interest in secure object-oriented databases. The last section of this chapter outlines the approach followed in this work.

1.2 Secure object-oriented databases

Information is an asset that needs protection against unauthorised access; this explains the need for secure databases. Further, a database is a shared resource and not everybody authorised to access some information is necessarily authorised to access all the information in the database—hence the need for multilevel secure databases.

The research effort regarding multilevel secure databases has primarily focused on relational databases. Since the object-oriented database model differs substantially from the relational model, results obtained for relational databases, as well as models proposed for relational databases, are not necessarily applicable to object-oriented databases. Amongst other issues, inheritance and the inclusion of methods in the database pose challenges in a secure object-oriented database. It is therefore necessary to extend the research on secure databases to include the object-oriented model.

The fact that sensitive information in an object-oriented database needs protection (just like sensitive information in any other database) is not the only reason for studying multilevel secure object-oriented databases—object-oriented databases have potential benefits for security that are not present in other database models. Firstly, since methods are stored as part of such a database, it is possible to protect sensitive information by restricting access to the relevant methods. Restriction of access to methods often provides a more natural protection of information than protection of variables does: it is relatively easy to determine which employees has to perform an operation on a given entity as part of their jobs (for example from job description documents); if the database closely models the real world, the methods supported by objects will in many cases reflect the day to day tasks of employees; this makes it relatively simple to determine appropriate access restrictions to methods based on the job functions of employees. This is known as *role-based* security.

The second security benefit stems from the encapsulation feature of object-orientation: Encapsulation restricts the ways in which an object may be accessed. The system prevents direct access to all instance variables (and possibly access to private methods). Firstly, this means that a user can only manipulate data in ways specified beforehand, and not in any *ad hoc* way. Secondly, it is possible to include the protection of an object as an integrated part of the object, similar to the way the data and methods are integrated parts of the object. If protection forms part of the object, it means that the object may flow, without any restrictions, to another location in the system without compromising security—the information encapsulated by the object will be just as protected in the new location as it was in the original location.

1.3 Outline of this work

The primary goal of this work is the proposal of a taxonomy for secure object-oriented databases. To reach this goal this work is structured as follows:

- Brief overviews of existing background material are given: chapter 2 describes the fundamentals of computer security, chapter 3 describes object-orientation and chapter 4 database theory, including the fundamentals of secure databases. The intentions of these chapters are to introduce terminology and to give references to existing literature.

- In order to make the taxonomy applicable as widely as possible we propose a new model, SECDB, for a secure object-oriented database; see chapter 6. SECDB differs in many respects from comparable models, amongst others the fact that SECDB is based on the Path Context Security Model (PCM) which we had to adapt for the object-oriented environment—see chapter 5 for details.

- The taxonomy for secure object-oriented databases will be found in chapters 7 through 11.

- A model for discretionary access control in object-oriented databases, DISCO, is proposed in chapter 12. It illustrates how the results obtained from the taxonomy may be applied when a new security model is proposed.
- An evaluation of the research, together with suggestions for future work, are contained in chapter 13.

In order to limit the extent of this work, we had to confine our attention to the issues described above. The fact that some issues had to be excluded does not indicate that we do not consider them as important, but only that they are not directly relevant to this work, or that they fit logically after this work.

Chapter 2

Computer Security

The three cornerstones of this thesis are computer security, object-orientation and database theory. Chapter 2 consists of an overview of the relevant computer security theory. Object-orientation and database theory will receive attention in the following two chapters.

In this work we will primarily be concerned with the secrecy aspect of computer security, in other words, ensuring that sensitive information is not (accidentally or intentionally) disclosed to unauthorised parties.

I know that's a secret for it's whispered every where.

William Congreve
Love for Love III.iii

'If everybody minded their own business,' said the Duchess in a hoarse growl, 'the world would go round a deal faster than in does.'

Lewis Carrol
Alice's Adventures in Wonderland, ch.6

2.1 Introduction

This chapter serves as a brief introduction to general computer security. It is intended for positioning of subsequent chapters rather than providing an overview of the field. Readers requiring an overview or a thorough background is referred to one of the many textbooks on the topic, for example [Pfl89]; the reader is also referred to [Pfl89] for details on any aspect in this chapter for which an explicit reference is not given.

2.2 Aspects of security

2.2.1 Secrecy, integrity and availability

Computer security has three goals:

- *Secrecy* of protected entities;
- *Integrity* of information; and
- *Availability* of service to authorised users.

Secrecy measures in a secure system ensure that a user (or *subject*) does not obtain (read) access to any protected entity (information, software, equipment, etc) in the system[1]. The secrecy goal often includes restrictions on the flow of information to ensure that a subject does not read sensitive information and store it in a less sensitive location where a less trusted subject may access it.

Integrity measures ensure that the information in the system is reliable; integrity is ensured by, amongst others

- Disallowing unauthorised users to modify or add information;
- Adding controls to limit an authorised user's ability to modify (or insert) information where the new values are inconsistent with other information in the system or the new values are obviously wrong;
- Inhibiting the flow of unreliable information to locations where it will be accepted as reliable; and

[1]Items in a computer system that may be accessed by a subject are normally referred to as *objects*; in this work we will use the term *entity* to refer to such an item and the term *object* to refer to an object in the object-oriented sense—see chapter 3

- Ensuring that physical phenomena (such as power failures, natural disasters, etc) do not cause inconsistencies or other integrity problems in the system.

Measures to ensure *availability* of service to authorised users include aspects such as the following:

- Ensuring that measures for secrecy and integrity do not hinder authorised users;

- Ensuring that other users (both authorised and unauthorised) do not monopolise the system such that an authorised user is denied service; and

- Establishing facilities and procedures to ensure that work can continue despite physical phenomena (such as power failures, natural disasters, etc).

In all these cases a threat may be intentional—from someone who deliberately sets out to access or damage the system. The threat may also be unintentional—a user can unknowingly do something that may (eventually) compromise security. Security measures have to address all threats, whether intentional or unintentional.

In this work we are concerned with secrecy and with those aspects of integrity that ensure that an unauthorised user does not modify information.

2.2.2 Aspects to protect

A number of aspects of a system has to be protected to maintain security. The three primary areas are

- *Physical protection* to prevent unauthorised parties from reaching the hardware, also to protect the hardware against the environment;

- *Communication protection* to prevent parties from obtaining information from the medium that is used to transmit the information and, also, to prevent such parties from directly putting information on such a medium; and

- *Logical protection* to prevent a user from accessing (using) the system, where the user does have physical access to (parts of) the system.

These areas lead to secondary goals, which may overlap: For both physical and logical protection it is, for example, necessary to *authenticate* a potential user–in other words, to ensure that a person is indeed who that person professes to be.

In this work we are only concerned with logical protection. This is especially important on a system where not every (legitimate) user is authorised to access everything on the system; on such a system it is necessary to ensure that every user is only allowed access to those parts of the system that the user has been authorised to access. A system that only allows users access to those parts of the system they are authorised to access, is known as a *multilevel secure* (MLS) system.

2.2.3 Mandatory security versus discretionary security

Logical protection is often subdivided into *mandatory security* and *discretionary security.*

Mandatory security refers to the security that is implemented based on the sensitivity of the information. The sensitivity of information is usually not determined by the users, but by a person with this specific responsibility in the organisation (often known as the *system security officer*). This person also determines the clearance of users of the system: The clearance of a user is an indication of the sensitivity of information the user is allowed to access. This clearance is based on the role of the user in the organisation and on the level of trust associated with the user.

Mandatory security will ensure that a user without the necessary clearance is prevented from directly or indirectly accessing protected information.

Note that, where a user 'owns' information, the owner does not have the power to grant access to another user who does not have the necessary clearance.

Discretionary security refers to the rights that users have to access protected entities and to the *discretionary* power they have to grant such rights to other users as well as to revoke such rights from other users.

In this work we are primarily concerned with mandatory security. We refer to the differences and similarities between mandatory and discretionary security in section 11.4. A model for discretionary security in object-oriented databases is proposed in chapter 12.

2.2.4 Development stages

Pfleeger [Pfl89] lists four stages in the development of a secure system:

1. *Modeling*, where a (mathematical) model is developed that reflects the environment in which the system will operate and describes the strategy that will be used to ensure security in the system;

2. *Design*, where a strategy to implement the model is selected (or developed);

3. *Demonstrating trust*, where it is indicated that the implementation strategy accurately represents the model and where the design is scrutinised for flaws; and

4. *Implementation*, where the design is implemented and tested.

In this work we are only concerned with models for secure systems (ie stage 1). The next section introduces some existing security models.

2.3 Security models

A number of security models have been proposed with various goals. We briefly look at some of these.

2.3.1 Modeling access restrictions

Some models are primarily concerned with the sensitivity of information, the clearance of subjects and the rules to determine which users are allowed to access which entities in the system. Usually, being "cleared" high enough, is not sufficient to access an entity; the subject must have a valid reason to access the entity before being allowed to do so.

In a computer system the sensitivity levels of entities and clearance levels of subjects are often integers; if the clearance level of a subject dominates the sensitivity level of an entity, the subject is "cleared high enough" to access the entity.

The second requirement before being allowed to access the entity, is that the subject must have a valid reason for accessing the entity; this requirement is referred to as the ***need to know*** requirement. For this reason one or more "categories" is associated with every subject and every entity. Subjects are associated with all those categories that they need to access in their day to day activities. Entities are associated with one or more categories to indicate the reason(s) why the information appears in the system. A subject is then only allowed to access an entity if, both

- The subject's clearance level dominates the entity's sensitivity level; and

- The subject is authorised to obtain information about all the categories that are listed with the protected entity.

If this is the case, the clearance of the subject is said to dominate the sensitivity of the entity.

A set of subjects and entities conforming to these rules form a mathematical structure known as a *lattice*: a lattice is a set with a partial ordering ($\geq$) such that any two elements have a greatest lower bound and a least upper bound. For this reason the described security models are known as *lattice models of security*. In chapter 5 we look at these models again and, specifically, at the military security model, which is one example of a lattice model.

2.3.2 Information flow models for secrecy

An authorised subject may read sensitive information and then write it in a location where an unauthorised subject may access it. In order to restrict such a "flow of information" a number of models have been proposed, the best known being the Bell-LaPadula model [Bel76].

The Bell-LaPadula model is based on two properties: the *simple security property* and the **-property*. Every subject and every entity are assigned to a fixed security class. According to the simple security property, a subject is allowed to read information from an entity if the security class of the subject dominates that of the entity. According to the *-property, if a subject has read access to an entity, it does not have write access to entities at lower security classes. Informally, a subject may not 'read up' (simple security property) and may not 'write down' (*-property).

The security classes for subjects and entities may be based on the lattice model; however, any model where it is possible to assign "security classes" such that a partial ordering exists, may be used.

Information flow models for secrecy is one of the main topics of this thesis.

2.3.3 Information flow models for integrity

Similar to the models that maintain secrecy despite the fact that information flows from one location to another, some models maintain integrity in the system. These models restrict the extent to which unreliable information will flow through the system. The Biba model [Bib77] is one example.

In the Biba model an integrity level is associated with every subject and every entity. The integrity level of an entity indicates the trustworthiness or reliability of the information contained in the entity. The *simple integrity property* states that a subject may only read information if the entity's integrity level dominates that of the subject. The ***integrity *-property*** states that a subject may only write information if the subject does not have read access to entities with a lower integrity level than the entity to be written. Together these properties ensure that unreliable information will not 'contaminate' reliable information and subjects.

We do not address integrity explicitly in this work. However, note that many of the models we mention for secrecy do have integrity benefits; a model that only allows trusted subjects to access a protected entity in specific ways partially addresses the integrity problem.

2.3.4 Modeling the propagation of access rights

A number of models has been proposed that model the propagation of access rights. These models typically use mathematical principles to represent mechanisms that may be used on a computer system to protect entities; examples of such mechanisms are capabilities and access control lists (see chapter 5). The model also contains mathematical rules that correspond with the operations a user may invoke (such as creating new entities and transferring access rights for such a new entity to other subjects). The mathematical representations are then used to answer questions such as whether a given subject may ever obtain access rights for a given entity.

Examples of models in this category include the ***Harrison-Ruzzo-Ullman** model* [Har76] and the ***take-grant*** system [Sny81].

2.4 Conclusion

This brief overview only touched on some aspects of security. Aspects such as encryption, secure operating systems, network security, etc, all play an important role in secure databases. Since understanding the rest of this work does not depend on a knowledge of these topics, we do not discuss them at this time.

Chapter 3

The Object-oriented Paradigm

Object-orientation is the second cornerstone of this work; this chapter is an introduction to the object-oriented paradigm. The other two cornerstones are computer security theory (see chapter 2) and database theory (see chapter 4).

Object-orientation is a software development methodology characterised by the use of ***objects:*** an object is a unit encapsulating both the data and the procedures necessary to model an entity in the real world (or other problem space).

Much of the power of the object-oriented paradigm comes from its inheritance facility: new object definitions can inherit (reuse) aspects from existing object definitions, just adding the lacking aspects. This obviates the need to give an entire definition for every different object.

I inherited it brick and left it marble.

Emperor Augustus
Divus Augustus, 28

3.1 Introduction

Software systems model aspects of the real world. A software development paradigm serves as a guide for the modeling process. Object-orientation is such a paradigm: it provides a set of concepts that are used to construct a software model. During the object-oriented software design process aspects from the real world that have to be included in the model are identified; representatives for these aspects are then constructed with the aid of these object-oriented concepts.

Object-orientation originated in the programming language field, with Smalltalk [Gol83] as the best known object-oriented programming language. Since then it has seen active service in the database, artificial intelligence, systems analysis and a variety of other fields.

3.2 Object-orientation: concepts

Object-orientation is a relatively new concept and experts do not fully agree about its exact composition. Adherents of the different object-oriented philosophies also often use dissimilar terminology, adding to the confusion. We will adopt the philosophy and terminology as used in the Smalltalk language [Gol83]. This philosophy is the original and currently predominant one. Also, most of what will be said about this viewpoint can be translated into the language of the different viewpoints without much trouble. The interested reader is referred to [Shr88] for descriptions of alternative views on object-orientation.

The basic components of an object-oriented system are

- Objects;
- Methods;
- Messages;
- Classes; and
- Inheritance via a class hierarchy.

We will briefly describe each of these components.

An *object* is a unit consisting of both the data (values) and code (procedures) to manipulate the data; all values in the system are objects; an object may be assigned to a variable or passed as a parameter to a procedure. A procedure forming part of an object (the 'code to manipulate the data') is known as a *method*. The object is encapsulated—the only way to access it is by executing one of its methods. As an example, consider a stack object. This object will have some data structure (probably a linked list or an array) to hold the values currently on the stack. Of course this data structure will be hidden from the user. The stack object will support a method to push a value onto the stack (say PUSH), a method to pop a value from the stack (say POP) and one to determine whether the stack is empty (say STACKEMPTY). Anyone wishing to use this object will do it by 'calling' the relevant method.

Messages are the means objects use to communicate with each other: if an object wants to push something onto the stack described in the previous paragraph, it will send the message PUSH together with the value to be pushed to the stack object. To retrieve a value from the stack, an object will send the message POP to the stack object, which will send the top value to the sender of the POP message.

Methods consist of a sequence of 'actions', where every 'action' is again a message to an object. When an object receives a message, the relevant method will be activated. This method will perform its required function by sending messages to other objects. This is similar to a procedure in a conventional (procedural) programming language, which consists of calls to other procedures and functions. Some methods—the so-called primitive methods—are not implemented as messages to other objects, but is executed directly by the hardware: addition of integers is an example. As an example, the PUSH method of the stack object may send a message to a stack pointer, requesting it to increment itself; then send the stack pointer and the value to be pushed to an array object, requesting the array to insert the value at the position indexed by the stack pointer.

Rather than repeatedly describing an identical object if multiple copies of that object are required, a *class* is described once and is used as a 'template' to instantiate (or create) the identical objects. In fact, in the object-oriented philosophy we are describing, an object is never directly defined: a class is always defined and the object(s) are instantiated by sending a message to the class. For example, the stack object described in previous paragraphs will typically be instantiated from a stack class that has been defined specially, or that was already available in the system. Alternatively, a class is sometimes viewed as a collection of similar objects.

A class description consists of a description of the methods and variables which will make up objects of that class. There are two categories of variables: instance variables and class variables. Whenever an object is instantiated it receives its own set of instance variables; if an object modifies one of its instance variables, it does not have any direct effect on any other object of that class (or any other class). Class variables, however, are shared by all objects belonging to a specific class; if an object modifies a class variable, all subsequent references by any object of that class will use the modified value.

Often objects are alike but not identical. In such a case *inheritance* is used to 'inherit' the similar parts rather than redefining those parts repeatedly. Usually the common aspects of a number of classes are identified (or 'abstracted') and a class is defined based on those common aspects. *Subclasses* may now be defined based on the class of common aspects: all the common aspects are inherited and only the additional requirements are defined. It is also possible to redefine any inherited aspect. The class on which the subclasses are based are known as their *superclass*. As an example, suppose we want to keep information about students and lecturers. It is possible to immediately define classes describing both, but it is also possible to describe a class containing the common elements like name, age, identity number, etc. A subclass can now be defined for students that inherits the common elements and adds the student details such as study record, degree registered for, etc. Similarly a subclass can be defined for lecturers adding their specific requirements such as courses currently taught, publication record, etc.

The Object-oriented Database System Manifesto [Atk89,Atw90] gives the following eight rules for a system to be considered "object-oriented":

1. Complex objects must be supported, ie one must be able to build objects from other objects;

2. Objects should have an identity independent of the value of such an object—this makes it possible for other objects to share a single instance of a specific object: all these objects refer to the shared object using this object's *identity*;

3. Objects should be encapsulated;

4. Types or classes (or both) should be supported: the *Manifesto* considers a type as a mechanism which is used to statically describe the composition of an object, similar to data types in languages such as Pascal; classes are objects able to create instances at run-time of the objects they represent;

5. Inheritance should be supported;

6. *Late binding* must be used to associate a message with the method in the target object: this allows objects to be created which will manipulate other objects, where the type (or class) of the objects to be manipulated is not fixed;

7. Any computable function must be expressible in the language; and
8. The system must be extensible: a user-defined type (or class) should behave exactly like a system-defined type.

3.3 Object-oriented software design

It is important to remember that a system supporting the described components and rules is not automatically "object-oriented"—these components have to play the central role from design through to implementation before a system can be considered "object-oriented". An object-oriented design process will identify the objects involved in a system. These objects will then each be analysed and decomposed into smaller objects. This process will continue until the required objects are simple enough to be primitive objects directly supported by the system.

Libraries of objects for particular applications or environments are often available off the shelf or, otherwise, custom built. These libraries allow the designer and implementer to deal with objects which reflect the real-life objects they are modeling in their systems—ie it allows them to work at a higher level of abstraction.

3.4 Related paradigms

Wegner[Weg88] distinguishes between *object-based*, *class-based* and *object-oriented* systems. An *object-based* system is a system that supports the concept of objects. A *class-based* system is an object-based system which additionally supports the class concept, where every object in the system is an instance of some class in the system. An object-oriented system is a class-based system that also supports inheritance from a superclass to a subclass.

A system that supports abstract data types thus is a class-based language.

The reader may consult [Weg90], [Shr88] or [Kim89] for a detailed exposition of the object-oriented paradigm—the first reference is a paper giving a detailed overview of the field, while the latter two are collections of papers addressing various aspects of the field.

Chapter 4

Databases, Object-orientation and Security

This chapter briefly discusses database theory, concluding our overviews of the underlying theory on which this work is based. A database is a computerised repository for information typically accessed by a number users through a diverse set of application programs.

Special attention is given to object-oriented database systems; in other words, databases that use the object-oriented model (see chapter 3) to represent information.

This chapter further discusses secure databases, drawing on computer security as discussed in chapter 2.

Secure object-oriented databases are discussed in a later chapter.

I only ask for information.

Charles Dickens
David Copperfield, ch.20

4.1 Introduction

This chapter first discusses databases, then object-oriented databases and, lastly, security in databases. Security in object-oriented databases is discussed in following chapters. No original work is presented here, but rather a summary of relevant areas from the database field.

4.2 Databases

The following is quoted from Date [Dat90]:

> A *database* consists of some collection of persistent data that is used by the application systems of some given enterprise (p 10).
>
> The term "entity" is widely used in database circles to mean any distinguishable object that is to be represented in the database (p 11). It is important to understand that, in addition to the basic entities themselves, there will also be *relationships* linking those entities together (p 11).
>
> ... entities (and hence relationships also) have *properties*. For example, suppliers have *locations*; parts have *weights*; projects have *priorities*; assignments have *start dates*; and so on. Such properties must therefore be represented in the database also (p 13).

The Object-Oriented Database System Manifesto (see page 19) lists five rules a system must satisfy before it can be considered a database:

1. Data (of any type) stored must implicitly be stored persistently;
2. The system must be able (and optimised) to handle very large volumes of data;
3. Multiple users should be able to access the database concurrently;
4. The system should be able to recover from hardware and software failures; and
5. It should be easy to formulate and execute ad hoc queries efficiently.

Databases developed from files when files no longer adequately addressed the needs of users. Change of user needs over the years have caused a variety of database models. These models are usually categorised into four generations.

The first generation consists of hierarchical databases. These databases are organised as inverted trees—any parent node can have one or more child nodes, but a child node may not have more than one parent node. Application programs navigate through the database by traversing the tree.

The second generation is the 'network' databases. Relationships between the various records are not limited to a single parent related to one or more children. One-to-one, one-to-many, many-to-one and many-to-many relationships are allowed. Application programs navigate through the database by following the links relating the applicable records.

The third generation databases are based on the relational model. Data is stored as tuples in tables. Application programs retrieve data by selecting the appropriate values from these tables. Several tables can be joined together over common columns to retrieve information not directly represented in a table.

The fourth generation database model is the object-oriented model. In this model code (or procedures, or methods) is stored with the data—that is the stored information includes the 'intelligence'. Data is not accessed directly by the application program—the application program sends a message to the relevant data in the database, which processes the message internally and then returns the requested information. The relationships between data can be brought about by building more complex entities from simpler ones, or by sending messages between separate entities. More information on object-oriented databases is given in the next section.

Many textbooks have been written about databases, including [Dat90]. The interested reader is referred to one of them for details.

4.3 Object-oriented databases

Object-oriented databases behave according to the general object-oriented paradigm described in a previous chapter: such a database consists of persistent "intelligent" objects, which may be accessed via the methods each provides.

Object-oriented databases look especially attractive in new application areas such as Computer Aided Design and Manufacturing (CAD/CAM) and Computer Aided Software Engineering (CASE) [Dat90, p 684]. However, as Date [Dat90, p 684] points out, while application programs are intended to solve specific problems, databases are (by definition) intended to solve a variety of different problems; the success of the object-oriented paradigm for writing application programs will therefore not necessarily extend to databases. See [Dat90, pp 683–707] for a critical discussion of object-oriented databases and, in particular, pp 700–704 for a list of problem areas remaining (or inherent) in the object-oriented database model.

One of the primary players in the database field, IBM, is basing their strategy on the relational model, with possible object-oriented extensions. Quoting Davis [Dav89]:

> One of IBM's basic concerns about the object-oriented approach is that it is not based on a rigorously defined model. The relational model, on the other hand, is built on a sound, extensible mathematical model. IBM acknowledges the utility of the object-oriented approach and its ability to deal with certain applications that are beyond the scope of current relational systems (particularly in the area of performance). However, the jury is still out on the basic question: Is it possible to extend the relational model to incorporate object-oriented capabilities and get good performance, or is it necessary to build a system from the ground up to gain the benefits of object orientation? Definitely not clear, according to IBM. Since IBM is basing its strategy on the relational model, we expect to see object-oriented extensions to IBM's relational DBMSs.

New commercial object-oriented databases are presently announced regularly; the following were (amongst others) released or upgraded during 1990:

- Objectstore release 1.0 from Object Design Inc for Sun Microsystems 3 and SPARCstation platforms
- Ontos release 2.0 from Ontologic (replaced an earlier product called Vbase) for Sun-3, Sun-4, SPARCstations, Apollo and DECstation workstations
- Gemstone version 2.0 from Servio Logic Corporation for Sun workstations, Digital VAX/VMS and DECstations under Ultrix
- Versant ODBMS from Versant Object Technology Corporation for Sun workstations
- Itasca (formerly Artemis, derived from MCC's Orion) from Itasca Systems for Apollo, Hewlett-Packard, Silicon Graphics, Sun and Avion workstations

See [Aye91] for a discussion of the development of one such object-oriented database.

The Object-Oriented Database System Manifesto (see page 19) lists five areas where object-oriented database systems may differ and that may be used as a basis for comparison between different offerings:

1. Is multiple inheritance supported or not?

2. Are types checked during compile-time (ie when a new class description is added to the database)? Is type inferencing supported, ie is it possible for the compiler to deduce the (derived) type of a combined collection of objects based on usage?

3. Is it possible to distribute the database over a number of platforms?

4. Does it support "design transactions", ie transactions that cannot be treated as classical serialisable "business transactions"?

5. Is it possible to maintain different versions of the same objects simultaneously?

Note that a simple *yes/no* answer to these questions is not sufficient: for instance, if multiple inheritance is supported, one can further distinguish between the various implementations of multiple inheritance.

See [Kim91] for a discussion of the strengths and weaknesses of object-oriented databases.

4.4 Security in databases

4.4.1 Secrecy, integrity and availability

Security is concerned with secrecy (or confidentiality), integrity and availability. In a database environment, secrecy refers to measures taken to ensure that information is not disclosed to unauthorised parties; integrity to the prevention of unauthorised changes of (including additions to) the database; and availability to the assurance that data is available to authorised users when they need it.

Denning [Den88, pp 1–2] lists the following components of a security policy:

- *A discretionary access control policy*, stating which operations specific users may perform on which data and, also, how a user may (at the user's discretion) transfer such access rights to other users.

- *A mandatory access control policy*, to ensure that users do not obtain direct or indirect access to data they are not cleared to access. Mandatory access control is of concern when dealing with multilevel databases, which we will address shortly.
- *A statistical inference policy*, specifying to what extent it will be possible to draw conclusions from statistical summaries obtained from a database.
- *A consistency policy*, defining the states in which the database is considered consistent (ie valid or correct).
- *An identification/authentication policy*, specifying measures to ensure that unauthorised users of the database do not masquerade as legitimate users.
- *An audit policy*, specifying how records will be kept of database operations for future reference.

Not all of these policies have to be present in a given organisation; also, some of the requirements may be dealt with by the operating system. On the research front, most of the effort has been spent on the first three issues.

Pfleeger [Pfl89, p 304] further distinguishes between *physical database integrity*, *logical database integrity* and *element integrity*. Physical database integrity must ensure that the database is immune to (or reconstructable after) physical problems, such as power failures. Logical integrity is concerned with the internal logical structure of the database, for instance ensuring that if data in some part of the database refers to other data, the other data does exist somewhere in the database. Element integrity ensures that the data contained in every element is accurate by, for example, ensuring that such values are between acceptable limits.

We will limit our attention in subsequent sections and chapters to measures taken to ensure the secrecy of protected information and to prevent unauthorised modification of information.

4.4.2 Multilevel databases

A multilevel secure system contains information of varying sensitivity; multiple users may access the system simultaneously, but any user is only allowed to access information which that specific user has been cleared to access. According to [Lun90b, p 593]

> The concern for multilevel security arises when a computer system contains information with a variety of classifications and

> have some users who are not cleared for the highest classification of data contained in the system.
>
> A security classification, or *access class*, consists of a hierarchical sensitivity level (eg TOP-SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED, etc) and a set of non-hierarchical categories. In order for a user to be granted access to information, the user must be cleared for the sensitivity level as well as for each of the categories in the information's access class.

Multilevel security is not only applicable to databases; however, in this work we will restrict our attention to multilevel secure (MLS) databases.

In a multilevel secure system, special care is taken that an authorised user does not write information in a location where it can be accessed by a user who was not cleared to access the information in its original location. As an example, using the familiar military terminology, a user with a SECRET clearance is not allowed to access information classified as TOP-SECRET; a user with a TOP-SECRET clearance may read information classified as TOP-SECRET but is not allowed to write the information in a location that may be accessed by a user with a SECRET clearance.

Security in a multilevel database may be compromised if a user is able to determine that a certain attribute exists in the database even if the user cannot determine the actual value stored for that attribute. Suppose a user tries to write an attribute value that already exists, but where the value is classified too high for this particular user to read. If the update request is rejected, the user may infer that such a value already exists. If the update is allowed, the existing (highly classified) information may be overwritten. A common solution for this dilemma is to allow more than one entry in the database for the same information; one entry for every classification level; an update request will then neither be rejected, nor interfere with higher classified processes. This technique is known as polyinstantiation. On the negative side, polyinstantiation places a bigger burden on the database management system, because multiple entries for the same information has to be maintained. Polyinstantiation may also cause consistency problems because the wrong entry may be retrieved for a given query.

The four stages in the development of a secure system as given by Pfleeger[Pfl89, p 242–243] are:

- Modeling;
- Design;
- Ensuring trust; and

- Implementation of the secure system.

In chapter 6 we propose a new theoretical model for secure object-oriented databases, which we have called SECDB. See chapter 7 for references to other examples of such models.

Much work has been done on implementation strategies (ie the second implementation stage according to Pfleeger) for multilevel secure databases; possible strategies for such databases include the following:

- The *trusted filter* or *trusted front-end* approach, where a (trusted) filter is placed between application programs and the database. It is the responsibility of this filter to ensure that no unauthorised information is disclosed. The mechanism often works by attaching fields containing the classification level and a cryptographic checksum to any stored data. The filter uses the classification level to check whether a given user should be allowed access. The cryptographic checksum is used to check that nothing has tampered with the record without going through the filter. Because the checksum acts as a lock, this mechanism is also often called the *integrity lock* approach.

- The *balanced assurance* approach, where responsibility is shared between a trusted operating system and the database system—here the operating system acts as a trusted back-end. This approach is also known as the *decomposed database architecture.* Database entities are split into parts which are then stored in single level operating system "containers", which are protected by the operating system. As an example, a multilevel relation may be reorganised into a set of single level relations; every relation may then be stored in a file, which is protected (at the relevant level) by the operating system.

- The *monolithic* or *uniform assurance* approach. Using this approach, the database is implemented to take sole responsibility for security of information stored in the database.

With the balanced assurance and the monolithic approaches, *views* or *windows* may be used to present the user with only that information he is entitled to access. See Laferriere [Laf90], Denning [Den88, pp 11–14] and Pfleeger [Pfl89, pp 331–340] for a discussion of the relative merits of each of these (and some related) approaches. Implementation strategies fall outside the scope of this work.

4.4.3 Statistical inference

Statistical inference refers to the possibility that it may be possible to draw conclusions from statistical summaries obtained from a database. In general, an *inference* problem may occur whenever data is only partially disclosed and it is possible to infer (undisclosed) sensitive information from the partially disclosed data. Although it is outside the scope of this research, we will briefly touch on the problem and mention some techniques that may be considered as solutions.

The problem occurs when a user is not allowed to access individual values for a given attribute, but needs some statistical summary values for that attribute. As an example, a given user may not be allowed to view salary entries for individual employees, but may need to know the average salary for certain categories of employees. If this user is allowed to get the "average" salary of the one employee in the "managing director" employee category, this user can easily "infer" the salary of the managing director. Less accurate, but close, conclusions may be made about the salary if the group is bigger than one, but still relatively small. One possible solution thus is to restrict the size to some minimum number of entries for any statistical value to be computed: if less than this minimum number of entries will take part in the computation, the computation is simply not performed. Examples of summary information that may be used in such an attack include the sum, count, average and mean of data conforming to a specified set of constraints.

To worsen the problem, a user may be able to draw conclusions on the basis of a number of queries. This happens when some relationship exists between various attributes, in which case the user may be able to algebraically manipulate returned non-sensitive data to calculate sensitive values. To further worsen the problem, such queries may be executed over a period of days. This means that it may be necessary for the system to maintain a list of queries executed by any given user, to be able to determine what the user already knows, and thus what the user would be able to infer given the next query from the user—if the user will be able to infer sensitive information, the query may again be abandoned. However, this still does not solve the problem if two users cooperate—with their combined knowledge they may be able to make a sensitive inference, without any possibility of the system noticing. And, of course, maintaining a list for every user of what that user already knows is very expensive.

In the case of very sensitive data, *perturbation* is often used. Perturbation refers to the deliberate distortion of information released by the database. The information is distorted in such a way that the necessary statistical calculations will still be correct (if calculated over a large enough population). However, obtaining individual values will be useless.

According to Pfleeger [Pfl89, p 27] "there are no perfect solutions to the inference problem". The three possibilities to consider are

1. Do not release any sensitive information, not even statistical summaries of such information;

2. Track what the user knows and do not release any further information if it will compromise security; or

3. Use perturbation to disguise the data.

Inference is covered in more detail by Denning [Den88, pp 14–19] and Pfleeger [Pfl89, pp 319–327].

The *aggregation* problem is related to the inference problem: often the subparts of an entity are not as sensitive as the complete entity. If a user is not authorised to access the complete entity, but is allowed to access the parts it may be possible for that user to combine (aggregate) the parts and obtain the sensitive information. Object-oriented databases often provide a very natural solution for the aggregation problem. We return to it in a later chapter.

4.5 Security in object-oriented databases

A few models have been proposed to deal with security in object-oriented databases. In chapter 6 we propose a new model, SECDB, for this purpose. A taxonomy for such models is developed later in this work—see chapter 7. Examples of other models for secure object-oriented databases are given there.

Chapter 5

The Path Context Model

Most security models do not realistically reflect the complexity of current computer systems. The Path Context Model (PCM) is a recent formal security model attempting to solve this problem. For any request, PCM notes all entities (software and hardware) encountered along the access path of the request; before the target entity is accessed, this collected information about all involved entities is considered to determine whether the access should be allowed or not.

PCM has not yet been defined precisely. This chapter first gives a formal definition of PCM. It is then shown that it is difficult to protect composite objects—objects consisting of other, less complex objects—with PCM. This problem can be solved by modifying PCM so that every level of such a composite object can do the access checks relevant to that level of the object. We illustrate this by defining an object-based version of PCM.

The object-based version of PCM uses object-oriented concepts discussed in chapter 3. The object-based version of PCM will be used in chapter 6 to define a new model for secure object-oriented databases; this model, together with other models for secure object-oriented databases, will be used in chapters 7 to 11 as a basis for a taxonomy for such models.

Should auld acquaintance be forgot,
And never brought to mind?

Robert Burns
Auld Lang Syne

5.1 Introduction

The Path Context Model (PCM) is a recent model for information security proposed by Boshoff and Von Solms [Bos89a,Bos89b,Bos90]. This chapter formalises the definition of the model given by the proposers. It is then shown that it is difficult to use PCM to protect composite entities—entities constructed from other (lower level) entities. This problem can be solved by modifying PCM so that every level of such a composite entity can do the access checks relevant to that level of the entity. This is illustrated in the last section of the chapter, where an object-based version of PCM is defined. This version of PCM will be used in the next chapter to propose the SECDB model for secure object-oriented databases.

5.2 Security background

We encountered security models briefly in chapter 2. In this section we revisit some aspects of such models to serve as background for the description of PCM.

5.2.1 Security in the human environment

Traditionally secrecy measures are based on a classification level and a clearance level: classified material is given a classification level based on the material's sensitivity; people are given clearance levels based on their trustworthiness and function in the organisation. If someone's clearance level dominates the classification level of classified information, that person is allowed to see the classified material. Additionally it is often required that someone must have a valid reason to access information before such a person may see it; this requirement is often referred to as the *need to know* requirement. Security restrictions therefore usually form a partially ordered structure known as a lattice.

Clearance and classification levels are typically named and ordered as follows:

$$\textit{Restricted} < \textit{Confidential} < \textit{Secret} < \textit{Top secret}$$

Further, specific projects (or compartments) are named and individuals and groups who *need to know* about those projects are identified. To determine whether someone should be allowed to see classified material it is necessary to determine whether

- That person's clearance level dominates the classification level of the material; and

- That person is a member of a group who may access information about the project.

Both conditions must be satisfied before allowing access to the material.

5.2.2 Traditional security in the computer environment

Security in the computer environment is traditionally based on the (human) model described above. Information is classified by assigning a classification level—often an integer. Users are cleared by being assigned a clearance level—again an integer. If a user wants to access information, the user's clearance level must be higher than (or the same as) the classification level of the information.

The requirement that a user may only access information he needs to access is often enforced by protecting individual resources by so-called ***discretionary access controls***. In the operating systems environment this is often accomplished by access control lists or capabilities. Access control lists are associated with protected entities in the system and contain a list of users who may access the entity. The owner of the protected entity may grant access by inserting a user's name into the list and revoke access by removing a user's name from the list. Capabilities are non-forgeable identifiers, giving the owner of the capability the right to access a protected entity. The owner of the entity may give a user a copy of the capability, which this user may then use to access the entity or pass on to another user, who may then access it. On a higher level passwords are often used similar to capabilities.

The primary differences between security in the human and computer environments are:

- It is much easier to copy information (often inadvertently) in the computer environment. This copied information may then not be protected, compromising security.

- The reader of information in the computer environment often has no indication of the identity of the writer, making it difficult (or impossible) for the reader to verify the validity of the information. The fact that the writer had an adequate clearance level to write the information in the first place is usually the only guarantee of validity.

- Greater scale in the computer environment. Things happen quicker, more often, more concurrently and on behalf of more users. This requires a finer granularity of classification and clearance levels, increases the possibility of covert channels and makes it less likely that security breaches will be noticed by someone.

- The new technology used in the computer environment offers new possible ways to breach security: electromagnetic radiation from computers, but especially from communication lines, the possibility to read disks, even after information has been deleted, amongst others, all require attention.

Models for computer security usually address the basic principles and some of the differences mentioned earlier. Clearance levels are usually assigned to subjects—subjects are the active entities in the system, usually processes acting on behalf of users. Classification levels are assigned to entities—entities are the static items in the system such as files, printers and other resources. Rules then determine whether a subject with a given clearance level may access an entity with a given classification level. Different rules may exist for reading and writing information.

5.2.3 PCM—another security model

The Path Context Model [Bos89a,Bos89b,Bos90] uses a different approach to the security problem. Simply put, whenever a request to access some resource is issued, information is collected about this request's progress through the system until the request reaches the resource to be accessed. The collected information (known as 'baggage') will include items such as

- The identity of the user who initiated the request;
- All software packages involved in the processing of the request;
- Networks used to transmit the request; and
- Computer system on which the request was initiated and all other computer systems involved.

A profile is described for protected resources. Such a profile specifies items which must be in the baggage and items which are not allowed to be in the baggage of a request, for the request to be serviced by the resource. The profile may, for instance, specify that only users *X*, *Y* and *Z* may access the resource, that the request may only be initiated from a computer system *C* and that the request may not be routed via an untrusted communications line *L*.

A validator is used to compare the actual collected baggage to the profile specification, and decide whether the request should be serviced.

We give a formal description of PCM in the next section.

5.3 The Path Context Model (PCM)

This section describes the Path Context Model formally. Firstly, we define notation and terminology, which is then used to define the concept of baggage and, after that, security profiles.

Our terminology and notation differs slightly from the original description of PCM [Bos89a,Bos89b] to facilitate a mathematically more rigorous approach. In accordance with [Bos89a,Bos89b] we use the term *object* in this section to describe a *static*, protectable entity.

5.3.1 Notation and terminology

The following notation is used in describing PCM:

$$\begin{aligned}
E &= D \times S \times I \times O \times A \\
D &= \{d \mid d \text{ is any valid domain}\} \times K \\
S &= \{s \mid s \text{ is any valid software component}\} \times K \\
I &= \{i \mid i \text{ is any valid integrity state}\} \times K \\
O &= \{o \mid o \text{ is any valid object}\} \times K \\
A &= \{a \mid a \text{ is any valid accessor}\} \times K \\
K &= \{k \mid k \text{ is any valid access class}\}
\end{aligned}$$

The examples given in [Bos89b] of this notation, are as given in table 5.1.

PCM is based on three concepts:

1. The baggage collection vehicle (BCV), used to collect information about the access request;
2. A profile for describing access restrictions to resources; and
3. A validator which will, given the baggage and the profile, determine whether a resource may be accessed or not.

In our formal treatment, we will view these components as additional levels of a security system: the baggage collection vehicle forms the basic layer, the profile includes the baggage collection vehicle, but adds the functionality to describe access restrictions, while the validator includes both

Element	Example excluding Cartesian Product with K
D	LAN, WAN, VAN
S	Network software, teleprocessing software, DBMS
I	Problem state, supervisor state, MESAS (multiple executions in a single address space), encrypted transmission
O	Program, file, block of data, data element
A	User-transformed accessor
K	Read, execute, write, delete, passthru, pre-access checking, post-access checking

Table 5.1: Notation used to describe PCM

the baggage collection vehicle and the profile, adding the access enforcing capability.

5.3.2 The baggage collection vehicle

The baggage collection vehicle is the mechanism responsible for the collection of baggage. It is described as a formal grammar, with productions specifying the format of the baggage. This grammar maps all changes in the physical environment to the baggage string generated for the request. Application of this grammar will be discussed after formally defining it.

Definition 5.1 *The baggage collection vehicle is a formal grammar*

$$C = (V_N, V_T, P, \Sigma)$$

where V_N is the set of non-terminal symbols, with

$$V_N = \{\Sigma; \mathcal{A}; \mathcal{D}; \mathcal{S}; \mathcal{I}; \mathcal{O}; \mathcal{C}\}$$

V_T is the set of terminal symbols, with

$$V_T = A \cup D \cup S \cup I \cup O$$

$\Sigma \in V_N$ is the starting symbol.
P is the following set of productions:

$$\mathcal{A} \rightarrow a_1|a_2|a_3|\ldots|a_k|\varepsilon$$

$$\begin{array}{lcl}
\mathcal{D} & \rightarrow & d_1|d_2|d_3|...|d_m|\varepsilon \\
\mathcal{S} & \rightarrow & s_1|s_2|s_3|...|s_n|\varepsilon \\
\mathcal{I} & \rightarrow & i_1|i_2|i_3|...|i_p|\varepsilon \\
\mathcal{O} & \rightarrow & o_1|o_2|o_3|...|o_r|\varepsilon \\
\mathcal{C} & \rightarrow & \mathcal{ADSIC} \\
\mathcal{C} & \rightarrow & \mathcal{O} \\
\Sigma & \rightarrow & \mathcal{C}
\end{array}$$

where

$$\begin{array}{lcl}
a_j & \in & A, 1 \leq j \leq k \\
d_j & \in & D, 1 \leq j \leq m \\
s_j & \in & S, 1 \leq j \leq n \\
i_j & \in & I, 1 \leq j \leq p \\
o_j & \in & O, 1 \leq j \leq r
\end{array}$$

The value ε is used whenever a value is not defined, cannot be determined or does not make sense in the specific context. □

Equivalently, the baggage can be described by

$$\mathcal{C} \rightarrow (\mathcal{ADSI})^{+}\mathcal{O}$$

where the plus denotes one or more repetitions of the preceding symbol.

A baggage string thus consists of one or more 4-tuples, v_j, followed by the identity of the object being accessed, with

$$v_j \in ADSI = \{adsi | a \in A \wedge d \in D \wedge s \in S \ \wedge i \in I\}$$

Such a v_j is known as a baggage vector. A baggage string is thus of the form

$$v_1 v_2 v_3 ... v_n o$$

Here v_1 represents the initial values for the accessor, domain, software and integrity state. Note that the value of such a baggage vector is defined the whole time from initiation of the request until the request has been serviced. Whenever the value of v changes it is appended to the baggage string, ie for any $1 \leq j < n, v_j \neq v_{j+1}$, but in the time between assuming values v_j and v_{j+1} no other values are assumed.

5.3.3 The profile

A security profile is a formal grammar designed with the following in mind:

- Productions of the grammar correspond to possible variations along the path of an access request. Examples of such variations are when the integrity state of the computer changes from *problem state* to *supervisor state* or when the request is transferred from one domain to another. The derivation process starts, as usual, with the starting symbol and productions are applied until the string only consists of terminal symbols. When one of these variations along the access path occurs it will be accompanied by the application of one or more productions in the derivation process. The end result of the derivation process is a string reflecting the access path from the user to the accessed object, including information on accessors, domains, software and integrity states along the way, similar to the baggage string described previously.

- The productions of the grammar are designed that a terminal string will only be reached at the end of the derivation process if a valid access path has been followed. In other words, if, at the point where the requested object is to be accessed, the derived string only consists of terminal symbols, access is granted to the object; if any non-terminal symbols remain which cannot be replaced by terminal symbols, access will be denied.

- Productions will usually be included to describe all (potential) access paths. However, the use of some productions will be restricted based on the access path followed thus far. This will be done with the allowing and forbidding contexts of random context grammars [Van70]. Such contexts are sets of symbols from the alphabet of the grammar. In order to apply a production with an allowing context, all symbols in the allowing context must appear somewhere in the sentential string derived thus far. To apply a production with a forbidding context, no symbol from the forbidding context is allowed to appear in the sentential string derived thus far. In practical terms, a production may have an allowing context containing the name of a specific user—only if that user was involved in the request will the derivation process be allowed to proceed by using this production. Similarly, a production may have a forbidding context containing the identity of some untrusted piece of software—if that software was involved along the access path of a request, the derivation process will not be allowed to continue by applying this production.

More formally, a security profile is an extended path context grammar [Bos89b], which is based on random context grammars. The primary difference between random context grammars and extended path context grammars is the fact that the restrictions associated with productions in the latter may also require that symbols be adjacent and/or in a specific order. This will be clear after the formal definition of a profile has been given.

The productions in the path context grammar correspond largely to the productions given earlier for the baggage collection vehicle. The important difference between those productions and the current ones is the fact that a non-terminal symbol, Ω_j, is introduced for every object, o_j, in the system. Productions with allowing and forbidding contexts are then included, allowing this non-terminal to be replaced by the terminal symbol representing the object; these contexts are used to specify security constraints affecting the concerned object. We return to this topic after defining a security profile formally.

Definition 5.2 *A security profile is an extended path context grammar*

$$C' = (V_N, V_T, P, \Sigma)$$

V_N is the set of non-terminal symbols, with

$$V_N = \{\Sigma; \mathcal{A}; \mathcal{D}; \mathcal{S}; \mathcal{I}; \mathcal{O}; \mathcal{C}\} \cup \{\Omega_j | o_j \in O\}$$

V_T is the set of terminal symbols, with

$$V_T = A \cup D \cup S \cup I \cup O$$

Let V represent the alphabet, ie

$$V = V_T \cup V_N$$

$\Sigma \in V_N$ is the starting symbol.
P is a set of productions of the form

$$\begin{array}{lll}
\Sigma & \rightarrow & \mathcal{C} \\
\mathcal{A} & \rightarrow & a_j, \text{where}\, a_j \in A \\
\mathcal{D} & \rightarrow & d_j, \text{where}\, d_j \in D \\
\mathcal{S} & \rightarrow & s_j, \text{where}\, s_j \in S \\
\mathcal{I} & \rightarrow & i_j, \text{where}\, i_j \in I \\
\mathcal{O} & \rightarrow & \Omega_j, \text{where}\, \Omega_j \in V_N
\end{array}$$

$$\begin{array}{lcl}
\mathcal{A} & \rightarrow & \varepsilon \\
\mathcal{D} & \rightarrow & \varepsilon \\
\mathcal{S} & \rightarrow & \varepsilon \\
\mathcal{I} & \rightarrow & \varepsilon \\
\mathcal{O} & \rightarrow & \varepsilon \\
\mathcal{C} & \rightarrow & \mathcal{ADSIC} \\
\mathcal{C} & \rightarrow & \mathcal{O} \\
\Omega_j & \rightarrow & o_j(B;F), \text{with} \\
 & & B \subseteq V \cup \{\langle x\rangle | x \in V^n, n \geq 2\} \cup \{\langle\langle x\rangle\rangle | x \in V^n, n \geq 2\}, \\
 & & F \subseteq V \cup \{\langle x\rangle | x \in V^n, n \geq 2\} \cup \{\langle\langle x\rangle\rangle | x \in V^n, n \geq 2\}, \text{and} \\
 & & V^n = \{\alpha_1\alpha_2\alpha_3...\alpha_n | \alpha_j \in V, 1 \leq j \leq n\}
\end{array}$$

□

As mentioned, the Ω_j symbols have been introduced as the 'placeholders' for objects; these symbols correspond one-to-one with the objects, o_j, in the system. The B in the production

$$\Omega_j \rightarrow o_j(B;F)$$

is the permitting context and the F the forbidding context. If we have a sentential string

$$\alpha_1\Omega_j\alpha_2 \in V^+$$

then the production

$$\Omega_j \rightarrow o_j(B;F)$$

can be applied resulting in $\alpha_1 o_j \alpha_2$ if

- all single symbol elements of B appear somewhere in $\alpha_1\alpha_2$; for all n-tuples of the form $\langle x_1, x_2, ..., x_n\rangle$, the $x_j, 1 \leq j \leq n$, all appear adjacent (but in any order) to one another in $\alpha_1\alpha_2$ (ie a permutation of $x_1, x_2, ..., x_n$ is a substring of $\alpha_1\alpha_2$); for all n-tuples of the form $\langle\langle x_1, x_2, ..., x_n\rangle\rangle$, $x_1x_2...x_n$ is a substring of $\alpha_1\alpha_2$; and

- no single symbol element of F appears anywhere in $\alpha_1\alpha_2$; for all n-tuples of the form $\langle x_1, x_2, ..., x_n\rangle, 1 \leq j \leq n$, no permutation of $x_1, x_2, ..., x_n$ is a substring of $\alpha_1\alpha_2$; and for all n-tuples of the form $\langle\langle x_1, x_2, ..., x_n\rangle\rangle$, $x_1x_2...x_n$ is not a substring of $\alpha_1\alpha_2$.

If access to object o_j is required, following the access path with productions of the profile will eventually yield a string of the form $\alpha\Omega_j$. Before accessing object o_j, the Ω_j will have to be replaced by o_j (that is $\alpha\Omega_j \Rightarrow \alpha o_j$).

This will only happen if a suitable production of the form

$$\Omega_j \rightarrow o_j(B; F)$$

exists, ie if the access path thus far (represented by α in $\alpha\Omega_j$) did follow the 'paths' described in B, but did not follow any 'paths' described in F.

5.3.4 The validator

The validator is the component mapping a derivation step (using notation defined in the previous section)

$$\alpha\Omega_j \Rightarrow \alpha o_j$$

to access to the physical object o_j.

The complexity of the model is concentrated in the previous two layers (in the BCV and profile) resulting in an extremely simple validator.

5.3.5 A critical look at objects

This section addresses the problems encountered when protecting composite objects with PCM. Remember that objects are traditionally viewed as the passive entities (such as files) in the system which can be manipulated by software, while subjects are active entities. Composite objects are objects which themselves consist of other, lower-level objects.

Let us first consider requests. The notion of requests is treated informally in PCM. Considering the given grammars, it is clear that PCM makes the following assumptions about requests:

- A request is initiated by a *primary accessor*, usually a human;
- A request terminates when it accesses the requested object;
- Zero or more intermediaries (software, hardware, etc) may take part in the request between the primary accessor and the object; and
- Access checks are only performed at the point where the object is to be accessed.

Now, consider objects which consist of other objects: examples are

- Database management systems which use files at a lower level to store information;
- Files which are stored as sectors on disks;

- Any resources managed by a 'server', where the server is then treated as the 'object'; etc.

If we take a file consisting of sectors as an example, it is easy to see that both the file and the sectors must be protected. This leaves us with three possibilities:

1. Protect the sectors. A typical profile constraint for such a sector will check that

   - the access request comes from an authorised user via a valid access path; and
   - the request did go via the file system.

   The problem with this approach is that it is unnatural to do access checking for a higher level object at the low level. The file is the logical object which needs protection—protection measures should be at this level.

2. Only protect the file with PCM and use an alternative mechanism to protect the sectors. This solution is unsatisfactory, because it uses a non-homogeneous approach to protection.

3. Protect the file with PCM and let the file system issue requests to access the sectors. PCM can then be used to check that any access to sectors was originated by the file system. Also this solution is unsatisfactory: the problem PCM addresses is precisely the one caused by 'agents' acting on behalf on users and concealing the identities of the real users.

As an even more problematic example, consider a database consisting of files, which each consists of sectors. In the case of a database, it is often desirable to protect the individual records (at possibly different classification levels). In this case, two logical records, with differing protection requirements, can be stored on the same physical sector, necessitating a very unnatural application of PCM—only possibilities (2) and (3) above are viable with the current definition of PCM.

In the next section we describe a version of PCM adapted for the object-oriented environment, which solves the described problem: simply put, we propose that a single request must be able to travel from the originator to the (high-level) object to be accessed; the access controls necessary at this level can then be performed, after which the access request travels to the next lower level object, where access controls necessary at that level can then be performed; this process continues until the lowest level (physical) object has been accessed.

5.4 Object-based PCM

This section describes a version of PCM for the object-oriented environment. (See chapter 3 and [Gol83,Weg90] for a description of this environment.) Our reasons for using the object-oriented environment include:

- The problem described in the previous section is aggravated in the object-oriented environment, because all entities are objects; further all non-primitive objects are built on top of other objects; and
- The fact that all entities in the object-oriented environment are objects, simplifies the resulting model.

Although we use the object-oriented environment and terminology, the described model can easily be adapted for conventional environments.

Our model assumes that objects are instantiated and classified before the profile is defined. This makes it applicable to real-world environments such as the client-server model, where the servers can indeed be instantiated and classified before defining the profile.

5.4.1 Notation

The underlying concepts used for describing PCM were

- Domains
- Software components
- Integrity states
- Objects
- Accessors
- Access classes

In the pure object-oriented environment software components, accessors and 'physical' objects are all viewed as ***objects***. Also, for any such object, the valid access classes are represented by the methods supported by such an object.

While domains and integrity states may also be modeled as objects, they do belong to a different level: they represent the underlying computing environment, while the objects represent the logical entities supported by that environment. To illustrate this consider the following: to the programmer it should not make any difference whether object x is residing on domain d_1 or d_2; also it is immaterial whether an object is one designed by an applications programmer or whether it is an operating systems object executing in privileged state; however these differences have major security implications.

In the version of OPCM we describe, we only make provision for objects to be protected. However, it is desirable that it should be possible to protect other entities such as individual methods and instance variables of a given method. Also we make no special provision for the protection of classes. We return to these issues after defining OPCM.

The following notation is used to describe OPCM:

$$\begin{aligned} E &= O \times M \times D \times I \\ O &= \{o \mid o \text{ is any valid object}\} \\ M &= \{m \mid m \text{ is any valid method}\} \\ D &= \{d \mid d \text{ is any valid domain}\} \\ I &= \{i \mid i \text{ is any valid integrity state}\} \end{aligned}$$

Note that the definitions of O, D and I differ from those given when defining PCM—this is because we have incorporated the access classes (represented by K there) as methods in the description above.

As mentioned earlier, $O \times M$ represents the active entities, while $D \times I$ represents the underlying computing environment. Elements of $O \times M$ are ordered pairs consisting of an object name, followed by the name of the method currently active in that object. Elements of $D \times I$ are ordered pairs describing a physical component and its current integrity state: this could be a CPU executing in, say, privileged state or a communications line currently carrying data in, say, unencrypted form.

As is the case in PCM, the described environment reflects security requirements in current systems; if any given system has more variables to keep track of, it is only necessary to add those variables to the environment and extend the baggage collection vehicle and profile descriptions accordingly.

5.4.2 The baggage collection vehicle

The definition of the BCV corresponds largely to that given earlier for the BCV of PCM.

Definition 5.3 *The baggage collection vehicle is a formal grammar*

$$C = (V_N, V_T, P, \Sigma)$$

where V_N is the set of non-terminal symbols, with

$$V_N = \{\Sigma; \mathcal{O}; \mathcal{M}; \mathcal{D}; \mathcal{I}; \mathcal{C}\}$$

V_T is the set of terminal symbols, with

$$V_T = O \cup M \cup D \cup I$$

$\Sigma \in V_N$ is the starting symbol. P is the following set of productions:

$$\begin{array}{rcl}
\mathcal{O} & \rightarrow & o_1|o_2|o_3|...|o_r|\varepsilon \\
\mathcal{M} & \rightarrow & m_1|m_2|m_3|...|m_n|\varepsilon \\
\mathcal{D} & \rightarrow & d_1|d_2|d_3|...|d_m|\varepsilon \\
\mathcal{I} & \rightarrow & i_1|i_2|i_3|...|i_p|\varepsilon \\
\mathcal{C} & \rightarrow & \mathcal{OMDIC} \\
\mathcal{C} & \rightarrow & \mathcal{O} \\
\Sigma & \rightarrow & \mathcal{C}
\end{array}$$

where

$$\begin{array}{rcl}
o_j & \in & O, 1 \leq j \leq r \\
m_j & \in & M, 1 \leq j \leq n \\
d_j & \in & D, 1 \leq j \leq m \\
i_j & \in & I, 1 \leq j \leq p
\end{array}$$

The value ε is used whenever a value is not defined, cannot be determined or does not make sense in the specific context. □

5.4.3 The profile

The profile definition for OPCM corresponds to the profile definition of PCM—reflecting the new environment.

Definition 5.4 *A security profile is an extended path context grammar*

$$C' = (V_N, V_T, P, \Sigma)$$

V_N is the set of non-terminal symbols, with

$$V_N = \{\Sigma; \mathcal{O}; \mathcal{M}; \mathcal{D}; \mathcal{I}; \mathcal{R}; \mathcal{C}\} \cup \{\Omega_j | o_j \in O\}$$

V_T is the set of terminal symbols, with

$$V_T = O \cup M \cup D \cup I$$

Let V represent the alphabet, ie

$$V = V_T \cup V_N$$

$\Sigma \in V_N$ is the starting symbol.
P is a set of productions of the form

$$\begin{array}{rcl}
\Sigma & \rightarrow & \mathcal{C} \\
\mathcal{O} & \rightarrow & \Omega_j, \text{where}\, \Omega_j \in V_N \\
\mathcal{M} & \rightarrow & m_j, \text{where}\, m_j \in M \\
\mathcal{D} & \rightarrow & d_j, \text{where}\, d_j \in D \\
\mathcal{I} & \rightarrow & i_j, \text{where}\, i_j \in I \\
\mathcal{O} & \rightarrow & \varepsilon \\
\mathcal{M} & \rightarrow & \varepsilon \\
\mathcal{D} & \rightarrow & \varepsilon \\
\mathcal{I} & \rightarrow & \varepsilon \\
\mathcal{C} & \rightarrow & \mathcal{OMDIR} \\
\mathcal{R} & \rightarrow & \mathcal{C}(\emptyset; G), \text{with} \\
 & & G = \{\Omega_j | o_j \in O\} \\
\mathcal{C} & \rightarrow & \mathcal{O} \\
\Omega_j & \rightarrow & o_j(B; F), \text{with} \\
 & & B \subseteq V \cup \{\langle x \rangle | x \in V^n, n \geq 2\} \cup \{\langle\langle x \rangle\rangle | x \in V^n, n \geq 2\}, \text{and} \\
 & & F \subseteq V \cup \{\langle x \rangle | x \in V^n, n \geq 2\} \cup \{\langle\langle x \rangle\rangle | x \in V^n, n \geq 2\}
\end{array}$$

□

In this case a baggage string consists of one or more 4-tuples, v_j, followed by the identity of the final object being accessed, with

$$v_j \in OMDI = \{omdi | o \in O \wedge m \in M \wedge d \in D \wedge i \in I\}$$

A baggage string is thus of the form

$$v_1 v_2 v_3 \ldots v_n o$$

Here v_1 represents the initial values for the accessor object, method, domain, and integrity state.

Note that the baggage at points where objects are to be accessed will always be of the form $\alpha\Omega_j\mathcal{R}$ or $\alpha\Omega_j$ with $\alpha \in V^*$. Similar to the case for PCM access to object o_j in such a case will only be granted if there is a production mapping Ω_j to o_j and α satisfies the constraints specified in the allowing and prohibiting contexts of the production. Also note that the production

$$\mathcal{R} \rightarrow \mathcal{C}(\emptyset; G)$$

will ensure that a request is not allowed to advance to another object if access has not been granted to all previous objects on the path of the request; ie a request will only be allowed to continue if every Ω_j encountered thus far has been replaced by its corresponding o_j.

5.4.4 The validator

The validator is identical to the validator of PCM—mapping the replacement of a non-terminal symbol representing an object with the terminal symbol representing that object to physical access to that object.

5.4.5 A critical look at OPCM

OPCM solves the problem of composite objects. However, some critical remarks are in order

- The notion of a request has still not been defined formally. Since a request is allowed to travel from object to object, requests could be of a relatively long duration (especially when compared to requests in PCM). This could be a problem because baggage is collected from initiation until termination of a request, possibly leading to an excessive amount of baggage. It may be possible to write software which will consider the allowing and forbidding contexts specified in the profile and then identify and discard superfluous baggage at some points along the path; it may even be possible to adapt the profile grammar, based on given contexts, so that some useless information is not collected in the first place. The formal nature of PCM/OPCM ensures that this can be done safely.

- We stated that it is desirable to be able to protect individual methods and instance variables of an object. In order to protect methods, it is a simple matter to add allowing and forbidding contexts to the production mapping $\mathcal{M}$ to M_j in the profile specification; this will be similar to the way such contexts were added to the production mapping $\mathcal{O}$ to o_j (with the introduction of Ω_j) to control access to the object. To protect instance variables, the reading from or writing to an instance variable may be viewed as a 'read' or a 'write' message sent to an 'instance variable object', in which case the given grammars will handle it without modification.

- Our assumption that objects are created and classified before definition of the profile side-stepped the problems associated with dynamic creation of objects and also with inheritance. These issues are addressed in subsequent research; the SECDB model for secure object-oriented databases (see chapter 6) makes use of OPCM and specifies additional rules to deal with both inheritance and object instantiation. Indeed, in the taxonomy described in chapters 7 to 11 these issues are assigned to different parameters, because it reflects the way most current models for secure object-oriented databases are designed: parameter X1.1 there specifies the underlying model (which may be for example OPCM), parameter X2.2 handles the security labeling of newly instantiated objects, while parameter X2.3 discusses the implications that inheritance (and other relationships) have on the security labeling of such related entities.

- Lastly, some remarks on the protection of classes: Classes are normal objects in an object-oriented environment. Therefore they may be protected just like any other object with OPCM. However, classes do have an additional purpose: they serve as templates for new objects that are created. This 'template section' of classes may be protected against unauthorised access (similar to the way methods and instance variables may be protected). Alternatively the 'template section' may be labeled with the intention that any objects created from the template must inherit the given protection, in which case the discussion in the previous point applies.

5.5 The role of OPCM

As mentioned earlier, OPCM will be used in the next chapter to define SECDB, a model for secure object-oriented databases. OPCM will also be used to develop the taxonomy for secure object-oriented databases in chapters 7 to 11.

In the remainder of this work we will use the term PCM to mean both PCM and OPCM: where PCM is used in an object-oriented context, OPCM should be understood.

Chapter 6

SECDB: A Secure Object-oriented Database Model

This chapter defines SECDB, a model for secure object-oriented databases. It uses the Path Context Model (see chapter 5) to protect the information contained by it.

SECDB uses normal object-oriented concepts—most of which are already present in an object-oriented database—to represent security features. This enables one to encapsulate security restrictions in an object—similar to the way instance variables and methods are already encapsulated in the object. This leads to objects that have the 'intelligence' to protect themselves.

See our overview of database theory in chapter 4 for information on object-oriented databases; see the discussion of object-orientation in chapter 3 for information on the object-oriented paradigm itself. SECDB is used as one of the models on which the taxonomy for secure object-oriented databases in chapters 7 to 11 is based.

Cet animal est très méchand,
Quand on l'attaque il se défend.
This animal is very bad; when attacked it defends itself.

Anonymous

6.1 Introduction

This chapter describes a model for a secure object-oriented database based on the Path Context Model (PCM) [Bos89a,Bos89b,Bos90]. We will refer to this model as SECDB (SECure DataBase).

The necessary extensions to an object-oriented database to support security are considered. Such extensions must fit as cleanly as possible into the object-oriented model. The proposed mechanisms are designed to be objects themselves, conforming to the normal object, class and inheritance structures. Also, in line with the object-oriented database model, the security information must be stored as part of the database, together with the data and code.

We will mainly be concerned with the secrecy aspect of security, although the proposed model has some advantages regarding integrity.

6.2 Object-oriented extensions

This section introduces the terminology used. The concepts are treated in more detail in subsequent sections.

6.2.1 Object-oriented basics

We consider objects, classes, inheritance, methods and messages as the necessary ingredients of object-orientation [Weg90, pp 8–11]. Also, in a secure system encapsulation [Weg90, p 11] must be enforced—that is an object's instance variables must be hidden from all methods except the object's own methods.

6.2.2 Messages and baggage

Throughout this chapter we use the term 'message' to refer to an active message: it starts to exist when it is sent (by a user or from some method which is being executed). We will say a message 'activates' a method; messages sent by the executing method are 'spawned' by the original message; the message directly responsible for sending a message will be referred to as the 'spawning' message.

Example 6.1 *See figure 6.1. Suppose a user sends the message* GETCOST *to an object* PROJECTX. *The method* GETCOST *sends the message* GETRATE *to the object Worker and then* GETDATE *to the object* CLOCK. *We say that* GETCOST *spawned* GETRATE *and* GETDATE. *The spawning*

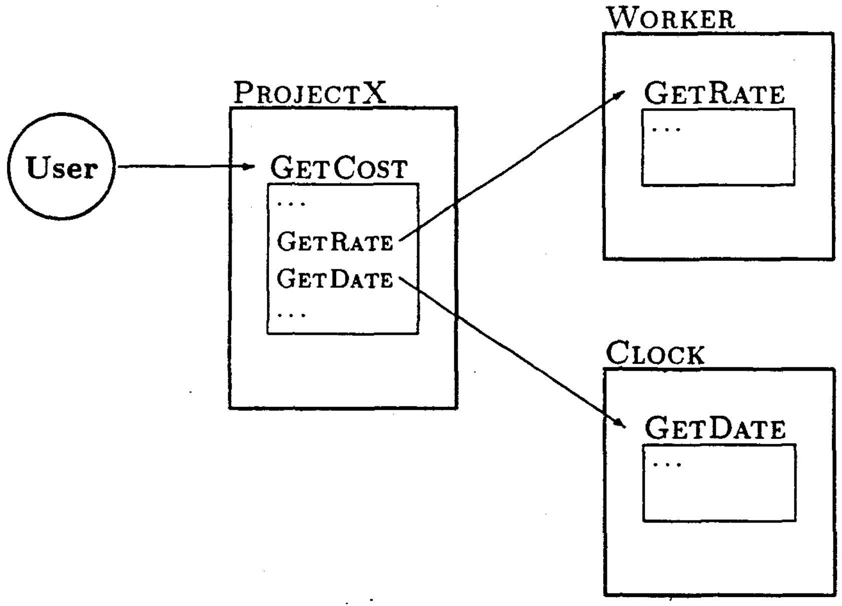

Figure 6.1: The operation of messages

message of both GETRATE *and* GETDATE *is* GETCOST. *Note that the message* GETCOST *starts to exist when it is sent by the user;* GETRATE *exists until it returns an answer. After that* GETDATE *starts to exist and ends its existence when it returns the date. After this the* GETCOST *message stops to exist when it returns its answer.*

Messages are required to carry 'baggage' with them. SECDB is designed to support the baggage concept as defined in the Path Context Model (PCM) [Bos89a,Bos89b,Bos90]; see also chapter 5. Informally, PCM requires that 'baggage' or information must be collected about all relevant software and hardware components involved in the handling of any request. A profile is associated with every item in the system to which access must be controlled—this profile specifies which components are required to take part in the handling of the request and which components are not allowed to take part. A profile may, for example, be defined to

1. Allow access only if one of a few specified users has originated the request; but

2. Deny access if some non-encrypted communications line has been used to transmit the request.

We expand on this later; initially this baggage may be viewed as the 'clearance level' of the sender or originator of the message: it is an indication of the trust associated with the request. Some researchers prefer the term *guardian* [Cae90] for what is known as a *profile* in SECDB.

6.2.3 Profile objects

SECDB defines a profile as an object containing security information about protected information; given the baggage carried by a message, a profile can use its security information to determine whether the message may be allowed to access the protected information. The security information will typically be allowing and prohibiting contexts as defined in PCM; initially this information may be viewed as the 'sensitivity level' or 'classification level' of the information protected by the specific profile—ie a restriction of who is allowed to access the information. Profiles are normal objects—instances of profile classes—placing no additional requirements on the underlying system.

Since messages carry information they need protection and will therefore also have profiles associated with them. The information carried by a message will typically consist of parameters; however, the information can be much more subtle: the fact that a message has been sent can compromise information. Note that messages are active entities which cannot be accessed, assuming that the operating system protects executing programs: messages exist as method activations—similar to an activation of a procedure in a conventional system [Aho86], with the activation record usually kept on an internal stack protected by the operating system and/or hardware. Messages therefore do not pose a security threat—the problem arises when such a message 'drops' information somewhere where it can be accessed, eg by assigning it to a variable or by creating an object. So, whenever a message is sent, profiles are taken along for all the information contained in it—contained directly in its parameters or contained implicitly. These profiles carried by messages will then be associated with any objects created or modified by that message. The profiles associated with a message will also be tagged to messages spawned by it. This will ensure that the *-property [Bel76] is not violated when messages are sent between objects; simply put, this property requires that no user (or application) can write information he has access to where someone without the proper authorisation can then read it. We will formalise this mechanism later. See section 6.5 for an example.

It is important to distinguish between baggage carried by messages and profiles carried by messages. Baggage is a record of the path a request has followed, ie which user initiated the request, which application programs were involved, which networks have taken part, etc. When a message arrives at an object, the object's profile(s) will look at this baggage to determine whether access should be granted to the message. On the other hand, a message also carries profiles with it to protect the information contained by the message. These profiles will have no influence on what the message may or may not access; the profiles will rather be attached to any variables modified or objects created by the message, controlling access to those variables and objects.

6.2.4 Gates

In SECDB a gate is a 'boundary' around every object in the system: whenever a message is sent to an object, the message will be intercepted by the sending object's gate and then by the receiving object's gate. The sending object's gate will tag any profiles associated with the sending object to the message. The receiving object's gate will ask access permission from the receiving object's profile(s). The profile will make this decision based on the baggage sent with the message and its internal profile information. If access is denied, it is the gate's responsibility to handle the rejection of the message. If access is allowed, the gate will associate any profiles carried by the message with the instance variables changed or created in the receiving object (in addition to any existing profiles).

6.3 Profiles

6.3.1 Description of profiles

Profiles will be defined for information in a SECDB database to which access is restricted. Profiles are objects containing access requirements: these requirements will typically consist of a list of objects (users, programs, etc) that may take part in a request to access the information and a list of objects that may not take part in a request to access the information. Based on the baggage carried by a message and these requirements in a profile the profile can determine whether the message satisfies the access requirements or not. PCM specifies that the access requirements must be in the form of a Random Context Grammar [Van70]: it will specify which contexts (ie human, software and hardware components) are required to be in the baggage and which contexts are not allowed to appear in the baggage.

The profile may also support methods to dynamically modify the profile data; however, if such mechanisms are too general, the resulting security may be too complex, reducing trust. We do not consider meaningful restrictions to methods supported by profiles in the current study.

SECDB additionally requires that a profile object provides a method GRANTACCESS, which may be called by the protected object's gate and which will reply with a *Yes* or a *No*.

Note that the PCM requirement of a Random Context Grammar profile may be modified to yield other interesting versions of SECDB; we discuss one such possibility later.

6.3.2 Profiles as objects

The implications of profiles being objects are

1. There are profile classes, subclasses and superclasses; and

2. Profile objects are protected by profiles themselves.

Considering (1), it is convenient to assume that a profile class has more restrictive access requirements than its (profile) superclass, ie that each subclass has a higher (or equal) 'sensitivity level' than its superclass. A hierarchy of profile classes then forms a partially ordered set of 'sensitivity levels'. In order to guarantee this, SECDB requires that GRANTACCESS in any profile subclass will send a message to GRANTACCESS in the superclass of the profile subclass; access will be denied if it is denied by the superclass.

From (2) it follows that there is potentially an infinite chain of profiles. This problem is easily (and soundly) solved by allowing one profile object to act as its own security profile—the problem is similar to the one caused by the fact that classes are indeed objects and therefore instances of some metaclass, which is again an object itself.

6.3.3 Associating profiles with information

In order to protect information, profiles are specified when a class is defined. The specified profile(s) will then be used for all instantiations of that class.

An object may be viewed as a layered entity: The outermost layer contains those instance variables and methods (and shares those class variables) defined in its class; the next layer those defined in the superclass of its class, and so on up to the innermost level which contains the items defined highest up in the class hierarchy. (In Smalltalk terminology [Gol83], it contains the class variables and methods defined in class OBJECT.) Figure

6.2 illustrates this. Since information at different levels in the class hierarchy may have different security requirements, different layers of an object may be protected by different profiles.

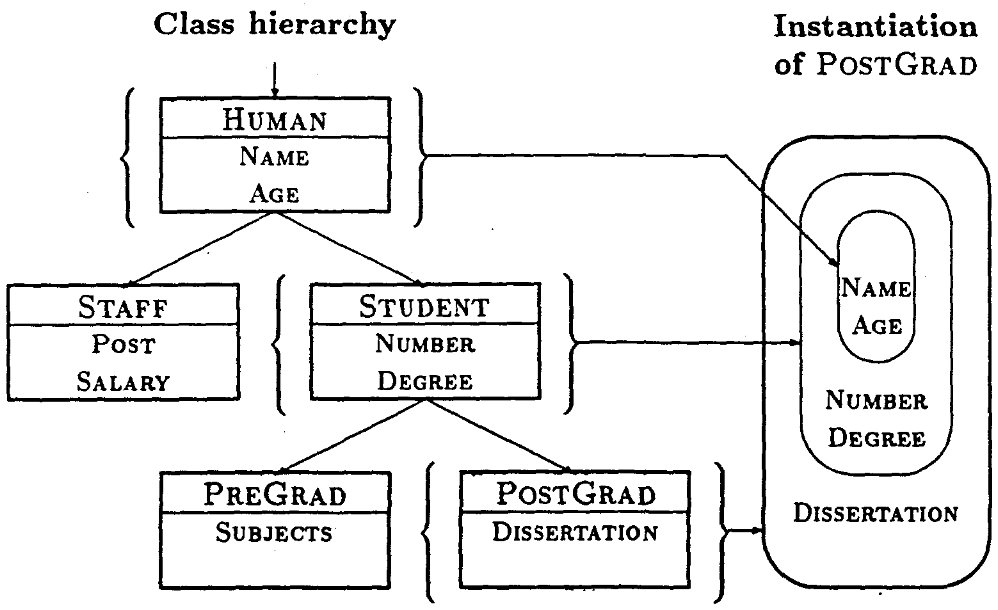

Figure 6.2: Objects are layered entities

A profile may be associated with one of more of the following:

1. A layer of an object;
2. Specific methods of an object; and/or
3. Specific instance variables of an object.

In order to activate a method

1. Profile(s) protecting all object layers surrounding the method must allow access;
2. The method's specific profile must allow access; and
3. None of the instance variables accessed by the method must deny access.

Typically, a profile will be associated with the object reflecting the object's overall 'sensitivity'. Profiles will be associated with the methods to implement role-based security: the set of operations any user can perform on an object does not only depend on that user's clearance level, but also on the user's function or role in the organisation. Profiles which are associated with instance variables will mainly be used as an additional safeguard to ensure that no method accesses information which the subject should not have access to.

An object, message or variable may have more than one profile associated with it, in which case all the profiles must allow access before access will be granted for a message.

In figure 6.3 OBJECT1 is protected by both PROFILE1 and PROFILE2, while METHOD1 is additionally protected by PROFILE3 and PROFILE4. In order to activate METHOD1, permission will have to be obtained from all four profiles; if any one denies access, METHOD1 will not be activated.

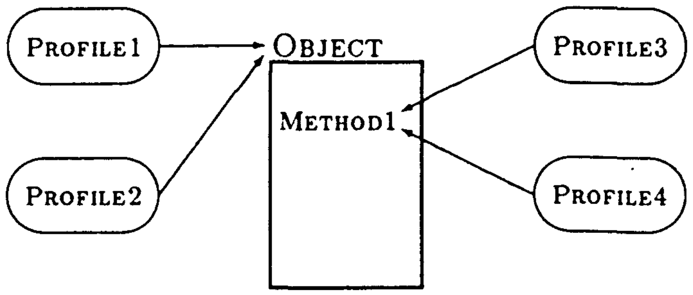

Figure 6.3: Multiple profiles

6.4 Gates

6.4.1 Operation of gates

A gate is a mechanism associated with every object, which will intercept any message leaving the object or arriving at the object. Arriving messages will only be allowed to activate the called method if permission can be obtained from the relevant profile(s). In the case of leaving messages, the profiles associated with the sending object will be tagged to the message. Whenever an instance variable with a profile is accessed, the message will be tagged with that profile. Messages spawned by another message will be tagged with all the profiles associated with the spawning message. If a spawned message returns a value, the spawning message will be tagged with the profiles carried by the returning message; if no values are returned the spawning message will resume execution with all the profiles it had before it spawned the message which has just returned.

All messages crossing the borders between object layers will be checked by the gate, verifying baggage when traveling inwards and attaching profiles when traveling outwards. In order to reference any method or variable at an inner layer, all border crossings between the origin of the reference and the referenced variable or method have to be checked. This ensures that information does not flow to a lower classified class by means of inheritance.

6.4.2 Examples of message tagging

Assume we are working with a payroll system. All employee salaries are protected by a profile P. The payroll object starts sending messages to all employee objects to determine their current salaries. When these messages return the result (the salary) the profile P is attached to the returning message. The payroll object inserts the salaries into its internal table—this implies that a variable is being modified, causing the profile P to be attached to this variable. Although the payroll object now contains the sensitive salaries, it is not possible to retrieve them illegitimately from this object because they are still protected by their original profile P. If this payroll object now sends a message to the cashbook object with this table as a parameter, the profile P will again be sent along and (eventually) be attached to the cashbook's variable(s) so that it will still not be possible to obtain the sensitive salaries from the cashbook without the (original) authorisation.

Another example of tagging profiles to messages is given in figure 6.4. There are three objects, O1, O2 and O3, six messages, M1 to M6, and eleven profiles, P1 to P11.

If we firstly assume that all messages return values and that the message M1 is sent to object O1, the sequence of events will be as given in table 6.1.

Note the following (numbers refer to the *Remark* column in table 6.1):

1. We assume that the message starts without any profiles attached to it.

2. Sending the message M3 to object O2 causes the activity to 'leave' object O1 and be transferred to object O2. The 'leaving' of O1 causes O1's profile P1 to be attached to the message.

3. Object O2 now sends the message M6 to O3. This message first 'leaves' the layer of O2 which is protected by P3 and then the layer protected

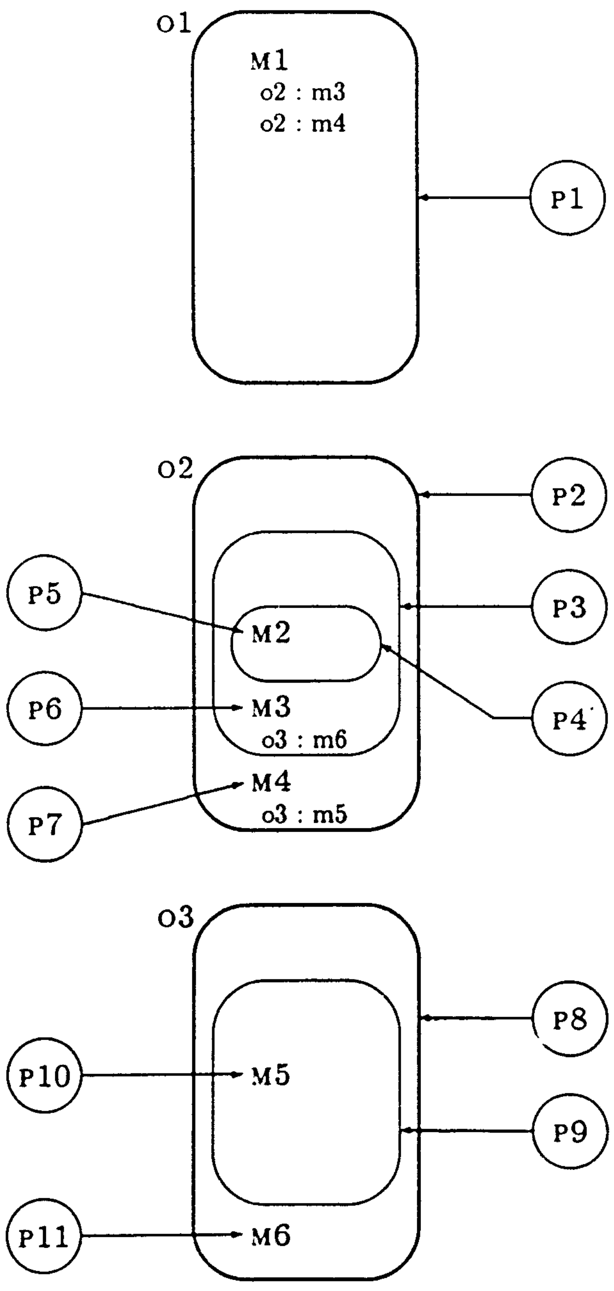

Figure 6.4: Tagging profiles to messages

Object	Active message	Profiles tagged to message	Remark
O1	M1	-	1
O2	M3	P1	2
O3	M6	P1,P3,P2	3
O2	M3	P1,P3,P2,P8	4
O1	M1	P1,P3,P2,P8,P3,P2	5
O2	M4	P1,P3,P2,P8,P3,P2,P1	
O3	M5	P1,P3,P2,P8,P3,P2,P1,P2	
O2	M4	P1,P3,P2,P8,P3,P2,P1,P2,P9,P8	
O1	M1	P1,P3,P2,P8,P3,P2,P1,P2,P9,P8,P2	

Table 6.1: Profiles tagged if the methods of figure 6.4 return results

by P2 causing both P3 and P2 to be attached to the message. P1 is still attached to the message because the message was spawned by M3 of O2 with which P1 was then associated.

4. Control now returns to method M3 of object O2. Since information is returned (our initial assumption), this information 'leaves' the layer protected by profile P8, causing P8 to be attached to the message.

5. Control has now returned from M3 of O2. Again, since the message returned a value to M1 of O1, 'leaving' from M3 caused profiles P3 and P2 to be attached to the message. (Note that P3 and P2 are already attached to the message—in an optimised environment it is not necessary to attach them again.)

If we assume, on the other hand, that no message returns a value, the sequence of events will be as given in table 6.2 if the message M1 is sent to object O1.

Note the following (numbers refer to the *Remark* column in table 6.2):

1. Control returned to method M3, object O2 from method M6, object O3. Since no value is returned (our assumption) M3 resumes execution with the profiles it had before sending message M6.

2. M1 resumes execution with the profiles it had before sending message M3.

3. M4 resumes execution with the profiles it had before sending message M5.

4. M1 resumes execution with the profiles it had before sending message M4.

Object	Active message	Profiles tagged to message	Remark
O1	M1	-	
O2	M3	P1	
O3	M6	P1,P3,P2	
O2	M3	P1	1
O1	M1	-	2
O2	M4	P1	
O3	M5	P1,P2	
O2	M4	P1	3
O1	M1	-	4

Table 6.2: Profiles tagged if the methods of figure 6.4 do not return results

6.4.3 Problems posed by gates

The concept of a gate is the only real deviation in SECDB from the classical object-oriented concept and (not surprisingly) poses the most challenges. These challenges include

1. Handling of messages when access is denied;
2. Cooperation with the object's profile, especially when the profile requires information from the object in order to decide whether access should be granted; and
3. Logical integration into the object-oriented model, especially where such a model is described formally.

Problem (1) can be treated like a normal failure to complete execution of a method, eg when the hardware fails or the disk media is unreadable. This will typically involve abortion of the current transaction and rolling the database back. We do not go into further details here.

Polyinstantiation is often used to solve (1): different 'slots' for a single field are maintained for every clearance level; when a process with clearance level n writes a value to a field, any other process with clearance level n (or higher) can read that value. However, if a process with clearance level m, where m is more trusted than n, writes a value to the same field, all processes at clearance level m and higher will be returned this value when they read the field, while all processes at clearance levels n to $m-1$ will still be returned the value written by the process at level n. (See chapter 11 or [Thu89] for a description of polyinstantiation in object-oriented systems.) Polyinstantiation has to be adapted to be usable in SECDB since data is not necessarily classified at different levels in the PCM model. This means that it is not always possible to determine (and, in any case, it might be dangerous when it is possible) how many 'slots' may exist for the same value. Polyinstantiation will have to be done by attaching 'cascading' profiles to a polyinstantiated object—one level of profile for every copy of the object. If a profile denies access, the profile one level lower in the cascade is asked for access; this process is continued until a profile allows access; in this case access will be granted to the (copy of the) object on the same level as the granting profile. Only if none of the profiles grants access, will the request be denied. We do not discuss cascaded profiles further in this chapter.

Problem (2)—cooperation between an object and its profile—can be illustrated as follows: Suppose some user must have access to every non-manager's employee information. The profile for the employee objects must then first determine whether a specific employee is a manager before granting access. If the fact whether an employee is a manager or not forms part of the employee information, this implies that a profile must sometimes access an object which may not be accessed by the sender of the original message. A similar, but slightly more difficult restriction to solve by *ad hoc* programming, is a rule allowing access to a salary if and only if the salary is less than $100 000. In order to handle this kind of restriction, SECDB allows a profile to access an object (by sending messages to it) without getting permission from a profile. To ensure that profiles are not used to extract protected information from an object, access to the profile may be restricted to come from gates (and possibly from some other highly trusted objects, but not from arbitrary objects).

6.5 Example

The example in figure 6.5 is intended to illustrate profiles, profile classes and hierarchies of profiles. In a real application we would like to see a classification scheme based on functions (or roles) in the organisation, rather than on traditional classification levels. In figure 6.5 SECPROFILE is the primitive security profile class. UNCLASSIFIED, RESTRICTED, SYSADMIN, etc, are all profile (sub) classes. SECPROFILE, UNCLASSIFIED, RESTRICTED, CONFIDENTIAL, SECRET and TOPSECRET form a non-decreasing sequence of profile classes. SYSADMIN also offers at least as much protection as SECPROFILE. However, no such relationship exists necessarily between SYSADMIN and UNCLASSIFIED, RESTRICTED, etc. In the example this is used where some profile classes are intended to protect information according to the classification level of the information and another class, SYSAD-

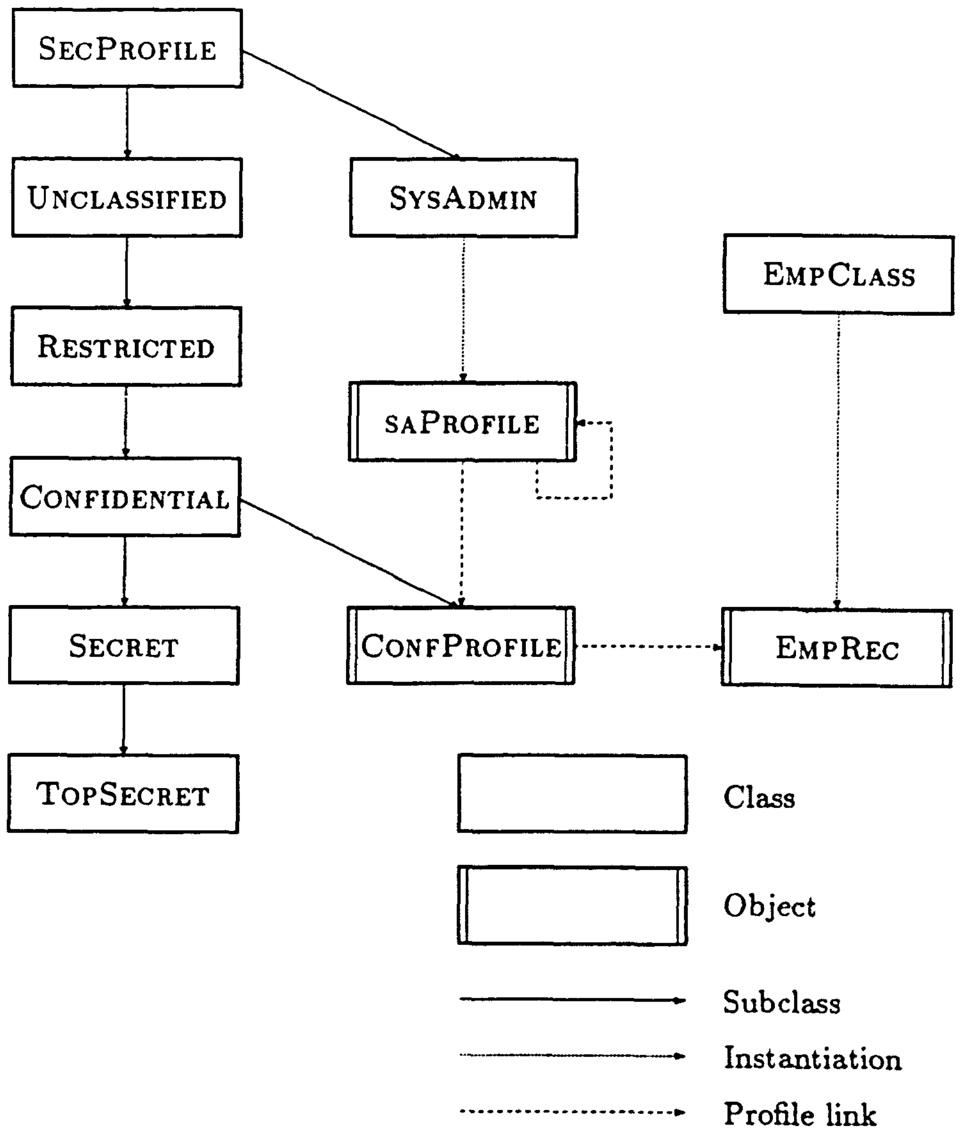

Figure 6.5: Example

MIN, is used to protect system structures, such as profile objects.

EMPCLASS describes a class of employee objects. When defining this class, the system security officer decided to link it to the CONFPROFILE profile. Every instance of EMPCLASS, such as EMPREC, will then be associated with CONFPROFILE.

6.6 Model characteristics

In SECDB a previously defined and instantiated security profile is specified (or inherited) for every class whenever such a class is defined.

Subclasses of a class will, by default, inherit the profile of the superclass. A security profile may, however, be specified when defining a subclass, overriding the inherited profile. Viewing an object as a layered entity implies that security (both the simple security property and the *-property) [Bel76] will not be compromised, even if the subclass has less restricting access requirements than its superclass. The simple security property requires that subjects should not have access to information they do not have permission to access and the *-property requires, as mentioned earlier, that no subject can write information where another subject without the proper authorisation can then read it.

The simple security property and the *-property have to be demonstrated for two cases:

1. Where messages are sent from one object to another (possibly at different classification levels); and
2. Where an object inherits a message or variable from some superclass (again possibly at a different classification level).

The following arguments do this for the respective cases.

1. When messages are sent from one object to another, the gate/profile combination(s) of every object will ensure the simple security property. Since information sent along with any message carries with it the profile of any object it leaves, the *-property is ensured: any access to that object will have to go through the original containing object's profile, implying that a lower "clearance" level will still not be allowed to access it, although it may now be contained in an object with a lower classification level.
2. When a message or variable is inherited from a superclass, the profile specified in that superclass will be associated with the corresponding layer of the object. Information contained may therefore not be

accessed without permission from that security profile, ensuring the simple security property. Also, any information leaving that layer will be tagged with that layer's profile, ensuring the *-property.

6.7 PCM in the SECDB environment

As we indicated earlier, SECDB has been designed to use PCM. The primary advantage of PCM is its ability to enforce security in potentially non-secure environments, especially in network environments where the identity of the originator of a request is often concealed by layers of application software, network servers and system software. SECDB also shows promise in a distributed database where multiple system security officers may be involved. (The original description of PCM uses the term *object* to refer to a static item; remember that we use the term *entity* to avoid confusion in the object-oriented context.)

PCM uses three concepts:

1. A baggage collector to collect 'baggage' about all software and hardware involved in handling a request; baggage is defined as 'the minimum amount of information that has to be collected and must accompany the access request on its route in order that responsibility and access authority checking can be performed even though various transformations or domain crossings may occur';

2. A profile for an entity which specifies which items are required to take part in handling a request and/or which items are not allowed to take part when accessing this entity; and

3. A validator to compare collected baggage to the accessed entity's profile.

In SECDB, to fit the object-oriented model, the profile data (containing the allowing and prohibiting contexts) and the validator have been merged into a single profile object.

Although PCM may be used to protect non-object-oriented databases, the object-oriented model is more appropriate, because

- Of the similarity between messages in the two models;

- Of the ability to store data, methods and security information as one coherent unit; and

- In the case of the object-oriented model it is possible to consider security implications at any abstraction level—specifically it is possible to implement role-based security; in the case of other models one can usually only make decisions on whether a subject is allowed to read and/or write specific data records or fields.

In an object-oriented database messages can originate from objects inside the database or from external sources. Although not the primary goal, SECDB can be used to ensure that requests follow an approved route through the database: some objects are for internal use only, or for use by specific other objects; it is simple to add restrictions to SECDB profiles to ensure that a request only comes via an acceptable predecessor helping to ensure the integrity of the database.

6.8 Implementation considerations

We do not address the implementation considerations in detail in this chapter. However, it must be admitted that a straightforward implementation of the model could be inefficient. Here we point out five areas where optimisation shows promise.

Firstly, an object protected by multiple profiles requires extensive checking. If some of the associated profiles have stricter requirements than other associated profiles, the less strict profiles may be ignored. The ordering imposed on profile subclasses stated that a subclass has stricter access requirements than its superclass(es). This means that if a profile from somewhere low in the hierarchy has granted access, the profiles from higher in the hierarchy (instances of ancestor classes) need not be asked for their permission.

Secondly, checks that messages sent inside the environment conform to the profile restrictions can often be done statically (at compile time or system configuration time).

Thirdly, the need for a profile to send a message to its superclass' GRANTACCESS can also be eliminated in some cases: if it is possible to check statically that the access requirements set by a profile class are indeed at least as strict as the requirements set by its superclass.

Fourthly, it is unnecessary to tag a profile to a message if the specific profile (or an instance of a descendant class) is already tagged to the message.

Lastly, it may be expensive if every object, down to the primitive objects such as numbers, may be protected. However, access checking will only take up time if such objects are indeed protected; if profiles are not associated with such entities there is no reason for access to them to be slower than the case in an unprotected database.

It is also worth pointing out that the association of a profile with an object is not expensive: this association can be done by having a pointer (often from the class and not the individual objects) to the relevant profile. This will not require much more memory than a simple sensitivity level would have taken.

6.9 Comparison with other models

One promising model for a secure object-oriented database system was proposed by Keefe, Tsai and Thuraisingham [Kee89,Kee90]. Their model, known as SODA, assigns a classification (sensitivity) level to a protected object. Every message which travels through the system, carries with it a current classification level and a clearance level. The current classification level is adjusted whenever an object with a higher classification is accessed. Rules, based on the current classification level and the clearance level and on an object's classification level, determine whether access should be granted to an object.

It can be shown that SODA is similar to a special case of SECDB: Define baggage to consist of

1. The clearance level of the sender of a message; and

2. The sensitivity level of any information accessed

and define profiles to contain the sensitivity levels of objects they protect. Note however, that the primary concern in SODA is the protection of variables and objects, whereas methods are the primary concern in SECDB, with variables and objects secondary.

The access control language of Mizuno and Oldehoeft [Miz90] is based on extended access control lists (ACLs): a four-tuple ACL entry

1. Specifies which user may activate a method;

2. Through which class the request may come;

3. Through which specific object (class instance) the request may come; and, lastly

4. Names the protected method.

SECDB is more general; however note that SECDB does not consider classes encountered on the route of the request, but rather objects—ie instances of classes.

Two other methods which may be used to ensure that a request follows an acceptable internal route through the database have been suggested in the literature: views [Hai90] and a law-based approach [Min87]. In the case of views, an object may define different views—or interfaces—for different objects. A profile in SECDB can specify which objects may access the protected object and also which methods are available to those objects, similar to the view concept. However, this specification in SECDB is not limited to the immediate predecessor object involved in a request. In the law-based approach a set of Prolog rules or laws may be specified to manage the exchange of messages in the system. This is a very general (and powerful) mechanism, enabling (and aimed at) the specification of basic properties, such as inheritance.

Various models for secure object-oriented databases will be encountered again in chapter 7. It will also be possible to compare the models thoroughly after the taxonomy for such models have been described; see for example appendix B.

6.10 Further research

Research still remains to be done in the following areas before SECDB will be practical:

1. Automated profile generation and automated validation of profiles (for consistency);

2. Identification of unnecessary baggage in order to keep baggage as small as possible;

3. Implications of multiple inheritance on the model;

4. Designing a notation for specifying security constraints (especially in real-world systems); and

5. The model should be applicable to distributed systems where requests may come from many sites, often not even directly connected to the database site; note that the distributed systems may operate concurrently, and the effects of this on the model should be investigated.

6.11 SECDB in perspective

Object-oriented systems are based on the premise that objects are self-contained entities consisting of data and code. It is meaningful to expect that objects should also have the responsibility to protect themselves. Further, protection in object-oriented systems must fit as cleanly as possible into the generally accepted object-oriented paradigm. In this chapter we have proposed a model, SECDB, for supporting security in an object-oriented database, which satisfies these criteria.

SECDB is based on the Path Context Model (PCM), which means that the entire access path is taken into account when it is decided whether a request should be allowed to proceed. This information collected while traversing the access path, is known as baggage. Two concepts are added to the object-oriented paradigm: profiles and gates. Profiles contain criteria for restricting access to an associated object. Gates are mechanisms which will compare baggage accompanying a request to the relevant profile(s), and then either allow or disallow the request to proceed. Profiles are objects themselves. They are also designed to be inherited by a subclass similar to the way instance variables are inherited.

Although SECDB is based on PCM, properties of traditional security models such as the Bell and LaPadula model can still be supported. We illustrated this by indicating how SECDB can emulate SODA, a model for a secure object-oriented database based on the Bell and LaPadula model.

In the next chapter we start with the discussion of the taxonomy for secure object-oriented databases. SECDB plays a central role in the discussion of the taxonomy.

Chapter 7

A Taxonomy of Secure Object-oriented Databases

Models for secure object-oriented databases may differ in a number of respects. Our taxonomy identifies eight major design parameters every designer of a multilevel secure object-oriented database must consider. These eight parameters are grouped into three categories.

Chapter 7 gives the background information on which the taxonomy is built. Amongst other, it joins the discussion of object-orientation in chapter 3 by introducing the notation that will be used in the taxonomy for the various object-oriented concepts. It further joins the discussion of security and object-orientation in databases in chapter 4 by describing examples of secure object-oriented databases.

Chapters 8, 9 and 10 each discusses one of the three categories of design parameters mentioned earlier, after which chapter 11 discusses some unresolved issues.

L'embarras des richesses.
The more alternatives, the more difficult
the choice.

Abbé D'Allainval
Title of comedy

7.1 Introduction

A number of models for multilevel secure object-oriented databases have been proposed in the literature. The variety exhibited by the proposed models is an indication of the great number of possibilities that exist. It is necessary to compare the proposed models in a structured way to highlight the real differences. This allows one to determine what the effects of specific choices are on the rest of the model. It also allows researchers to focus on specific issues, rather than on a whole model at a time.

Unfortunately, classifying such models is not an easy task. As Sandhu put it [San90]: "The underlying assumptions adopted by each one and their motivating forces are somewhat different. This makes a relative comparison difficult since different assumptions and motivations inevitably lead to different design trade-offs."

The paper by Varadharajan and Black [Var91] follows a similar approach to ours: they indicate some issues that are relevant when designing a secure database model and also consider some of the alternatives that are available for each issue. However, their approach is less formal and less general than ours.

Both the appeal and concerns regarding security of object-oriented databases are mentioned in [Spo89]. On the positive side object-orientation supports encapsulation, the possibility to model security in real-world terms and the potential of inheritance. On the negative side the main problem occurs when a class and its superclass have different sensitivity levels: this may allow information to flow from a higher sensitivity level to a lower level.

This chapter serves as a background for the taxonomy: it describes the approach we follow, the assumptions we make about object-orientation and the various existing models for secure object-oriented databases. The taxonomy is described in chapters 8, 9 and 10 while chapter 11 discusses aspects of secure object-oriented databases not addressed in the taxonomy. A summary of the taxonomy is contained in appendix A and examples of the use of the taxonomy in appendix B.

7.2 Approach

Our goals are twofold. Firstly we want to give a classification structure that will enable one to compare different models for secure object-oriented databases. Secondly, we want to indicate some implications of possible choices.

To realise the first goal, we give a number of design parameters—issues one should consider when designing a model. We give eight such design parameters that represent the major issues for consideration. These eight parameters are grouped into three categories:

X1 Labeling semantics: *Underlying model* and *Protection interpretation* (see chapter 8);

X2 Structural labeling: *Protectable entities*, *Label instantiation* and *Relationship restrictions* (see chapter 9); and

X3 Dynamic labeling: *Authorisation flow*, *Sensitivity flow* and *Information flow restrictions* (see chapter 10).

For most of these design parameters we list a number of alternatives that are available.

Implications of choosing a specific alternative for a design parameter are given as theorems. Some proposed models already describe restrictions to their models [Thu89,Lun90a]; often it is not clear whether a model imposes such restrictions because they simplify the model in some way, or because they are necessary to ensure that security is not compromised. Further, it is not clear from those models how generally applicable such restrictions are, in other words do the restrictions still apply if some aspects of the model are modified? To address these problems, our assumptions are given as axioms; for theorems the circumstances under which they apply are given.

In order to classify and analyse models, the meaning of terms have to be clear; we describe the important terms as definitions. It is important to note that we do not claim that our definitions are the only proper definitions of the terms; nor do we claim that our classification structure gives the final word on secure object-oriented database classification. We do claim that if our definitions fit a given model, that model may be classified using the described classification structure and, also, that the given theorems are valid for such a model. We also hope that our definitions and classification structure will prompt researchers to investigate models that do not fit these definitions or the classification structure.

We now define a secure database for the purposes of this discussion.

Definition 7.1 (Secure database) *An object-oriented database is secure if*

1. *No subject is able to obtain information without authorisation;*

2. *No subject is able to modify information without authorisation;*

3. *No mechanism exists whereby a subject authorised to obtain information can communicate that information to a subject not authorised to obtain it; and*

4. *No subject is able to activate a method without authorisation.*

This definition does not allow any covert channels—neither storage, nor timing channels—which may mean that it is too strict for practical use. However, these points cover the four primary issues addressed by models for a secure database. Issues (1) and (3) are normally addressed by models to enforce mandatory security, while issues (1), (2) and (4) are usually addressed by models to enforce discretionary access control (see section 11.4).

7.3 Object-orientation

See chapter 3 for an introduction to the object-oriented paradigm. Our view of this paradigm comes from the Smalltalk programming language [Gol83]. We now list the relevant assumptions we make about object-orientation.

An object o is a set of facets (methods, instance variables, etc)—ie if m is a method of o, we will denote it by $m \in o$; similarly, the fact that o has an instance variable v, will be denoted by writing $v \in o$. The system consists of a set of objects. We will refer to this set as U. (More precisely, v is the *name* of the corresponding variable, m is the *signature* of the corresponding method and o additionally contains a unique identifier that distinguishes it from all other objects in the system U.)

Remember that classes are also objects (they are instances of meta-classes) and are therefore also members of U. They also have facets (class methods and class variables) that behave like their object counterparts. However, some facets of a class are not intended to be used directly, but serve as templates to be used when creating an instance of the class (or of one of its subclasses). To reflect this dual nature of a class we divide a class into an 'object section' (containing the class methods and variables) and a 'template section' (containing the template which is used when new objects are instantiated); every 'class object section' corresponds to exactly one 'class template section'. The term *class* in this taxonomy is used to refer to the 'template section' of a class, unless stated otherwise. (Everything said about objects, also apply to the 'object section' of any class.)

Let C be the set of all classes. (More precisely, C is the set of all 'template sections' of classes.)

For any object $o \in U$ we will denote the class of o by *class(o)*, with

$$class : U \rightarrow C$$

For any class $c \in C$, we will denote the set of superclasses of c by *sup(c)*,

$$sup : C \rightarrow \mathcal{P}C$$

where $\mathcal{P}C$ is the powerset of C. In the case of single inheritance, *sup(c)* will consist of a single element.

The assumptions we make about object-orientation are now stated as axioms.

Axiom 1 *If an object o is in the system, then the class of o is also in the system; formally*

$$o \in U \rightarrow class(o) \in C$$

Axiom 2 *Any facet x of an object $o \in U$ is also a facet of the class of o; formally*

$$x \in o \rightarrow x \in class(o)$$

Axiom 3 *If a class c has a facet x, any instance o of c will also have the facet x; formally*

$$(x \in c) \wedge (\exists o \in U)[c = class(o)] \rightarrow x \in o$$

Axiom 4 *If a class c is in the system, then all the superclasses of c are also in the system; formally*

$$c \in C \rightarrow (\forall d \in sup(c))[d \in C]$$

Axiom 5 *For any class $c \in C$, if x is a facet of a superclass of c, then x is inherited or re-defined by c; formally*

$$d \in sup(c) \wedge x \in d \rightarrow x \in c$$

These axioms do not model polyinstantiation—we will return to polyinstantiation later (see section 11.6).

7.4 Security in object-oriented databases

The following security models serve as examples for our discussion. Here we only mention the models; specific issues will be addressed when the design parameters are discussed.

- SODA [Kee89,Kee90] is a model for a secure database based on a general object-oriented model. Objects or instance variables are assigned ranges of sensitivity levels. Subjects are assigned clearance levels. Every message that travels through the system, carries with it a current sensitivity level and a clearance level. The current sensitivity level is adjusted whenever an object or variable with a higher sensitivity is accessed. Rules, based on the current sensitivity level and the clearance level of a message and on an object's or variable's sensitivity level, determine whether the method should be permitted access to an object or variable.

- SORION [Thu89] is based on the ORION object-oriented data model. Entities in the system (subjects, objects, variables, messages, etc) are assigned security levels. An extensive list of properties is given constraining assignment of levels to entities. A given security policy constrains access to protected items based on these security levels.

- Lunt has given some initial properties for a multilevel object-oriented database system [Lun90a]; here we will refer to these properties as the Lunt model. The properties specify minimum requirements for a secure database based on a general object-oriented data model.

- Views [Hai90] allow an object to display different "views" of itself to other objects. This is done by restricting from which other objects any given method may be invoked. An alternative approach for defining views is given in [Shi89]: multiple interfaces may be defined for a single object. A 'client' object may then decide through which view it wants to access the object. The first approach is more applicable for our purposes since it is precisely specified through which interface a given client object is permitted to access the protected object. Note that the use of views in relational databases for security is well-known, see for instance [Lun90b].

- The access control language of Mizuno and Oldehoeft [Miz90] is based on extended access control lists (ACLs): a four-tuple ACL entry specifies

  - Which user may activate a method;

  - Through which class the request may come;
  - Through which specific object (class instance) the request may come; and, lastly
  - Names the protected method.

- SECDB (see chapter 6) is based on the Path Context Model (PCM) [Bos89a,Bos89b,Bos90]; see also chapter 5. A profile object is associated with every protected entity. A request collects baggage as it moves through the system. Before a request is allowed to access an entity, the baggage carried by the request is considered by the entity's profile object; based on this baggage, the request is either allowed to proceed, or rejected. When a protected entity is accessed, that entity's profile is tagged to any messages subsequently sent. Rules specify how the protection of other entities are influenced when they are accessed by a message that has such associated profiles.

- In the law-based approach as proposed by Minsky [Min87] a set of Prolog rules (or laws) may be specified to manage the exchange of messages in the system. This approach is intended to describe general aspects of object-orientation such as inheritance; it can be used to describe security restrictions, but is too general to be practical for this application. We include it nonetheless because the ability to give constraints based on logic rules is useful, amongst other to control the assignment of sensitivity levels and to ensure integrity in general.

7.5 Design parameters

The design parameters considered in this taxonomy are grouped into the three categories:

- Labeling semantics;
- Structural labeling; and
- Dynamic labeling.

In accordance with the rest of this work, we will use the term *entity* to refer to an item that may be accessed in a computing system; we will use the term *object* with the meaning usually associated in the object-oriented environment. The term *subject* is used to refer to an active item. Referring to the mathematical notation introduced earlier, an entity can be any object in U; it can be any class in C; it can also be any facet of such an object or class; formally the set of entities E is defined as follows

$$E = U \cup \{\langle x, o\rangle | x \in o \wedge o \in U\} \cup C \cup \{\langle x, c\rangle | x \in c \wedge c \in C\}$$

Note that not all models allow all entities to be protected (see parameter X2.1 in chapter 9).

In a security model *subjects* normally send requests to *entities*. In the object-oriented environment an object is both the target for requests (ie it acts as an entity in the security sense) and an object is the issuer of requests to other objects (ie it acts as a subject). Consider any message. We will use the term *subject* to refer to the object that sent the message under consideration or, depending on the model, to refer to the sequence of objects that were involved in the sending of the sequence of messages prior to the message under consideration; the first item in such a sequence is usually the human 'object' that initiated the chain of messages; the other items are normal objects that received a message and then sent messages to other objects. We expand on this description when we discuss the underlying model (parameter X1.1 in chapter 8) and the flow of authorisation (parameter X3.1 in chapter 10). Although the precise definition of *subject* depends on the specific security model, we will assume that it is defined and that a set S of all such subjects exists; the exact composition of S will be discussed later. The term *entity* refers to an object, method or variable (or other facet) in the role of receiving the message under consideration.

7.6 The taxonomy

This chapter introduced the concepts that will be used in the taxonomy. The next chapter describes the first group of design parameters for the taxonomy.

Chapter 8

Taxonomy: Labeling Semantics

This chapter describes the first two of the eight design parameters used in this taxonomy. These two parameters answer the questions: "How is an entity protected?" and "What does it mean if an entity is protected?"—hence the heading *labeling semantics*.

Note that the two parameters described here are applicable for any type of database—not only object-oriented databases. The remaining two groups of parameters (described in chapters 9 and 10) are linked closely to the object-oriented paradigm.

You see it's like a portmanteau—there are
two meanings packed up into one word.

Lewis Carrol
Through the Looking-Glass,ch.6

8.1 Introduction

We use the term *labeling* to refer to the assignment of a security category to an item. In the case of a subject a *clearance* is usually assigned. In the case of an entity a *sensitivity* or *classification* is usually assigned. If a subject attempts to access an entity, the system will use the clearance label of the subject and the sensitivity label of the entity to decide whether the subject should be allowed to access the entity; the exact way this decision is made depends on the underlying security model.

The first design parameter we consider is consequently the underlying model used by the secure database model. Some of the possibilities for an underlying model that we encountered in previous chapters are the lattice security models (see chapters 2 and 5) and the Path Context Model (see chapter 5).

The second design parameter considers the question exactly what is protected if an entity is labeled. On the one end of the scale an unauthorised subject is simply not allowed to retrieve information from such a protected entity (or to modify such an entity); on the other end of the scale an unauthorised subject is not even allowed to know (or infer) that the protected entity exists. It will be shown in the next chapter that this issue does indeed have major implications for the rest of the database security model.

8.2 Labeling semantics

We now describe the two concerned parameters:

(X1.1) Underlying model; and

(X1.2) Protection interpretation.

8.2.1 (X1.1) Underlying model

The model on which labeling is based falls in one of three broad categories, or a combination of some of these categories:

- *Explicit levels*: sensitivity levels are assigned to entities and clearance levels to subjects. These levels are normally integers. Rules determine when a subject may access an entity; often a subject may read an entity if the subject's clearance level dominates the entity's sensitivity level. In general, the level labels need not be integers—as long as the $\geq$ relation is defined for (some of) the labels associated with the subjects and the entities. In many models the same label act as an indication of an item's clearance when viewed as a subject and its sensitivity when viewed as an entity.

- *Access control lists* (ACLs) are lists associated with entities, containing the identities of subjects that are authorised to access the entity. Extensions of ACLs have been proposed that do not only contain the identity of authorised accessors, but also the path such a request has to follow; see for example [Bos89a,Miz90].
- *Capabilities* are non-forgeable identifiers possessed by subjects. Such a capability is similar to a key for a padlock: a subject will be allowed to access a protected entity only if it presents an acceptable capability.

A combination of the first two approaches is popular: Entities are classified using a sensitivity level and a category. Only subjects with a proper clearance level and belonging to the specified category are allowed to access the entity. Classifications thus form a (partially ordered) lattice. However, most models based on this combination ignore the category aspect of the classification, and only address the (fully ordered) classification levels when developing the model.

With the underlying models in mind, we can consider the set of subjects S again. The following are examples of subjects (ie elements of S) that may be authorised to access an entity e:

- An object with a clearance level that dominates the sensitivity level of e;
- An object in possession of a capability to access e;
- An object listed in the access control list of e; or
- An acceptable access path (as defined in PCM [Bos89b] or Mizuno and Oldehoeft's extended access control lists [Miz90]) via which a request may reach e.

Whichever underlying model is used, it is often necessary to ensure that one entity is 'more protected' than a second entity, implying that only a subset of the authorised users of the less protected entity is allowed to access the more protected entity. In such a case we will say that the 'more protected' entity has a *higher classification*, even if we are not using explicit levels as the underlying model. We will also say that the classification or clearance of an item *dominates* that of another item, meaning that the first has a higher (or equal) classification or clearance than the second item. To formalise this, let $subj(e)$ be the set of all subjects authorised to access e, with

$$subj : E \rightarrow \mathcal{P}S$$

where $\mathcal{P}S$ is the powerset of S and E the set of all entities that may be accessed (as defined in chapter 7).

Note that the definition of *subj* represents a generalisation of the simple security property of the Bell-LaPadula model [Bel76].

Without defining the sensitivity of an entity formally, we will denote the sensitivity of an entity e by $L(e)$.

Clearly, if all the subjects authorised to access an entity e_1 are also authorised to access an entity e_2, then the sensitivity of e_1 can be no more than the sensitivity of e_2. We will indicate this by writing $L(e_1) \leq L(e_2)$. Formally

$$L(e_1) \leq L(e_2) \Leftrightarrow subj(e_1) \supseteq subj(e_2)$$

Let L_{low} indicate the sensitivity of an entity that may be accessed by all subjects in the system and L_{high} indicate the sensitivity of an entity that may be accessed by no subject in the system. Then, for any entity $e \in E$,

$$L_{low} \leq L(e) < L_{high}$$

Because this inequality holds for any entity, the *least upper bound* and the *greatest lower bound* of any set of sensitivities are defined. (In mathematical terminology, $L(E)$ is a lattice.) We will denote the least upper bound of a sequence of entities $e_1, e_2, e_3, ..., e_n$ by $\lceil e_1, e_2, e_3, ..., e_n \rceil$, and the greatest lower bound by $\lfloor e_1, e_2, e_3, ..., e_n \rfloor$.

The function *subj* returns the subjects that may access a given entity. It is also useful to define an 'inverse' function, indicating which entities may be accessed by a given subject. For any subject $s \in S$ let *ent(s)* be the set of all entities that may be accessed by s. Formally, *ent* is a function

$$ent : S \rightarrow \mathcal{P}E$$

where $\mathcal{P}E$ denotes the powerset of E and, for all $s \in S$

$$ent(s) = \{e \in E | s \in subj(e)\}$$

8.2.2 (X1.2) Protection interpretation

The second aspect regarding labeling semantics is the question exactly what is protected if an item is labeled. Some models attempt to protect the fact that an item exists, while others protect the contents of an item; we will refer to the first category as existence protection and to the second category as access protection. Note that, in the case of access protection, users may know that a variable or method exists. but will get 'access denied' error messages when they try to activate a method or to read or modify an instance variable without authorisation.

Definition 8.1 *In an* existence protected model *the fact that a labeled entity exists is hidden from unauthorised subjects.*

In an existence protected model, if the existence of one entity implies the existence of a second entity, then the sensitivity of the first entity must be at least as high as that of the second—ie

Lemma 1 *In an existence protected model, if*

$$(e_1 \in E) \wedge (e_2 \in E) \wedge (e_3 \in E) \wedge ... \wedge (e_n \in E) \rightarrow f \in E$$

and $s \in subj(E_i)$ *for every* i *then* $s \in subj(f)$.

□

If this were not the case then a subject cleared to access every e_i, but not f, will be able to infer that f exists, contrary to the definition of existence protection. This fact, together with the axioms given earlier, leads to a number of interesting theorems—see later.

To illustrate existence protection, suppose that a class EMPLOYEECLASS has a protected method INCREASESALARY; unauthorised subjects will simply not 'see' this method in the class. Since an unauthorised subject cannot determine that this method exists, it cannot possibly access it. Similarly, this class may have instances JOHN and JAMES. A subject cleared to access JOHN's information, but not that of JAMES, will simply not know that JAMES exists.

One advantage of existence protection is illustrated by the following example: If a subject can see that EMPLOYEECLASS has a GETUNDERCOVERASSIGNMENT method, the user will be able to infer that some employees have undercover assignments even if the user is not able to activate that method.

A disadvantage of existence protection is that a subject that does not know that an object already exists, may create it; in the example above, a subject that does not know about JOHN's existence may insert a JOHN object into the database—if the create request is rejected, the user will infer that the object existed, contrary to the labeling interpretation. This interpretation therefore necessitates polyinstantiation with its associated problems (see section 11.6). A similar problem occurs when a subject defines a class that already exists without the class knowing.

Conjecture 1 *An existence protected model must support polyinstantiation.*

Lunt (amongst others) uses existence protection. Property P3 of SORION also appears to have this objective.

In some cases existence protection is not practical. One example is when sensitivity cannot be pre-determined; for example when it depends on

- Content;
- Context; and/or
- Time.

In the case of content-dependent sensitivity the sensitivity varies according to the content of the item: a salary may be classified *secret* if it is below $100 000, but *top-secret* if it is above that amount. Context-dependent security addresses the aggregation problem: the sensitivity of a combination of entities are often higher than that of the individual entities. As an example, the list of employees in a firm may be unclassified, the list of salaries earned may be *secret*, but the list of employees with the salary each earns may be *top-secret*. An example of time-dependent sensitivity is a military database entry containing the date on which the enemy will be attacked—it will be *top-secret* up to that date but unclassified after that. In these cases existence protection may be meaningless: if a subject is allowed to access a SALARY variable in one object and the SALARY field does not exist in another similar object, it is easy to infer that the particular variable does in fact exist, but is classified. Thuraisingham [Thu89] gives some solutions to this problem, but it still does not solve the problem that the existence of hidden items may be inferred.

Definition 8.2 *In an* access protected model, *an unauthorised subject is not allowed to access a protected entity; 'not allowed to access' means that*

- *Any unauthorised message sent to a protected object will fail;*
- *Any unauthorised message sent to a protected method will fail; and*
- *Any method attempting to access (read or write) an instance variable illegally will fail.*

Note that 'accessing an object' does not include passing that object as a parameter; a message that is not authorised to access an object may pass it as a parameter when it sends a message to another object.

Some underlying models we mention are only aimed at preventing an unauthorised subject from obtaining information from a protected entity (compare the simple security property of Bell-LaPadula [Bel76]). Information flow restrictions (see parameter X3.3 in chapter 10) then restrict the locations where information may be sent or written (compare the *-property of Bell-LaPadula). To accommodate these models, we allow a weak form of access protection where *access* means obtaining information with the aid of a method or *reading* a variable.

Some remarks on message failing: Note that failing does not only occur in access protection: In an existence protected model, an object may receive a message from an authorised subject and activate the corresponding method. If this method now

- Sends a message to an object that does not exist as far as the original subject is concerned;
- Sends a message to another method that does not exist as far as the original subject is concerned; or
- Accesses a variable that does not exist as far as the original subject is concerned

then, in any of these cases, can the message sent by this subject not complete.

For consistency, if a message fails for security reasons, we will assume that it behaves exactly as when it fails in the case where the object, message or variable really does not exist. One possibility is to return a special value *nil* whenever a variable is read that does not exist (or is hidden) and to ignore any attempt to write a value to a non-existent (or hidden) variable. Similarly, messages to such non-existent (or hidden) methods may just be ignored, and the value *nil* may be returned by any such message from which a return value is expected. This solution could be dangerous because errors in the database software could be overlooked; other solutions should be investigated.

Note that neither existence protection, nor access protection implies that the contents of an object can never be accessed by a subject that is not authorised to access the object: in the case of a composite object a constituent part may be independently accessible and have a different (lower) classification; it is then possible that the part may be accessed by subjects not authorised to access the composite object. This may, for example, occur where the relationship between to entities is more sensitive than the contents of the entities themselves. This aspect will be mentioned again in the next section and discussed in the section on aggregation.

Further note that existence and access protection are not the only protection models. Other protection models worth consideration include the following:

- Use existence protection for objects, but hide classes totally from ordinary subjects. Thus no ordinary subject can gain information from the class and, from there, infer information about the instances of the class. In this case the model will not attempt to limit conclusions about the class, but will attempt to limit the inferences one can make about one instance of a class by accessing another instance.

- Use access protection for classes (and therefore do not attempt to hide the structure of objects), but do hide the fact that an instance exists from a subject not authorised to access the instance.

We do not consider these possibilities further in this work.

8.3 Labels and object-oriented databases

This chapter considered the 'format' of the security labels and the intention with which such a label is associated with an entity. In the next chapter these parameters will be used to indicate how the various entities in an object-oriented database can be protected. The implications that specific choices for these two parameters have on the labeling of related entities will also become clear in that chapter.

Chapter 9

Taxonomy: Structural Labeling

An object-oriented system consists of a non-homogeneous set of entities (objects, methods, variables, etc). These entities are related through mechanisms such as inheritance and encapsulation. The three design parameters described in this chapter address the implications that the structure of an object-oriented system has on the protection of the system.

In the previous chapter we considered the way an entity can be protected, and what it means when an entity is protected. However, not all object-oriented systems allow all entities to be protected; the first parameter considered in this chapter deals with the entities a specific model does protect. The second parameter considers the way the initial security labels are assigned for any protectable entity; this initial assignment of labels often stems from the way the system is structured. The third parameter considers the restrictions that the relationships between the entities place on their security labels.

This chapter considers the implications that the (relative static) structure of an object-oriented system has on its protection. The next chapter considers the implications that the (dynamic) activities in an object-oriented system, such as message passing, have on the protection of the system.

O let us love our occupations,
Bless the squire and his relations,
Live upon our daily rations
And always know our proper stations.

Charles Dickens
The Chimes, 2nd Quarter

9.1 Introduction

This chapter considers the influence of the structure of the data on the labeling of entities.

The object-oriented model has a rich variety of entities with relationships between such entities. For example, an object is an *instantiation* of a class; an object may be an *aggregation* or *composition* of other objects; objects *contain* variables and methods; etc. These entities and relationships describe the structure of an object-oriented data model.

The first aspect we consider is the question on which entities may be labeled. Possibilities include objects, classes, methods and instance variables. If entities such as instance variables and methods may be labeled, a finer grain of protection is available; however, in such a case the resulting model may be more complex than a model which only allows objects to be labeled.

The second aspect we consider is the initial assignment of security labels for newly created entities. Often such labels are inherited from the class of the newly created object; a number of alternatives exist.

The third aspect considered in this chapter is the restrictions that exist for the labeling of related entities. It is, for instance, shown that in many cases an instance of a class is at least as sensitive as the class itself. Such a restriction is, of course, related to the first aspect we mentioned: if a class cannot be labeled, the previous restriction does not make sense; however, in such a case an alternative restriction may make sense.

9.2 Structural labeling

We now describe the three concerned parameters:

(X2.1) Protectable entities;

(X2.2) Label instantiation; and

(X2.3) Relationship restrictions.

9.2.1 (X2.1) Protectable entities

A model for a secure object-oriented database must specify which entities may be protected. In this section we consider some possibilities and indicate implications of allowing certain entities to be protected. Examples of protectable entities are objects, methods, instance variables, classes, class methods, class variables, etc.

If only objects are allowed to be labeled, the whole object has the same sensitivity; we will refer to such an object as a *single level object*. If portions of an object (ie methods and instance variables) may be labeled individually, it provides finer granularity from a security viewpoint. Because the sensitivity of portions of such an object may be different, we will refer to such an object as a *multilevel object*.

Models that only support single level objects and models that support multilevel objects have been proposed and both types seem practical [Lun89,Gar90,Lun90a,Gar91]. Models supporting only single level objects have the benefit of being simple; also many of the relationship restrictions given later (X2.3) are trivial in such a case. On the other hand, it seems quite natural to label the query and update methods of the same object differently, because there is an inherent difference between the sensitivities of these two methods. Similarly, it seems natural to label the SURNAME and SALARY instance variables of an EMPLOYEE object differently, also supporting the case for multilevel objects.

Multilevel objects present the *multilevel update problem* [Thu89]: suppose that some variables are classified higher (or merely different) than other variables of the same object. The problem to be answered is at what clearance level must a subject be to update the object. If the subject is at the high level, and writes variables with a lower sensitivity, then the subject has 'written down' possibly compromising security. If the subject is at a lower level, it is not authorised to access the more sensitive variables anymore. The only solutions are to either log out and log in for every concerned sensitivity or to polyinstantiate some variables (or the whole object). As Thuraisingham points out, neither of these solutions are desirable. Note that this problem only occurs when instance variables are allowed to have a different sensitivity than the containing object; methods may differ without any adverse effects.

Classes also have entities that may be protected. Remember that classes are objects themselves; they have methods (known as class methods, such as CREATE), they have variables (known as class variables) and they are instantiations of metaclasses. Therefore any remarks about labeling of objects or their facets also hold for the 'object section' of a class. Note that a class variable must be labeled in the class because it is available to all instances of the class (although the sensitivity in a particular instance may be higher than the sensitivity specified in the class—see X2.3 later). It is also possible to label the 'template section' of classes (including methods and variables defined there) with the intention that the given sensitivity label should apply to all instances of the class, rather than to protect the class itself—we address this below (X2.2).

The protectable entities of SODA are instance variables and objects; however SODA allows either the entire object to be labeled, or the individual instance variables, but not both.

In addition to the usual protectable entities SECDB also allows 'layers' of objects to be labeled: The class of an object may have many superclasses. Portions of an object are therefore defined in a number of (super) classes. These portions form layers—the innermost layer defined in the 'highest' superclass, while the outermost layer is defined in the (immediate) class of the object. Protecting the outermost layer corresponds to protecting the entire object in other models; labeling any other layer protects those portions of the object that were defined in the corresponding superclass and any of its superclasses.

9.2.2 (X2.2) Label instantiation

An object-oriented system is a dynamic system: objects are instantiated and destroyed continually. In order not to compromise security, newly created (instantiated) objects must be protected immediately. The initial sensitivity of an entity reflects the inherent sensitivity of the entity. For example, it can be predetermined which subjects will be allowed to invoke the INCREASESALARY method of an EMPLOYEE object. Normal database activities will have no influence on the sensitivity of this method. Similarly, the inherent sensitivities of the instance variables of such an object may be predetermined reflecting the sensitivity of the value of such a variable or the sensitivity of the relationship between the object and the contents of that variable. However, some models allow the sensitivity of such a variable to be increased dynamically if a particularly sensitive value is stored in that variable. Stated differently, the initial sensitivity of a variable reflects the sensitivity of the 'container'; in some cases the 'contents' of the variable will be more sensitive than the 'container' itself, and at that point the sensitivity of the variable may be higher than its initial sensitivity. Here we are only interested in the initial sensitivity; see ***dynamic labeling*** later for details about relabeling of entities.

Three primary possibilities exist for determining the initial sensitivity of an object.

1. The class must be labeled and the label(s) specified for the class must apply for all instances of the class;

2. Every object (and possibly its variables and methods) must be explicitly labeled when or after the object is instantiated; [Var91] proposes

that the method that instantiates a new object may specify the sensitivities of the interface variables (parameters) from which the class of the new object may then derive the sensitivities of all instance variables; or

3. Constraints may be specified—ie separate (logic) rules that determine the sensitivity of a newly instantiated object and then ensures that the entity is sensitivity labeled immediately.

Of course, a combination is also possible with default labels derived from the class and individual labels given after instantiation where the default labels do not suffice.

Since it is unreasonable to trust normal methods to sensitivity label a newly instantiated object, and since constraints fall outside the scope of this work, the mechanism that will be used here for labeling of newly instantiated objects is inheritance: mechanism (1) above. SODA and SECDB use this mechanism.

9.2.3 (X2.3) Relationship restrictions

The third parameter described in this chapter—on the labeling restrictions of related entities—leads to some interesting results. The relationships to consider are:

- *Composition*: object – facet (name, instance variable, method);
- *Instantiation*: class – object;
- *Inheritance*: class – subclass; and
- *Data structure membership*: data structure (such as a list) – member of the data structure; also members amongst themselves.

Relationship restrictions may be divided into *compulsory* and *additional* restrictions. Compulsory relationship restrictions are those restrictions that a model must enforce as a result of design choices made elsewhere or as a result of the inherent object-oriented structure. Additional relationship restrictions are other restrictions a model may prescribe because they simplify the model or have some other benefit. We discuss compulsory relationship restrictions first.

The first relationship restrictions we consider, are imposed by the *composition* of an object. An object encapsulates everything inside it. This implies that a facet thus encapsulated cannot be accessed by a subject that is not allowed to access the encapsulating object in the first place. This holds whether the model uses existence protection or access protection.

Lemma 2 (Encapsulation corollary) *The sensitivity of a facet of an object dominates the sensitivity of the object itself, ie* $L(\langle x,o\rangle) \geq L(o)$ *for every facet* x *of any object* o*. Similarly, the sensitivity of a facet of a class dominates the sensitivity of the class itself, ie* $L(\langle x,c\rangle) \geq L(c)$ *for every facet* x *of any class* c*.*

Proof: Clearly, if a subject is not authorised to access o, that subject is also not authorised to access any facet x of o. Formally,

$$\begin{aligned} & s \notin subj(o) \rightarrow s \notin subj(\langle x,o\rangle) \\ \Rightarrow\ & s \in subj(\langle x,o\rangle) \rightarrow s \in subj(o) \\ \Rightarrow\ & subj(\langle x,o\rangle) \subseteq subj(o) \\ \Rightarrow\ & L(\langle x,o\rangle) \geq L(o) \end{aligned}$$

The proof for the second part of the lemma is similar. □

This lemma is similar to the *facet property* (property 3) of Lunt.

Our second group of relationship restrictions are imposed by instantiation: the relationships that exist between a class and its instances.

If the model uses existence protection, both inheritance and instantiation restrict labeling: If (part of) a class is existence protected, that (part of the) class does not exist as far as an unauthorised subject is concerned. However, if such an unauthorised subject is authorised to access a subclass or an instance of the protected class, this protected information becomes visible to the subject (even if not accessible by the subject): from the subclass the subject can 'see' the names of methods, instance variables and even the composition of the superclass; similar information can be gathered from an instance about the concerned class. This motivates the next theorems.

Lemma 3 *In an existence protected model, the sensitivity of an instance must dominate the sensitivity of its class, ie*

$$(\forall o \in U)[L(o) \geq L(class(o))]$$

Also, the sensitivity of a facet in an instance must dominate the sensitivity of the facet in the class and also dominate the sensitivity of the instance itself, ie

$$(\forall o \in U)(\forall x \in o)[L(\langle x,o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil]$$

Proof: From axiom 1 we know that $o \in U \rightarrow class(o) \in C$. Since $U \subseteq E$ and $C \subseteq E$, it follows that, for every $o \in U$,

$$o \in E \rightarrow class(o) \in E$$

Applying lemma 1 proves that $L(o) \geq L(class(o))$.

From axiom 2 $\langle x; o\rangle \in E \rightarrow \langle x, class(o)\rangle \in E$. Applying lemma 1 to this proves $L(\langle x, o\rangle) \geq L(\langle x, class(o)\rangle)$. From lemma 2 follows that $L(\langle x, o\rangle) \geq L(o)$ and therefore

$$(\forall o \in U)(\forall x \in o)[L(\langle x, o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

□

From the discussion on label instantiation above (X2.2) we may assume that similar restrictions exist for the access protected model. We state this as an axiom.

Axiom 6 *In an access protected model, if a class (including variables and methods defined in that class) is sensitivity labeled, the intention is that such labels should apply for all instantiations of the class; in other words, the sensitivity of an instance must dominate the sensitivity of its class; and the sensitivity of any facet of an instance should dominate the sensitivity of that facet as defined in the class.*

From lemma 3 and axiom 6 above we have

Theorem 4 (Instantiation restriction) *The sensitivity of an instance must dominate the sensitivity of its class, ie*

$$(\forall o \in U)[L(o) \geq L(class(o))]$$

Property 2 of Lunt (when interpreted for classes and instances) and property P7 of SORION are the same as our instantiation restriction (theorem 4).

Theorem 4 constrained the sensitivity of an instance of a class. The following theorems constrain the sensitivity of a facet of such an instance.

Theorem 5 *In an existence protected model, the sensitivity of a facet x of an object o is given by*

$$L(\langle x, o\rangle) = \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

Proof: Let $c = class(o)$. From lemma 3, $L(\langle x,o\rangle) \geq \lceil L(\langle x,c\rangle), L(o)\rceil$. Thus $subj(\langle x,o\rangle) \subseteq subj(\langle x,c\rangle) \cap subj(o)$. Let $s \in subj(\langle x,c\rangle) \cap subj(o)$. Then $s \in subj(\langle x,c\rangle)$ and $s \in subj(o)$. From axiom 3, since $o \in U$ and $c = class(o)$ and $x \in c$, the fact that $x \in o$ is implied. According to lemma 1, $s \in subj(\langle x,o\rangle)$. Therefore $subj(o) \cap subj(\langle x,o\rangle) \subseteq subj(\langle x,c\rangle)$. As indicated earlier, $subj(\langle x,o\rangle) \subseteq subj(\langle x,c\rangle) \cap subj(o)$. Therefore $subj(\langle x,o\rangle) = subj(\langle x,c\rangle) \cap subj(o)$, and thus

$$L(\langle x,o\rangle) = \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

□

The practical implication of this theorem is given in the following corollary.

Corollary 6 *In an existence protected model, the sensitivity of a facet x of an object o may only be changed from its inherited sensitivity*

$$L(\langle x, class(o)\rangle)$$

by increasing the sensitivity of the entire object o.

□

Theorem 7 *In an access protected model, the sensitivity of a facet x of an object o dominates both the sensitivity of the facet in its class and the sensitivity of the object itself, ie*

$$L(\langle x,o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

Proof: From lemma 2, we know that $L(\langle x,o\rangle) \geq L(o)$. From axiom 6 follows that $L(\langle x,o\rangle) \geq L(\langle x, class(o)\rangle)$. □

Our next group of relationship restrictions are imposed by inheritance: the relationships that exist between superclasses and their subclasses.

Lemma 8 *In an existence protected model, the sensitivity of a subclass must dominate the sensitivity of its superclass(es), ie for any class $c \in C$*

$$(\forall d \in sup(c))[L(c) \geq L(d)]$$

Proof: This proof is similar to the proof of the first part of lemma 3. □

Axiom 7 *In an access protected model, if a class (including variables and methods defined in that class) is sensitivity labeled, the intention is that such labels should be inherited by all its subclasses (unless the variable or method is redefined, in which case it may be relabeled); in other words, the sensitivity of a subclass must dominate the sensitivity of its superclass(es); and the sensitivity of any facet* inherited *from a superclass should dominate the sensitivity of that facet in the superclass.*

Note that this axiom does not apply to SECDB: if a class is labeled in SECDB, the intention is that that label should be applied to the corresponding layer of any instance of that class or of any instance of some (eventual) subclass of the labeled class. Results based on this axiom will thus not hold for SECDB.

From lemma 8 and axiom 7 above we have

Theorem 9 (Inheritance restriction) *The sensitivity of any subclass must dominate the sensitivity of its superclass(es), ie for every class $c \in U$*

$$(\forall d \in sup(c))[L(c) \geq L(d)]$$

□

Property 2 of Lunt (when interpreted for superclasses and subclasses) and property P9 of SORION are the same as our inheritance restriction (theorem 9).

Lemma 10 *In an existence protected model, if any class $c \in U$ inherits a facet x from a superclass $d' \in sup(c)$ or redefines a facet x that occurs in a superclass d, then*

$$L(\langle x, c\rangle) \geq \lceil L(\langle x, d'\rangle), L(c)\rceil$$

Proof: This proof is similar to the proof of the second part of the lemma 3. □

Property 4 of Lunt (when interpreted for superclasses and subclasses) is the same as lemma 10.

Theorem 11 *In an access protected model, if any class $c \in U$* inherits *a facet x from a superclass $d' \in sup(c)$, then*

$$L(\langle x, c\rangle) \geq \lceil L(\langle x, d'\rangle), L(c)\rceil$$

Proof: This proof is similar to the proof of the second part of the lemma 4. □

Multiple inheritance is supported if the data model allows a class to, simultaneously, be a subclass of more than one superclass. In general, if multiple inheritance is supported it is necessary to specify which facet (variable or method) will be inherited if it is defined in more than one superclass. In the case of existence protection problems may occur: Assume that a subject is cleared to access facet F in class C1. Assume further that class C3 is a subclass of both classes C1 and C2. If the subject does not see the facet F in C3 it implies that facet F is inherited from C2. The subject can therefore infer that facet F exists in both C2 and C3, although the subject is not cleared to know about the facet's existence. This is addressed by the following theorem.

Theorem 12 (Facet inheritance restriction) *If, in an existence protected model, x is a facet of a class c and x also appears in (at least) one superclass of c, then the sensitivity of $\langle x, c\rangle$ is bounded as follows, for every superclass $d \in sup(c)$ that has a facet x*

$$L(c) \leq L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$$

Further, if x is inherited from a specific superclass $d' \in sup(c)$, then, for every superclass $d \in sup(c)$ that has a facet x

$$\lceil L(\langle x, d'\rangle), L(c)\rceil \leq L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$$

Proof: The fact that $L(c) \leq L(\langle x, c\rangle)$ was stated in lemma 2. The fact that, if x is inherited from a specific superclass d', then $\lceil L(\langle x, d'\rangle), L(c)\rceil \leq L(\langle x, c\rangle)$ was stated in lemma 10.

To prove the rest of the inequality, ie to prove

$$L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$$

select any $d \in sup(c)$ such that $x \in d$. Let $s \in subj(\langle x, d\rangle) \cap subj(c)$. From axiom 5 and lemma 1 follows that $s \in subj(\langle x, c\rangle)$. Therefore $subj(\langle x, d\rangle) \cap subj(c) \subseteq subj(\langle x, c\rangle)$ and thus $L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$. □

This theorem makes a number of statements about inherited (and redefined) facets in an existence protected model. Some of these statements are given in the next corollary.

Corollary 13 *In an existence protected model*

1. *If a class c* inherits *a facet $\langle x, c\rangle$, the sensitivity of $\langle x, c\rangle$ may only be different from its sensitivity in the superclass when $L(\langle x, c\rangle) = L(c)$; in other words, the only way to increase the sensitivity of an inherited facet $\langle x, c\rangle$, is by increasing the sensitivity of the entire class c. (Corollary 6 made a similar remark about instances.)*

2. *The sensitivity of a* redefined *facet $\langle x,c\rangle$ must be dominated by that of every like-named facet in any superclass of c, whenever the sensitivity of the like-named facet dominates the sensitivity of c.*

3. *If a facet is defined in only one superclass of a given class c (or the class c has only one superclass, or the object-oriented model only allows single inheritance), that facet may be inherited without any problems; the sensitivity of the inherited facet will be the least upper bound of its sensitivity in the superclass and the sensitivity of the class c.*

4. *The sensitivity of an inherited facet $\langle x,c\rangle$ must be dominated by the sensitivity of all like-named facets in superclasses of c, whenever the sensitivity of the like-named facet dominates the sensitivity of c.* □

The last point of the corollary above indicates two strategies an existence protected model may follow to prevent the problems presented by multiple inheritance:

1. Ensure that the like-named facet with the lowest sensitivity is always inherited; or

2. Ensure that the sensitivity of the class is an upper bound for the sensitivities of all concerned facets in superclasses.

Strategy 1 is only feasible if a facet with a lowest sensitivity does indeed exist; if the sensitivities are partially ordered the existence of such a facet is not guaranteed. Of course, a model may also solve these problems by disallowing multiple inheritance.

Property 5 of Lunt requires that the facet with the lowest sensitivity must be inherited—ie strategy 1 above. Properties P15 and P16 of SORION require that the facet with the highest sensitivity must be inherited—contradictory with this theorem.

We conclude our discussion of compulsory restrictions with a few remarks about restrictions imposed by data structure membership. In an array-like structure it is usually possible to infer the structure of one element from that of another. This indicates that all members of the array must have the same sensitivity in an existence protected model. Of course, if the elements of such a data structure are not necessarily homogeneous, this requirement may be dropped. We do not address the sensitivity of elements of data structures in detail in this chapter, but give one last example: property P4 of SORION specifies that the sensitivity of a 'set object' is the least upper bound of the sensitivities of the element objects.

The restrictions on labeling of related entities described above are necessary because security will be compromised if the restrictions are not enforced; some models have additional restrictions because they simplify the model or enhance security in some other way. We briefly look at one of these. Some models (eg Lunt property 1 and SORION property P2) specify that basic objects (or system objects) must have a system-low sensitivity (L_{low}). If a model includes such a specification, the model has to specify exactly what is meant by basic objects or system objects. If such a requirement is not included, the system security officer and/or the database system has to define the sensitivity of all objects supplied with the database.

Theorem 4 :	$L(o) \geq L(class(o))$ for every object $o \in U$
Theorem 5 :	$L(\langle x, o\rangle) = \lceil L(\langle x, class(o)\rangle), L(o)\rceil$ for every object $o \in U$ and every facet x of o
Theorem 9 :	$L(c) \geq L(d)$ for every class $c \in C$ and every superclass $d \in sup(c)$
Theorem 12 :	$L(c) \leq L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$ for every class $c \in C$ and every superclass $d \in sup(c)$ of c that has a facet x; $\lceil L(\langle x, d'\rangle), L(c)\rceil \leq L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$ if c inherited the facet x from $d' \in sup(c)$

Table 9.1: Relationship restrictions for an existence protected model.

Theorem 4	$L(o) \geq L(class(o))$ for every object $o \in U$
Theorem 7	$L(\langle x, o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil$ for every object $o \in U$ and every facet x of o
Theorem 9	$L(c) \geq L(d)$ for every class $c \in C$ and every superclass $d \in sup(c)$
Theorem 11	$L(\langle x, c\rangle) \geq \lceil L(\langle x, d'\rangle), L(c)\rceil$ where $c \in C$ is any class and x a facet that c inherited from a superclass $d' \in sup(c)$

Table 9.2: Relationship restrictions for an access protected model.

Property 1	$L(c) = L_{low}$ for every system-defined class c $L(o) = L_{low}$ for every system-defined object o
Property 2	$L(c) \geq L(d)$ for every class c and every superclass $d \in sup(c)$ $L(o) \geq L(class(o))$ for every object o
Property 3	$L(\langle x, c\rangle) \geq L(c)$ for every facet x of any class c $L(\langle x, o\rangle) \geq L(o)$ for every facet x of any object o
Property 4	$L(\langle x, c\rangle) \geq L(\langle x, d'\rangle)$ for every facet x of any class c where $\langle x, c\rangle$ is inherited from a superclass $d' \in sup(c)$ $L(\langle x, o\rangle) \geq L(\langle x, class(o)\rangle)$ for every facet x of any object o
Property 5	$L(\langle x, d'\rangle) \leq L(\langle x, d\rangle)$ for every $d \in sup(c)$ whenever any class c inherits a facet x from a superclass $d' \in sup(c)$
Property 7	$L(c) \geq L(\langle x, d\rangle) \rightarrow L(\langle x, c\rangle) \leq L(c)$ for every superclass $d \in sup(c)$ of any class c that has a facet x $L(o) \geq L(\langle x, class(o)\rangle) \rightarrow L(\langle x, o\rangle) \leq L(o)$ for every facet x of any object o

Table 9.3: Relationship restrictions imposed by the Lunt model.

The relationship restrictions imposed on existence protected models are summarised in table 9.1. The restrictions imposed on access protected models are summarised in table 9.2.

The relevant restrictions from the Lunt model (in our notation) are given in table 9.3. Property 6 does not appear in that table because it deals with run-time access restrictions, which we discuss under the heading *dynamic labeling* later—see equations 10.8 and 10.9. Also note that, since Lunt does not distinguish between a subclass and an instance of an object, we have interpreted most of the Lunt properties for subclasses and for instances. Remember that the Lunt model uses existence protection and, therefore, table 9.3 has to be compared to table 9.1. Lunt's property 2 is identical to our theorems 4 and 9. Nothing in [Lun90a] necessitates Lunt's property 1, and therefore it falls in the *additional relationship restriction* category. Lunt's properties 3, 4 and 7 are easily derived from our theorems 5 and 12, but the converse is not true. Lunt's property 5 is stricter than necessary; for those situations where it is required, it can be derived from our theorem

12.

9.3 Beyond the static structure

It is clear from this chapter that the structure of an object-oriented system has definite implications for the protection of entities in the system. This structure is relatively static: Occasionally entities may be created and deleted, influencing the structure; during normal operation of the system the structure will remain mostly unchanged.

However, an object-oriented system is all but a passive system. The fact that code (or methods, or behaviour) is encapsulated with the data to form an integrated system is an indication that quite the opposite is true. We consider the implications of this dynamic nature of the system in the next chapter.

Chapter 10

Taxonomy: Dynamic Labeling

This chapter discusses the third and last group of design parameters considered in this taxonomy. This group of parameters are intended to ensure that secrecy is not compromised by the (dynamic) activities occurring in an object-oriented database.

This last group consists of three parameters: Accessing various entities in the database may have implications on the access rights of the subject; the first parameter considers such implications. The second parameter specifies the way the sensitivity of a message is determined—a message exists only for a short span of time and the rules to label it therefore differs from the rules used to label the (relatively) permanent entities. The third parameter has to ensure that information does not flow from a more sensitive location to a less sensitive location (where it can be accessed by a subject that is not supposed to access it).

This chapter completes the formal part of the taxonomy. Chapter 11 discusses a number of aspects that may be considered for a future taxonomy; at present they have not received enough attention in the literature to be included in the current taxonomy. Appendix A contains a summary of the proposed taxonomy, while appendix B illustrates the taxonomy by way of a number of examples.

Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.

Robert Frost
The Road Not Taken

So it is in travelling; a man must carry knowledge with him, if he would bring home knowledge.

Samuel Johnson

Let's find out what everyone is doing.
And then stop everyone from doing it.

A. P. Herbert
Let's Stop Somebody

10.1 Introduction

This chapter concerns itself with the flow of authorisation (X3.1), the flow of sensitive data (X3.2) and the restrictions on such data flows to ensure secrecy despite such flows (X3.3). Restrictions based on flow of authorisation and flow of information that a model introduces to ensure that security will not be compromised represent a generalisation of the *-property of the Bell-LaPadula model [Bel76].

10.2 Notation

The dynamic activities in a system may be modeled by the following simple productions:

$$\Sigma \rightarrow M \tag{10.1}$$
$$M \rightarrow a_i T \tag{10.2}$$
$$T \rightarrow MT \tag{10.3}$$
$$T \rightarrow r \tag{10.4}$$
$$T \rightarrow q \tag{10.5}$$

Here Σ represents the *primary accessor*, in other words the (probably human) object that sends the original message to the database. The nonterminal M represents a message; the production $\Sigma \rightarrow M$ (production 10.1) models the message sent by the primary accessor.

A message causes a method (of a specific object) to be activated; in the productions above a_i represents such an active object (or, more precisely, method-object pair $\langle m, o \rangle$). The 'task(s)' such an active method performs are represented by T. The production $M \rightarrow a_i T$ (production 10.2) indicates that a specific method (a_i) is activated on receipt of a message, after which that method 'executes' a list of tasks T.

Production 10.3, $T \rightarrow MT$, represents the case where a task T consists of sending a message, before handling the next task; production 10.4, $T \rightarrow r$, represents the task of terminating execution of the active method and sending a reply to the calling method, while production 10.5, $T \rightarrow q$, represents the task of terminating execution of the active method and returning control to its calling method without sending a reply to the calling method.

In order to model access to instance variables, the following two productions have to be added to those given above:

$$M \rightarrow px_i \tag{10.6}$$

$$M \rightarrow gx_i \tag{10.7}$$

These productions model accessing variables by sending messages to them: Production 10.6, $M \rightarrow px_i$, models writing ('putting') a variable x_i (where $x_i = \langle v, o\rangle$ for some object o and some variable $v \in o$). Similarly production 10.7, $M \rightarrow gx_i$, models reading ('getting') such a variable x_i.

Note that the intention of all these productions is not to generate strings, but to reflect events in the system.

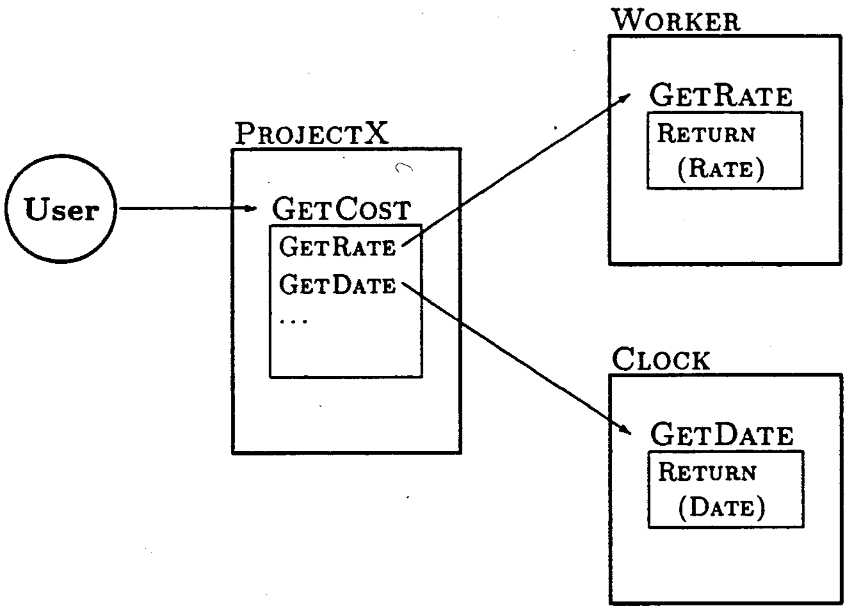

Figure 10.1: A sequence of messages

Example 10.1 *Consider the events depicted in figure 10.1. Table 10.1 lists the events that occur in this figure, the sentential string before the event and the production that accompanies the event. The non-terminal that will be replaced in each case is underlined.*

10.3 Dynamic labeling

Authorisation flow deals with the question whether and how the clearance of a subject is influenced by the method activations acting on its behalf—see X3.1 below.

Event	Sentential string	Production
The primary accessor sends the message GETCOST to PROJECTX.	$\underline{\Sigma}$	$\Sigma \rightarrow M$ (10.1)
Method GETCOST is activated; assume that a_1 represents GETCOST	$\underline{M}$	$M \rightarrow a_1T$ (10.2)
GETCOST has to send a message, followed by other tasks	$a_1\underline{T}$	$T \rightarrow MT$ (10.3)
GETCOST's first message is GETRATE (a_2)	$a_1\underline{M}T$	$M \rightarrow a_2T$ (10.2)
GETRATE has to send a message, followed by other tasks	$a_1a_2\underline{T}T$	$T \rightarrow MT$ (10.3)
GETRATE sends a 'read message' to variable RATE	$a_1a_2\underline{M}TT$	$M \rightarrow gx_1$ (10.7)
GETRATE's final task is to return the value	$a_1a_2gx_1\underline{T}T$	$T \rightarrow r$ (10.4)
GETCOST again has to send a message, followed by other tasks	$a_1a_2gx_1r\underline{T}$	$T \rightarrow MT$ (10.3)
GETCOST's second message is GETDATE (a_3)	$a_1a_2gx_1r\underline{M}T$	$M \rightarrow a_3T$ (10.2)
...	$a_1a_2gx_1ra_3\underline{T}T$	...

Table 10.1: Productions for the events of figure 10.1

Information flow deals with the flow of sensitive information through the system (as messages) and, more particularly, the restrictions that a model may enforce to ensure that such information does not flow to somewhere where it is less protected.

As long as an object moves as a unit from one location to another location in the system (probably as a parameter of a message) information will not be compromised: The object will still have its original sensitivity label; similarly, the methods and variables of the object will still have their original sensitivity labels. As an example, suppose that a SALARY object has a PRINTSALARY method. A subject that obtains SALARY from the EMPLOYEE object will still not be able to invoke the PRINTSALARY method if it is not authorised to: even though it obtained the SALARY it cannot do anything with it. The normal encapsulation feature of object-orientation, combined with sensitivity labels, provides very natural protection.

However if, in the example above, the salary was not encapsulated in a SALARY object, but rather stored as a normal real number, the value would have no natural protection once it leaves the EMPLOYEE object. A similar problem occurs anywhere where an object provides methods that return values of instance variables—when the value leaves the protection of the encapsulated object, security may be compromised.

In the final part of this section we consider restrictions that may be dynamically applied to *information flow* through the system to ensure that, even if such information is removed from the encapsulated object, it will still not be exposed to unauthorised access. We assume that information retrieved from a variable or object is as sensitive as the label for that variable or object indicates and that such information must stay at least as sensitive wherever it might flow in the system. One cannot reasonably make this assumption about information obtained via a sensitive method because another (much less sensitive) method may return the same information. To avoid such inconsistencies, we only consider restrictions for information flowing from a sensitive variable or sensitive object. One possibility to include the sensitivity of methods when considering information flow, is to ensure that any variable is at least as sensitive as the greatest lower bound of any method that has access to the variable. We do not investigate this option in the current work.

Under dynamic labeling we consider three aspects:

(X3.1) Messages act on behalf of a subject and therefore the clearance of the message depends on that of the subject. A model has to specify how the clearance (or authorisation) of a message is determined.

(X3.2) Messages also carry information—this information may be sensitive, requiring labels. A model has to specify how the sensitivity of a message is determined.

(X3.3) If some of the sensitive information contained in a message is stored in variables of the receiving object, it must be ensured that an unauthorised subject cannot now access the information in this object. This can be ensured by either relabeling the object or variable with a suitable label (if the existing labels are not suitable) or by disallowing information to be saved if the existing labels are not suitable. The model should indicate any flow restrictions and any conventions for relabeling.

10.3.1 (X3.1) Authorisation flow

Consider the productions of section 10.2 above. Which objects should be taken into account when the clearance of a message is determined? The following possibilities exist:

1. Consider only the *primary accessor* (represented by Σ);
2. Consider all method activations that have been activated since the original message was sent, but ignore those that have already returned control to their calling methods (ie consider all *active objects*), including the primary accessor; or
3. Consider all objects on the *access path* of the request, ie every method activation a_i that has been activated since the original message was sent, whether that method has terminated execution or not, including the primary accessor.

In SODA the primary accessor determines the clearance of a message (possibility 1 above).

Property 6 of the Lunt model states (in our notation)

$$L(s) \geq L(\langle m, o\rangle) \geq L(o) \tag{10.8}$$
$$L(M) = L(s) \tag{10.9}$$

where $s \in S$ is any subject, $\langle m, o\rangle$ is a method of object o that s wants to execute and M is a message sent by s to this effect. It seems from this property is that Lunt denotes the *clearance* of a subject s by $L(s)$; if the clearance of the subject dominates the sensitivity of the method to be activated, the subject is allowed to activate it (10.8). Further, the message M has exactly the same *clearance* that its sending subject had (10.9). It therefore seems that this property indicates that authorisation in the Lunt model depends only on the primary accessor (possibility 1 above).

SECDB uses the complete access path to determine authorisation (possibility 3 above); however, an individual profile is free to ignore any part of the access path when it determines whether access should be granted. As an example why consideration of the entire access path might be useful, consider a stock control system, where the INVENTORY object may only be asked to issue stock (decrease its current stock level) if a message have previously been sent to the CHECKCREDIT method of the requesting object.

In practice, clearances are assigned as follows for the three possibilities listed above:

Primary accessor: All messages sent (by any method of any object) on behalf of the primary accessor have the same clearance—that of the primary accessor;

Active objects: All messages sent by a method of an object will have the same clearance for a given activation of the method; that clearance depends on the clearance of the object-method pair and on the clearance of the message that activated the method; and

Access path: The clearance of any message sent depends on all objects and methods involved anywhere previously in the request.

The clearance of messages are influenced by other objects involved in the request if either possibility 2 or 3 above is used. As an example, a model may require that the clearance of a message is not higher than the clearance of any object involved in the request thus far. To generalise this concept, we introduce semantic actions to associate a clearance attribute $M.c$ with every message sent. These rules are given in tables 10.2, 10.3 and 10.4. We also associate a sensitivity attribute $e.l$ with every protected entity e. The possible values of these attributes depend on the underlying model (X1.1): in the case of explicit levels $e.l$ may be an integer indicating the sensitivity of entity e. (Compare this to $L(e)$ which is an abstract indication of the sensitivity level of entity e and not directly related to the underlying model). We give examples of attribute values for the various underlying models later; for the time being we assume that it is possible to determine whether message M is permitted to access entity e given the attributes $M.c$ and $e.l$. $\Sigma.c$ denotes the clearance of the primary accessor, $a_i.c$ denotes the clearance of an individual method activation and the function *clear* maps the clearance of a message and a method activation to their combined clearance. The use of the $.c$ attributes should be clear after studying tables 10.2, 10.3 and 10.4; the use $e.l$ will be illustrated later.

Table 10.2 illustrates the situation where message clearance is only based on the primary accessor. In this case, the message sent by the primary accessor gets its clearance attribute from the primary accessor ($M.c := \Sigma.c$), every method activation sends its messages with the clearance of the message that activated it ($T.c := M.c$) and where a task consists of sending a message followed by another task, both the message and the other task have the same clearance as the original task. In summary, every aspect 'inherits' its clearance from the primary accessor.

Table 10.3 illustrates the case where message clearance is based on all active objects. Here a message gets its clearance attribute from the primary accessor that sent it ($M.c := \Sigma.c$) or from the task of which it is part

Event	Semantic rule
$\Sigma \rightarrow M$	$M.c := \Sigma.c$
$M \rightarrow a_i T$	$T.c := M.c$
$M \rightarrow px_i$	
$M \rightarrow gx_i$	
$T \rightarrow MT_1$	$M.c := T.c$ $T_1.c := T.c$
$T \rightarrow r$	
$T \rightarrow q$	

Table 10.2: Semantic rules to determine message clearance based on the primary accessor

Event	Semantic rule
$\Sigma \rightarrow M$	$M.c := \Sigma.c$
$M \rightarrow a_i T$	$T.c := clear(M.c, a_i.c)$
$M \rightarrow px_i$	
$M \rightarrow gx_i$	
$T \rightarrow MT_1$	$M.c := T.c$ $T_1.c := T.c$
$T \rightarrow r$	
$T \rightarrow q$	

Table 10.3: Semantic rules to determine message clearance based on active objects

($M.c := T.c$); also if a task consists of a message followed by another task, the second task has the same clearance as the original task ($T_1.c := T.c$).

The only situation where the clearance changes, is where a new method is activated ($M \rightarrow a_i T$): Here the clearance of the task performed by the new method activation (ie all messages sent by it) will depend on the clearance of the message M and the clearance of the new method activation a_i. The function *clear* is used to determine the clearance of the messages to be sent; this clearance depends on the clearance attribute of the received message and that of the method activation: $T.c := clear(M.c, a_i.c)$. Exactly how *clear* combines these two attributes depends on the particular model; we give some examples later.

Event	Semantic rule
$\Sigma \rightarrow M$	$M.c := \Sigma.c$
$M \rightarrow a_i T$	$T.c := clear(M.c, a_i.c)$. $M.r := T.r$
$M \rightarrow px_i$	$M.r := M.c$
$M \rightarrow gx_i$	$M.r := M.c$
$T \rightarrow MT_1$	$M.c := T.c$ $T_1.c := M.r$ $T.r := T_1.r$
$T \rightarrow r$	$T.r := T.c$
$T \rightarrow q$	$T.r := T.c$

Table 10.4: Semantic rules to determine message clearance based on the access path

Table 10.4 indicates how a model operates that uses the entire access path to determine the clearance of a message. It introduces another attribute $.r$; it is used to take the influence of an activity on the clearance of subsequent messages into account when the activity terminates (or returns to its caller). For example, if a task consists of a message followed by another task ($T \rightarrow MT_1$) the clearance attribute of the second task depends on the clearance 'returned' by the method which has just terminated ($T_1.c := M.r$). Note that when a method terminates ($T \rightarrow r$ or $T \rightarrow q$), the clearance with which it 'returns' is the last clearance the task had ($T.r := T.c$). We do not discuss determining the clearance of messages based on the entire access path further here; an example of it using SECDB will be given later in this section.

From the preceding discussion, it is clear that any secure database model must not only specify which approach it uses to determine the authorisation of a message, it also has to specify up to four additional pieces of information:

- The format of the clearance attributes ($\Sigma.c$, $a_i.c$ and $M.c$);
- The format of the sensitivity attribute ($e.l$);
- For which values of $M.c$ and $e.l$ message M may access e; and
- How clearances are combined, ie define *clear*.

Having given rules to determine the clearance of a message we will now consider how the clearance attribute may be used to specify when a given message is authorised to access an entity. As stated earlier, this depends not only on the clearance attribute of the message, but also on the sensitivity attribute of the entity.

The entities that need protection are methods and instance variables. (The object-oriented model does not allow direct access to other entities at all.)

It is easy to attach semantic actions to the productions $M \rightarrow a_iT$, $M \rightarrow px_i$ and $M \rightarrow gx_i$ (productions 10.2, 10.6 and 10.7) to let the message fail when access rights are insufficient; table 10.5 contains an example of a complete specification of authorisation flow for a model based on explicit levels and using the complete access path to determine the clearance of a message. Note the semantic rules to fail a message if its clearance attribute is not sufficient (compared to the sensitivity attribute of the entity to be accessed) to allow the message to proceed. Remember that these semantic rules will be different if an alternative underlying model is used.

Some concrete examples of the security attributes denoting the clearance of a subject and the sensitivity of an entity are now given. Note that specific models may differ from these examples.

Explicit levels: The clearance attribute will typically be an integer indicating the clearance level of the primary accessor ($\Sigma.c$), individual method ($a_i.c$) or a message ($M.c$). The sensitivity attribute $e.l$ for an entity e will also be an integer. A message M will be allowed to access an entity e if $M.c \geq e.l$. The combination of two clearance levels may be the greatest lower bound of the two, ie $clear(M.c, a_i.c) = \lfloor M.c, a_i.c \rfloor$.

Access control lists: The clearance attribute will typically be the identity of the primary accessor ($\Sigma.c$), the identity of the individual

Event	Semantic rule
$\Sigma \rightarrow M$	$M.c := \Sigma.c$
$M \rightarrow a_i T$	$T.c := \lfloor a_i.c, M.c \rfloor$
	$M.r := T.r$
	if $M.c < a_i.l$ then fail
$M \rightarrow px_i$	$M.r := M.l$
	if $M.c < x_i.l$ then fail
$M \rightarrow gx_i$	$M.r := M.l$
	if $M.c < x_i.l$ then fail
$T \rightarrow MT_1$	$M.c := T.c$
	$T_1.c := M.r$
	$T.r := T_1.r$
$T \rightarrow r$	$T.r := T.c$
$T \rightarrow q$	$T.r := T.c$

Table 10.5: Example of access checking rules for a specific model

method ($a_i.c$) or the collected identities associated with the message ($M.c$). The sensitivity attribute $e.l$ will be an access control list, giving the identities of subjects permitted to access the entity. (The access control list may be given explicitly as a list; SECDB, for example, gives it as a formal grammar.) A message M will be allowed to access an entity e if the identities collected in $M.c$ appear in the access control list $e.l$. The combination of two clearance attributes may be the concatenation of the two, ie $clear(M.c, a_i.c) = M.c || a_i.c$ where $||$ denotes concatenation.

Capabilities: The clearance attribute may contain the capabilities presented by the primary accessor ($\Sigma.c$), the capabilities presented by the individual method ($a_i.c$) or the capabilities collected by the message ($M.c$). The sensitivity attribute $e.l$ may contain the capability necessary to access entity e. A message M will be allowed to access an entity e if the capability $e.l$ appears in the list of capabilities $M.c$ presented by M. The combination of two clearance attributes may be the union of the two, ie $clear(M.c, a_i.c) = M.c \cup \{a_i.c\}$.

As an example, SECDB builds a string representing the access path which is subsequently used to verify access restrictions. Therefore the 'clearance' of the primary accessor ($\Sigma.c$) and the methods ($a_i.c$) are *baggage* *vectors*, representing

1. The identity of the object;
2. The method;
3. The domain (processor) on which the object resides; and
4. The integrity state (or mode) the domain uses while executing the method.

These vectors are concatenated to form strings; the clearance combination function *clear* concatenates its arguments, ie $clear(M.c, a_i.c) = M.c||a_i.c$. The sensitivity of any entity $e \in E$, represented by $e.l$, is a *profile*. The profile is given as a formal grammar; M may access e only if $M.c$ is derivable in the grammar associated with $e.l$.

Although the informal notion of the set of subjects S used thus far is sufficient for our purposes, we are now in a position to formalise it: a subject is any sentential form ending with an M obtainable by a leftmost derivation from Σ using the productions from section 10.2 above:

$$S = \{\omega M \in V^* | \Sigma \overset{*}{\underset{lm}{\Longrightarrow}} \omega M\}$$

where V represents the alphabet of the grammar. This definition is only given for completeness; it is not used again in this work.

10.3.2 (X3.2) Sensitivity flow

A message carries information. Some of that information may be explicit values in the form of parameters. The mere fact that a message has been sent, is implicit information about the contents of the database. This information—both explicit and implicit—might be sensitive. Nobody can access this information while it is carried by the message; however, the receiver of the message may assign the information to an instance variable where it is possible to access it. It is therefore necessary to keep track of the sensitivity of information contained by a message. We will use the notation $L(M)$ to indicate the sensitivity of any message M with the same meaning that $L(e)$ has for any entity e. Later (see X3.3) we will discuss how the current sensitivity of a message will be used to prevent information from being assigned to a variable that is not as least as protected as the information contained by the message.

The following rules may be used to keep track of a the sensitivity of a message:

1. The initial message from the primary accessor has low sensitivity (L_{low}) because it does not contain any information;

2. A method starts execution with the sensitivity of the message that activated it;

3. The sensitivity of the method activation is adjusted whenever it receives a reply after sending a message—the sensitivity is adjusted to the least upper bound of its current sensitivity and the sensitivity of the reply;

4. All messages sent by a method activation has the sensitivity currently associated with the activation; whenever the sensitivity of the method activation is adjusted, all subsequent messages will be sent at the new sensitivity level; and

5. If the method sends a reply to its calling method, the reply has the last 'current' sensitivity of the method activation.

For the purposes of sensitivity flow, access to variables is modeled in terms of messages: writing to a variable is done by sending a message to the variable and reading by sending a message and receiving a response.

The rules are formalised in table 10.6. The suffix $.l$ again represents the sensitivity of an entity; for example $M.l$ is an attribute representing the sensitivity of message M. The suffix $.u$ is used to keep track of the sensitivity of information received by a method that was returned as the result of a message. The value l_{low} is the attribute value that corresponds to the lowest sensitivity L_{low}.

These rules correspond largely to rules 2.1 to 2.4 of SODA. SODA further requires (in its rule 2.3) that the current sensitivity of a method activation must be increased when a value is written (or "added to a polyinstantiated set") at a higher sensitivity level. This restriction is not necessary: any messages that will be rejected because of this restriction subsequent to such a write may be avoided by reordering the statements (messages) in the concerned methods. Of course, nothing prevents a model from including such a specification; it may be included by changing the semantic rule for $M \rightarrow px_i$ to $M.u := x_i.l'$ where $x_i.l'$ denotes the initial sensitivity (see X2.2) of variable x_i.

Since Lunt does not distinguish between sensitivity levels and clearance levels, Lunt's property 6 (see equation 10.9 above) probably also indicates that the sensitivity of a message is the same as the 'classification level' (clearance) of the subject that sent it. This 'classification level' of a message is only defined in that model and then never referred to again.

Event	Semantic rule
$\Sigma \rightarrow M$	$M.l := l_{low}$
$M \rightarrow a_iT$	$T.l := \lceil a_i.l, M.l \rceil$
	$M.u := \lceil M.l, T.u \rceil$
$M \rightarrow px_i$	$M.u := l_{low}$
$M \rightarrow gx_i$	$M.u := x_i.l$
$T \rightarrow MT_1$	$M.l := T.l$
	$T_1.l := M.u$
	$T.u := T_1.u$
$T \rightarrow r$	$T.u := T.l$
$T \rightarrow q$	$T.u := L_{low}$

Table 10.6: Semantic rules for message sensitivity

An alternative to the rules given above come from [Var91]: They suggest that it is possible for any method to start execution at the system-low sensitivity (see our rule 2 above). This may compromise security: method m_1 may access a sensitive variable and based on its value decide to send a message to method m_2; if m_2 does not access any variable and send a (non-sensitive) message to m_3 then m_3 may assign a value an unclassified variable. If it is known that this sequence of events will only occur if the sensitive value accessed by m_1 has a specific (range of) values, it is possible to infer whether that variable does indeed have such a value by examining the unclassified variable written by m_3, contrary to the information flow restrictions discussed later (X3.3). Of course, if the composition of methods is not public knowledge (and cannot be inferred) this strategy is viable.

Note that some secure database models partition the set of entities E such that any subject $s \in S$ is allowed to access all the entities in at most one partition E_i of E, that is, if $s_1 \in S$ and $s_2 \in S$ then either $ent(s_1) = ent(s_2)$ or $ent(s_1) \cap ent(s_2) = \emptyset$. Further, these models do not transform the subject because of authorisation flow (X3.1) or, if it does transform a subject s_i into a subject s_{i+1} when a new method joins the composite subject, it ensures that $ent(s_i) = ent(s_{i+1})$. In such a model a subject cannot cause information to flow to a less protected location and it is unnecessary to keep track of the sensitivities of messages or to restrict information flow based on the sensitivity of the information. A model with this property is said to support *single level subjects*.

10.3.3 (X3.3) Information flow restrictions

Information contained by a message cannot be accessed directly. However, an object that receives a message, may store the received data in instance variables. It is then possible that other objects may obtain the information from this object (by sending messages to this object). Therefore it must be ensured that sensitive information is not stored in variables that may be accessed by subjects not authorised to access the sensitive information. This can be done by either ensuring that the variable(s) storing the sensitive information may only be accessed by properly cleared subjects, or by ensuring that the entire object may only be accessed by properly cleared subjects.

Suppose that a message containing sensitive information arrives at an object and that the activated method attempts to write information to a variable. If the sensitivity of either the concerned object or concerned variable dominates the sensitivity of the received message no problem occurs: even if sensitive information is stored in the receiving object will it not be compromised because it is at least as protected as in its original location. In other words, any message M, accessing any entity e, must be allowed to proceed if $L(M) \leq L(e)$—and, of course, if the subject represented by M is in $subj(e)$.

Apart from allowing the message to access the protected entity when the sensitivity of the entity dominates the sensitivity of the message, the model may also decrease the sensitivity of the variable (or object) if its sensitivity strictly dominates that of the message. If the sensitivity is decreased, it must still dominate that of the message. Further, the sensitivity of the variable (or object) must never be decreased below its initial (inherent) sensitivity (see X2.2).

If the sensitivity of the object or variable to be accessed does not dominate the sensitivity of the received message, security may be compromised. So, if the message attempts to modify the state of the object, protective steps must be taken. The following possible strategies exist:

1. Reject the message;

2. Increase the sensitivity of the receiving variable or object; or

3. Polyinstantiate the receiving variable or object.

To formalise the previous remarks we will associate semantic actions with the production

$$M \rightarrow px_i$$

because information will only be compromised if information is actually written to a variable; information temporarily 'inside' an object if a message is just 'passing through' without modifying the state of the object will not compromise information. We identify four possible behaviours when a message attempts writing to a variable:

Proceed/Reject: If the sensitivity of the message strictly dominates that of the variable, reject the write; otherwise the protection of the variable is adequate and the write is allowed to proceed. This is accomplished by associating the following semantic action with the production $M \rightarrow px_i$:

$$\text{if } L(M) > L(x_i) \text{ then fail}$$

Increase sensitivity: Here the sensitivity of a variable (or its containing object) is increased if the information to be written is too sensitive for the current sensitivity of the variable. If the sensitivity is increased, it is increased to that of the message; this may be done by either setting $x_i.l$ or $o.l$ (where $x_i = \langle v, o\rangle$) to $M.l$. Formally the production $M \rightarrow px_i$ has the semantic action

$$\text{if } L(M) > L(x_i) \text{ then } x_i.l := M.l$$

If the level of the entire object is to be adjusted, the *then* part of the semantic action has to be changed:

$$\text{if } L(M) > L(x_i) \text{ then } o.l := M.l, \text{ where } x_i = \langle v, o\rangle$$

Note that we are changing the level of a 'terminal symbol'—on a real system it means that the actual security attributes of the related entity must be updated.

Modify sensitivity: Here the sensitivity of the variable is adjusted to reflect the sensitivity of the new value stored in it. This means that the sensitivity of the variable may be increased or decreased. The only provisos are that the sensitivity of the variable must never be decreased below its inherent (initial—see X2.2) sensitivity, or below the sensitivity of its containing class (see lemma 2). Put differently, the sensitivity of a variable $\langle v, o\rangle$ being written to by a message M, is changed to $\lceil L(M), L(o), L'(\langle v, o\rangle)\rceil$ where $L'(\langle v, o\rangle)$ is the initial sensitivity of $\langle v, o\rangle$. Formally, $M \rightarrow px_i$ takes the semantic action

```
if $L(M) > \lceil L(o_j), L'(x_i)\rceil$ then $x_i.l := M.l$
elseif $L(o) > L'(x_i)$ then $x_i.l := o.l$
else $x_i.l := x_i.l'$
```

where $x_i = \langle v, o \rangle$, $L'(x_i)$ represents the initial sensitivity of x_i and $x_i.l'$ is the corresponding sensitivity attribute.

Note that in the case of structured variables (for example array-like structures) the sensitivity of the entire variable cannot be decreased if a single low-sensitivity element is written. This is not a problem if structured variables are objects constructed from the individual 'element' variables. In such a case only the relevant element variable will be relabeled.

Polyinstantiate: See section 11.6 for a discussion of polyinstantiation.

SECDB takes the option of *increasing* the sensitivity of the receiving object or variable. In SECDB the security profiles associated with the message are attached to the concerned variable. In effect this means that the variable is relabeled with the least upper bound of its original sensitivity and the sensitivity of the message—ie the sensitivity of the variable is 'increased' so that it dominates the sensitivity of the active message.

SODA uses polyinstantiation here—see section 11.6 for a discussion of polyinstantiation. SODA also specifies a highest sensitivity that may be associated with any given variable; if the sensitivity of the message exceeds this maximum, the message will also fail (but in this case the database will inform the user that the message failed). Formally, if the maximum sensitivity for a variable x_i is $L_{max}(x_i)$, the semantic action

$$\text{if } L(M) > L_{max}(x_i) \text{ then fail} \tag{10.10}$$

associated with $M \to px_i$ will ensure this.

We stated earlier that either the sensitivity of the concerned variable or the concerned object must be increased if the *increase sensitivity* strategy is used. Corollary 6 restricts strategy 2 for existence protected models: Since the sensitivity of a variable may only be increased by increasing the sensitivity of the entire object, it is not possible to relabel variables; the containing object will have to be relabeled. In an access protected model, it is possible to relabel either the modified variable or the containing object; it seems that it is better to relabel the variable, since it minimises the risk that an (authorised) user attempting to access another part of the object, will be denied, only because this variable contains sensitive information. SECDB therefore relabels the variable. SODA provides another solution by allowing either the entire object or its instance variables (but not both) to be labeled; in such a model the already labeled entity will simply be relabeled.

10.4 Conclusion

This concludes the description the taxonomy. The next chapter discusses a number of aspects that may be considered for a future taxonomy—aspects that, at present, have not received enough attention to be included in the current taxonomy. Appendix A contains a summary of the proposed taxonomy, while appendix B illustrates the taxonomy by way of a number of examples.

Chapter 11

Taxonomy: Remaining Issues

A taxonomy for secure object-oriented database models such as the one described in chapters 7 to 10 cannot attempt to be the final word on classification of such databases since new models are presently being proposed regularly. Although most of the new proposals focus on one or more of the aspects covered by the taxonomy, some proposals do indeed investigate new areas. It is conceivable that such a new model could introduce an aspect that should indeed be included in a taxonomy. This chapter concludes the discussion of the taxonomy by considering a number of these aspects.

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill
Mansion House, 10 Nov.1942

11.1 Other design parameters

In this chapter we briefly look at aggregation, implementation, discretionary access controls and integrity constraints—all aspects that need consideration when one defines a secure database model. However, too few models have addressed them to see any clear alternatives. Further, some of these issues, such as implementation, are not primary design alternatives, but must rather be used to evaluate a given model. Polyinstantiation is an aspect of secure object-oriented databases that may influence the entire described taxonomy; polyinstantiation is discussed in section 11.6.

11.2 Aggregation

Aggregation is a security problem because the sensitivity of a number of pieces of information combined is often higher than the sensitivity of the individual pieces.

A related problem, is the problem of inference. Often, if it is possible to obtain some portions of information, it is possible to infer the missing (sensitive) portions. One common example occurs where it is possible to obtain the average salary of a set of employees—if it is possible to make that set very small (say only one employee) the salary of an employee is easily obtainable.

The encapsulation facility of object-orientation provides a natural way to sensitivity label the relationship between objects higher than the individual objects: If an object is classified, the composition of that object is hidden from unauthorised subjects; ie no unauthorised subject will be able to access the instance variables of the object, and therefore such a subject will not know about the relationship between them which is encapsulated in the object. Similarly, if an instance variable is labeled, the fact that the contents of that variable is related to the rest of the object is protected, although the contents of that variable may be unclassified. See [Lun90a] for a discussion. Suitably labeled methods can also be used to protect information regarding relationships.

11.3 Implementation

Most models do not address implementation. However, an impressive model that is difficult to implement is not worth much. Here we briefly address some implementation considerations; a thorough study is necessary before detailed comments can be made.

As always one should strive to make security related code as small as possible—a small section of code is easier to verify, increasing trust and reliability. This may be done by building the database on a trusted computing base (TCB); SeaView [Lun90b] is an example of a secure relational database making use of a TCB. The database partitions information according to its sensitivity and stores it in entities (usually files) managed and protected by the TCB. If the TCB has already been verified, this provides a quick, reliable way to implement a secure database. It also makes it possible to port a secure database from one computer to another, if the other supports a comparable TCB.

Lunt [Lun90a] suggested that a secure object-oriented database can be built on top of a trusted relational database. Much work has been done in the field of trusted relational databases which makes this option attractive. Such a database will probably map classes to relations, objects to tuples and instance variables to tuple entries. However, we feel that protection of methods is the most exciting possibility offered by object-oriented databases and, sadly, nothing exists in the relational model to which methods may directly be mapped.

One interesting possibility is a database where the checking may be done statically (in other words, before run-time; compare [Var91]). Consider a secure database model that

- Is based on explicit levels, and uses the same label to indicate the clearance and the sensitivity of any labeled entity (X1.1);
- Labels (at least) methods and variables (X2.1);
- Uses the entire access path to determine the authorisation of a message (X3.1) where the combination of two security attributes (*clear*) is the greatest lower bound of the two;
- Only allows a message to activate a method if the security attribute of the message dominates that of the method (X3.1);
- Only allows a method to read a variable if the security attribute of the method dominates that of the variable (X3.1); and
- Only allows such a method to write to a variable if the security attribute of the variable dominates that of the method (X3.3).

In such a model the sensitivity of any message (X3.2) is the same as the clearance (X3.1) of such a message as computed by the rules in table 10.6. It is therefore not necessary for such a model to keep explicit track of the sensitivity of a message. Obviously such a model employs the *proceed/reject* option to restrict information flow (X3.3).

The described model will not relabel variables or objects since a value will never be written to a variable that is not already sensitive enough. Therefore the security level of any method will dominate that of all variables that it *may potentially* read (and not only those that it actually reads during a given activation); further the security labels of all variables that *may potentially* be written to will dominate the label of the method. This makes it possible to statically check the security of any object that is added to the database. In this case an underlying monitor is not necessary to check access to sensitive information at run-time. It is only necessary to ensure that the primary accessor (the human operator) only activates methods this accessor is entitled to—that is

- Methods that have a higher security level than the accessor if the method will not send a reply to the accessor; or
- Methods with the same security level as the accessor if the method will send a reply to the accessor.

It is relatively simple to implement a view mechanism that will only allow an accessor to 'see' methods the accessor is entitled to access. In such a model it is only necessary to trust the message passing mechanism; if this mechanism does nothing other than passing messages we can trust the database. However, it needs to be shown that this model is flexible enough for practical use.

The reader is referred to [Laf90] for a discussion of the classical secure database implementation strategies.

11.4 Discretionary access controls

Discretionary access controls refer to the rights subjects have to access entities; especially the *discretionary* power they have to decide what restrictions should apply for entities under their control and the power to grant rights to other subjects. For instance, the 'owner' or creator of a file often has the power to grant other users the right to access the file; similarly such an 'owner' may have the power to revoke the access rights of other users to the file.

In a database, discretionary access rights are of less concern: a database contains information shared by many users and usually owned by the enterprise. However, a scenario is possible where a central database administrator or system security officer is not responsible for the security of all information contained in the database; in stead a number of 'guardians' may be appointed to be responsible for various sections of the database. Mechanisms enable the existence of such 'owners' of information may be worth investigation; this includes investigation of mechanisms to verify their actions.

Often access control lists or capabilities are used to implement discretionary access control. Note that we included those facilities in our treatment of mandatory security. Chapter 12 contains a proposal for a discretionary security model for object-oriented databases.

11.5 Integrity constraints

Ensuring the integrity of information has two aspects:

- Ensuring that an unauthorised subject does not modify information; and
- Ensuring that an authorised subject does not modify information to unacceptable values such as values inconsistent with other entries in the database.

Although we do not address the latter aspect in this work, it is worth pointing out that some models for secure databases also have definite integrity benefits in that area.

One such example is security models that include the whole access path and not only the primary accessor when access rights are determined. In such a model it is possible to ensure that an (authorised) accessor may not access an object without following an 'authorised' path. For example, if a LOCATION object consists of a LONGITUDE and a LATITUDE object, it is easy for such a model to ensure that LONGITUDE and LATITUDE are only updated if the request is routed via LOCATION; if the update method of LOCATION sends messages to both the update method of LONGITUDE and the update method of LATITUDE, it is impossible to update only one constituent object, leaving the database in an inconsistent state. An inconsistent state may easily result if a subject is allowed to directly send a message to update, say, LATITUDE.

Note that the described models do not solve the integrity problems that arise if an 'impossible' value is inserted into the database by an (authorised) subject; for example if a person's age is given the value 999 years. This may be solved by including integrity constraints—logic rules constraining what values may be inserted into the database.

11.6 Polyinstantiation

Thuraisingham [Thu89] described the many faces of polyinstantiation in an object-oriented system:

- Different subjects may see differing values (or contents) in the same object;
- Different subjects may see differing (class) structures for the same object;
- Different subjects may see a different set of methods supported by the same object; and/or
- A single method may have a different definition for different subjects.

Abstractly, one may view a database that supports polyinstantiation as a set of databases that co-exist—one database for every sensitivity level. Rules then describe which 'databases' must be updated whenever an update request is received. Similarly, rules describe from what database a query request must be answered. The axioms and theorems given earlier may be adapted, based on this view of polyinstantiation. The affected productions given while discussing authorisation flow (X3.1) are $M \rightarrow a_i T$, $M \rightarrow px_i$ and $M \rightarrow gx_i$. In the case of polyinstantiation more than one a_i or x_i may exist. In the first case rules must be given to determine to which method a_i the message must be directed to based on $M.c$. In the second case rules must be given to determine which set of variables x_i (often more than one, but possibly none) must receive the new value. In the third case, rules must determine from which variable x_i the read must be attempted; in some cases more than one value is read and the set of values returned. The functions *class* and *sup* and the corresponding axioms about object-orientation also need adaptation: the class and superclass may depend on which subject wants to know. The rules for determining the initial sensitivity of a newly created object may also have to be adapted. We do not elaborate at this time.

Clearly, polyinstantiation can cause integrity problems: a user who is not cleared to access a value, may insert it into the database; if the new value differs from the existing one, it is a good question which value should be viewed as the *correct* value—ie the more sensitive value or the more recent value?

In an access protected model, polyinstantiation is not necessary. In that case it is possible to introduce a weaker form of polyinstantiation, such as that suggested by SECDB: Protected entities (primarily methods) may be replicated in a cascaded fashion. When a request arrives at such a protected entity, the appropriate access checks are performed; if access is granted, the request proceeds by accessing the corresponding entity; if access is denied, access to the next 'polyinstantiated' entity in the chain will be requested. This process will be continued, until either an entity is found that allows access, or until the chain of polyinstantiated entities has been exhausted, in which case the request must be aborted. This form of polyinstantiation enables *ad hoc* entities to be polyinstantiated; it is possible to apply it without the integrity problems associated with 'full' polyinstantiation; however, it is not strong enough to support an existence protected model. As an example where weak polyinstantiation may be useful, consider some political candidate's database where the VOTESFORUS object may have a polyinstantiated method NUMBEREXPECTED; this method may return different values depending on whether it is invoked by the candidate, by the candidate's sponsor or by the candidate's press secretary.

Another 'weak' form of polyinstantiation is the view concept as defined by [Shi89] where multiple interfaces may be defined for a single object. As it is described in [Shi89], other 'client' objects may decide through which view it wants to access the object; however, when it is intended as a form of polyinstantiation, security constraints may specify through which view another object accesses the multiview object.

11.7 Conclusion

This chapter indicated a number of ways the taxonomy may be extended. No doubt, more possibilities exist. However, listing them at this time will serve little purpose; we rather leave it to the reader to consider the possibilities we mentioned, along with any other possibilities, in order to identify those that deserve attention. If this work prompts someone to address an aspect of secure object-oriented databases that has been overlooked until now, this work has served a purpose.

Chapter 12

A Discretionary Security Model

This chapter proposes an initial discretionary security model for object-oriented databases. The purpose of the model is to indicate how results obtained in the taxonomy of chapters 7 to 11 may be utilised when a new security model is defined.

Entities in the database are protected by capabilities. A subject that possesses a capability is authorised to access the corresponding entity. Additionally, under certain conditions, a subject may pass the capability on to another subject, authorising this other subject to access the protected entity. Passing the capability on to another subject is done at the first subject's discretion, hence the term *discretionary* security. We consider the restrictions that apply in such a database to the granting and revoking of capabilities.

Property has its duties as well as its rights

Thomas Drummond
Letter to the Earl of Donoughmore,
22 May 1838

12.1 Introduction

In this chapter we propose a model for discretionary security in an object-oriented database. We refer to the proposed model as DISCO (*DISCretionary Object-oriented security model*).

DISCO uses capabilities to protect entities. A capability is an unforgeable identifier authorising the possessor to access the corresponding entity. Under certain conditions (for example when the possessor of the capability is the 'owner' of the protected entity) the possessor may grant access to the protected entity to other subjects; this is done by giving them (a copy of) the capability for the protected entity. Similarly, access to an entity may be revoked by 'taking back' the capability.

The intention of our model is to study the implications that stem from passing a capability to another subject and to identify the restrictions that should apply to such transferring of capabilities. These restrictions follow from results obtained in the taxonomy.

Very little has been published about discretionary security models for object-oriented databases; [Dit89] is one example, but they do not address the complete object-oriented model. Much more has been published about mandatory security for object-oriented databases—see chapter 7 for examples.

12.2 Capability-based protection

A capability is an unforgeable token enabling the possessor to access a related entity in some way. It is well known in operating systems; see for example [Sal74].

A subject can, under certain conditions, transfer or propagate a capability to another subject, enabling the other subject to access the protected entity. If the capability is propagated the original subject will still have a copy and, therefore, may still access the entity. If the capability is transferred, the original subject "gives it away" to the new subject. The right to transfer or propagate capabilities is a right that does not exist automatically: Capabilities may be viewed as entities themselves and be protected by, say, other capabilities. In such a case a subject needs a 'transfer' or 'propagate' capability to be able to transfer or propagate a specific capability to another subject. Alternatively (or additionally) a specific subject may 'own' a given entity; such ownership may entitle the subject to propagate capabilities to other subjects.

See chapter 2 for an introduction to capabilities.

12.3 The new model

DISCO uses capabilities to protect entities. A possessor of a capability is authorised to access the corresponding entity. In DISCO a subject that does not possess a capability is not supposed to know (or even infer) that the corresponding entity exists.

In this section we will use an employee database of some organisation to illustrate the concepts. We will assume that the personnel manager of that organisation is the 'owner' of all employee objects; this personnel manager therefore has the right to pass capabilities for the various entities contained in the database to other subjects.

12.3.1 Protecting the entities

DISCO enables classes, objects, variables and methods to be protected.

If an object is protected, a subject may not access the object at all if it does not possess a capability for the object; if there is a JOHN object in the example database, only those subjects that possess a capability for this object may access it. Remember that the only way to access an object in an object-oriented system is by sending a message to it; this means that any message sent to the JOHN object without the corresponding capability will be rejected.

Methods and variables may be protected to provide a finer protection granularity: If the INCREASESALARY method of an employee object is protected it means that a subject needs the corresponding capability to invoke this method. (In addition the subject may require another capability to access the employee object in the first place.) In our example the personnel manager may be the owner of this method (in addition to being the owner of all employee objects).

Initially this personnel manager may be the only subject authorised to send a message to this method. However, for the duration of the annual salary review period, the manager may delegate the salary increase function to a personnel clerk; this is then reflected in the database by passing the capabilities for the INCREASESALARY methods to the personnel clerk. At the end of the review period, the personnel manager will revoke this capability from the personnel clerk. This example illustrates the power of being able to protect individual methods of an object. In older database models one is only able to distinguish between a number of primitive operations such as reading and writing; even in [Dit89] (for a 'structurally object-oriented database') the methods are divided into *read, write, delete* and *exist* classes—a subject authorised to access such a class of operations is authorised to invoke *every* method in that class.

The provision to protect instance variables is intended as an additional safeguard that sensitive information is not accidentally disclosed. For instance, the SALARY variable of the employee objects may be protected; this means that a subject will not be able to access the salary variable via *any* method if that subject does not possess the capability for SALARY; this prevents another subject from inadvertently passing a capability for such a method without also explicitly passing a capability for SALARY.

As indicated, DISCO also allows classes to be protected. Remember that a class is also an object, an instance of some metaclass. The class may therefore be protected just like an ordinary object with the aforementioned mechanisms. However, a class also contains descriptive information (a template or schema) for all instances of the class. This template (or a specific template for a method or instance variable) may also be sensitive and may have to be protected; for this reason DISCO allows classes (and parts of classes) to be protected with capabilities.

12.3.2 Creating and owning entities

To complete the initial picture of DISCO we have to describe how the protection for an entity is established, in other words how a subject specifies that a capability will in future be required for accessing the entity. DISCO uses the following approach: Whenever a new object is instantiated, the subject that instantiated the object becomes the owner of that object; the system creates a new capability, protects the entire object with it and passes the capability to the owner. The owner may then

- Request the system to generate a new capability and use it to protect a method, variable or even the entire object;

- Request the system to use a specified capability to protect a method, variable or the entire object;

- Request the system to 'unprotect' an entity; in other words, to make the entity available for use to all subjects; or

- Request the system to transfer ownership of an entity to another subject.

Note that the creator of a new object is initially the only possessor of a capability to access the object. It is therefore able to set up the access restrictions for any parts of the object any way it wants to without any other subject being able to interfere; similarly, it can do any other initialisation work, before it allows any other subject to access the object.

In the cases where an entity is protected by a new capability, the old capability will be deleted if it is not used to protect any other entity.

DISCO will automatically pass ownership of any entity that is protected with a specified (existing) capability to the subject that owns the other entities protected by that capability. Further, if a subject transfers ownership of an entity to another subject, it must simultaneously transfer ownership of all other entities protected by the same capability. This means that at any point all entities protected by the same capability will have the same owner.

To illustrate the way these rules may be used, consider the employee database again. The CREATE method of the employee class may include code to accomplish the following:

- Protect all the methods and variables of the object that need special protection (such as INCREASESALARY); these facets will probably be protected with the same capabilities that are used to protect the corresponding facets of other employee objects;

- Reprotect the entire object with the same capability that is used to protect other employee objects; and

- Thereby pass ownership of the new object to the personnel manager (since the personnel manager owns all the other employee objects protected by this capability).

The code just presented assumes that all employee objects require the same protection; ie that the same capability may be used to access any employee object; similarly, the same capability may be used to invoke the INCREAS-ESALARY method of any employee object. In many cases this is the proper solution: our earlier example where the personnel manager delegated the salary increase function to the personnel clerk is best reflected in the database if the manager only has to (temporarily) give the clerk the *one* capability that may be used to increase every employee's salary.

The code further makes it possible that not only the personnel manager can instantiate new employee objects: if the personnel clerk has a capability to invoke the CREATE method of the employee class, the clerk is perfectly capable of instantiating new employee objects; however, the personnel manager automatically becomes the owner (ie the one with the discretionary power) of any employee object that is created—it is not possible for anyone else to instantiate employee objects for 'private use'.

Note that the discussion above also holds for newly created classes: only those subjects with appropriate capabilities will be able to send messages to the metaclasses to create new classes. The creator of the class will own the class and be able to pass capabilities to access the class to subjects it chooses. Also, the creator of the class will be able to specify the initial security actions in the CREATE method of the new class, exactly as was the case in the employee class above.

12.3.3 Deleting entities

Two major approaches exist to delete objects in an object-oriented system. The first option is to delete the objects explicitly via an appropriate method. The other option (used by Smalltalk [Gol83]) is to 'garbage collect' any objects that have no remaining references to them.

In a DISCO database different parts of an object may have different owners: one subject may own the JOHN object, while another may own INCREASESALARY of JOHN. To sidestep the question of which owners are allowed to delete such an object and how permission is asked from other owners involved (if permission is indeed to be asked for deletion), DISCO does not supply a facility for explicit deletion of objects; they will automatically be deleted once they cannot be referenced anymore. However, it is possible that this solution—which is acceptable in the programming language field—is not sufficient for databases.

12.4 Transferring capabilities

In DISCO only the owner of an entity can pass capabilities for his entities to other subjects. Similarly, only the owner can revoke rights from other subjects. However, the owner is not free to grant and revoke rights to any subject at any time—the model has to limit these powers. These restrictions are discussed in this section.

12.4.1 Granting rights

A model for a secure object-oriented database may specify security restrictions that must exist between related entities in the database—see chapter 9. DISCO does not introduce any new restrictions; therefore only the normal restrictions for models such as DISCO apply: a summary of these restrictions are given in table 12.1—see chapter 9 for details about these restrictions. If these restrictions are violated some subjects will be able to make inferences about entities they are not allowed to access (and according to our definition of DISCO an unauthorised subject is not even supposed to know that such an entity exists).

E1:	$L(o) \geq L(class(o))$ for every object o
E2:	$L(\langle x,o\rangle) = \lceil L(\langle x, class(o)\rangle), L(o)\rceil$ for every object o and every facet x of o
E3:	$L(c) \geq L(d)$ for every class c and every superclass d of c
E4:	$L(c) \leq L(\langle x,c\rangle) \leq \lceil L(\langle x,d\rangle), L(c)\rceil$ for every class c and every superclass d of c that has a facet x; $\lceil L(\langle x,d'\rangle), L(c)\rceil \leq L(\langle x,c\rangle) \leq \lceil L(\langle x,d\rangle), L(c)\rceil$ if c inherited the facet x from a superclass d' of c

Table 12.1: Relationship restrictions for DISCO.

In this table the notation $L(e_1) \geq L(e_2)$ means that entity e_1 is more sensitive than entity e_2 or, in terms of capabilities, every subject that has a capability to access e_1 must also have a capability to access e_2. As an example, rule E1 specifies that a capability to access an object o may only be passed to a subject that already possesses a capability to access the class of o; otherwise the subject will be able to access the object o and make unauthorised inferences about the class of o. Similarly, according to rule E3, a capability to access a subclass c may only be passed to a subject that already possesses capabilities to access the superclasses of c.

The notation $\lceil l_1, l_2\rceil$ in table 12.1 is used to indicate the least upper bound of l_1 and l_2. Thus rule E2 states that a subject may only possess a capability to access a facet (ie a method or an instance variable) of an object o if it both possesses a capability to access o itself and possesses a capability to access the definition of the method or variable in the class of o. The other implication of rule E2 is even more interesting: A subject that possesses a capability to access an object o and a capability to access a method or variable x in the class of o must necessarily possess a capability to access x in the object o. To illustrate this, suppose that subject s has a capability to access INCREASESALARY in the EMPLOYEE class. Suppose further that JOHN is an instance of EMPLOYEE. If we pass a capability to access JOHN to s, s will possess capabilities to access JOHN and to access INCREASESALARY of EMPLOYEE but not to access INCREASESALARY of JOHN—contrary to rule E2. Similar problems occur if we pass a capability to access INCREASESALARY of JOHN to s before we pass a capability to access JOHN. We therefore need a facility to transfer the capabilities for JOHN and INCREASESALARY of JOHN *simultaneously*.

Rule E2 restricts the passing of capabilities for instances of a class; rule E4 has similar implications for passing capabilities for subclasses of a class. However all four rules have an influence on one another and should be considered together to determine their influence on the passing of capabilities. When one does indeed consider all four rules together, passing of capabilities becomes very involved and it is necessary to pass great numbers of capabilities simultaneously. A better solution is to make the following assumptions:

A1 Force the system to check that the subject has the capability for the concerned object as well as the capability for the method or instance variable before allowing access to the method or variable (alternatively it would have been necessary to check for at most one capability for every access); and further

A2 Make it impossible for a subject to determine in which classes and objects any given method or variable is available, even if the subject possesses a capability for the method or variable; of course, if the subject already possesses a capability for both the object (or class) and method (or variable) there is no harm in allowing the subject to determine that the method (or variable) is indeed available in the object (or class).

If these two assumptions are enforced, it is only necessary to ensure the following:

- If a capability is passed that will allow a subject access to an object o then

  G1.1 The subject must already possess a capability for the class of object o; and

  G1.2 If the subject possesses capabilities for facets (methods or variables) of o then the subject must already possess a capability for that facet in the class of o;

- If a capability is passed that will allow a subject access to a class c then

  G2.1 The subject must already possess capabilities for all the superclasses of the class c (and all their superclasses etc); and

G2.2 If the subject possesses capabilities for facets (methods or variables) of c that also exist in a superclass of c, then the subject must already possess capabilities for that facet in all the superclasses of c up to the level where that facet was first defined;

- If a capability is passed that will allow a subject access to a facet x (method or variable) of an object o then

  G3.1 If the subject possesses a capability for object o, it must already possess a capability to access the facet x in the class of o;

- If a capability is passed that will allow a subject access to a facet x (method or variable) of a class c then

  G4.1 If the subject possesses a capability for class c, and if x is inherited from a superclass d' then the subject must already possess a capability to access the facet x in the superclass of d' of c;

  G4.2 If the subject possesses a capability for a subclass of c then the subject must already possess a capability for the facet x of that subclass;

  G4.3 If the subject possesses a capability for an instance of c then the subject must already possess a capability to access x in the instance.

Rule G1.1 is a direct consequence of rule E1; similarly rule G2.1 follows directly from rule E3: in these cases the intention is to prevent unauthorised inferences about the class or superclass of the concerned entity. Rule G1.2 may be motivated as follows: If a subject s did not have a capability to access an object o but had one to access a facet x of o, it was not permitted to access x anyway—see assumption A1. Since it now received permission to access o it is now in a position to use its capability to access x; if rule G1.2 is not ensured, s may be able to make the unauthorised inference that x is available in the class of o. Note that rule G1.2 and assumption A1 together cater for all the restrictions stated in rule E2. Rule G2.2 is similarly motivated; compare it to rule E4. Rules G3.1, G4.1, G4.2 and G4.3 are derived from rules E2 and E4.

To illustrate these rules consider figure 12.1—it is an extension of the employee database discussed earlier. The EMPLOYEE class has two subclasses: BLUECOLLAR and WHITECOLLAR. JOHN is an instance of BLUECOLLAR and JAMES is an instance of WHITECOLLAR. Suppose that the owner of the JOHN object wants to enable a subject s to access JOHN; it cannot pass the capability for JOHN to s if s does not already possess a capability to access BLUECOLLAR (G1.1); if the owner of JOHN is also the owner of BLUECOLLAR it can easily first pass a capability for BLUECOLLAR to s (but only if s already possesses a capability for EMPLOYEE; G2.1). This illustrates two points:

- A sequence of steps may be necessary to transfer a capability; and
- Sometimes the right of an owner to transfer capabilities may be restricted: if the BLUECOLLAR class is owned by s_1 and JOHN by s_2 then s_2 may only transfer capabilities for JOHN to those subjects that received capabilities for BLUECOLLAR from s_1.

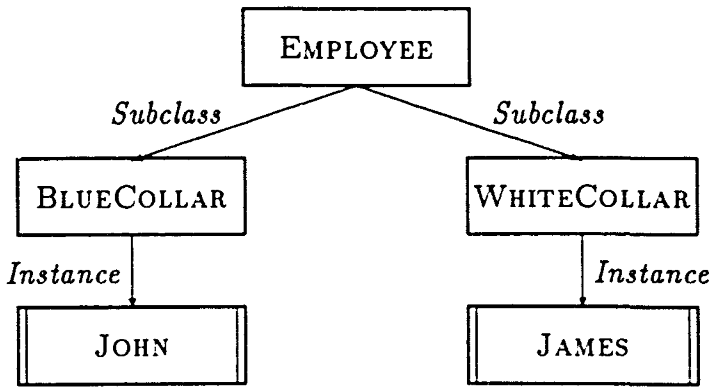

Figure 12.1: Example of a database

As a further example, suppose that s_1 is the owner of INCREASESALARY in JOHN and that s_1 wants to pass a capability for it to s_2. From the rules given above, it is clear that s_2 must possess capabilities for JOHN and for INCREASESALARY of BLUECOLLAR before s_1 will be allowed to transfer the required capability.

However, a problem still exists: Suppose s already has capabilities for EMPLOYEE, BLUECOLLAR and JOHN. If the owner of INCREASESALARY of EMPLOYEE now wants to grant this capability to s, it either has to grant the capabilities for INCREASESALARY of EMPLOYEE, BLUECOLLAR and JOHN simultaneously (G4.1, G4.2 and G4.3) or, alternatively, revoke the capabilities for object JOHN and class BLUECOLLAR, grant the capability for every INCREASESALARY and then grant the capabilities for object JOHN and class BLUECOLLAR to s again. If the same capability is used to protect INCREASESALARY in class EMPLOYEE (where it is defined) and in any instance or subclass (that inherits or redefines it), then it is very easy to grant access rights simultaneously.

If we accept the general rule that the same capability will be used to protect any facet x

- In the class c where it is defined;
- In any subclass of c that inherits or redefines x; and
- Any instance of c or one of its subclasses

then the restrictions given earlier are much simplified: rules G1.2, G2.2, G3.1, G4.1, G4.2 and G4.3 fall away, leaving only rules G1.1 and G2.1. In fact, it seems worthwhile trading the earlier generality for the simplicity gained by the latter rule. In contrast, many mandatory security models do allow inherited facets to be relabeled without any apparent problems, compare [Kee89,Kee90,Lun90a,Var91]; see also chapters 6 and 9.

It is important to note that the mentioned restrictions (or equivalent restrictions) do not only apply to DISCO: they apply to any similar model, in other words any model that

1. Hides the existence of an entity from a subject if the subject is not authorised to access the entity; and
2. Allows objects, classes and methods and/or instance variables to be protected;

If requirement (1) above is dropped, the required restrictions are much less strict. We do not go into details at this time.

12.4.2 Revoking rights

The rules restricting the passing of capabilities given above, also restrict the revoking of capabilities. In general, a capability may not be revoked if revoking it will leave some subject in possession of a capability that may not be passed to it if it does not possess the revoked capability.

The following specific restrictions exist:

R1 A capability for a class may not be revoked from a subject if

.1 The subject possesses a capability for an instance of the class; or

.2 The subject possesses a capability for a subclass of the class;

R2 A capability for a facet (method or instance variable) of an instance may not be revoked from a subject if the subject possesses a capability for the instance; and

R3 A capability for a facet (method or instance variable) of a class may not be revoked from a subject if the subject possesses a capability for the class;

R4 However, despite rules R2 and R3 it is permissible to revoke a capability for a facet of a class *c* if access rights for that facet are simultaneously revoked in all subclasses of *c*, in all superclasses of *c* and in all instances of any of these classes; a capability for a facet of an object may be revoked if the capability for that facet in its class (and in the subclasses, superclasses and instances of its class) are revoked simultaneously.

To illustrate these rules, suppose again that we have the database depicted in figure 12.1. Suppose that the personnel manager wants to revoke the right to invoke the INCREASESALARY method (defined in EMPLOYEE) from some subject *s*. According to rule R4, this may be easily done by revoking the capabilities for INCREASESALARY of EMPLOYEE, BLUECOLLAR, WHITECOLLAR, JOHN and JAMES simultaneously. This is particularly easy if all those methods are protected by the same capability.

Alternatively, to revoke the capability for INCREASESALARY of EMPLOYEE, the owner will proceed as follows: If *s* has a capability to access BLUECOLLAR, that capability will first have to be revoked; before doing that the capability to access JOHN will have to be revoked (if *s* does possess such a capability). Similarly the capabilities for WHITECOLLAR and JAMES will be revoked. The capabilities to access INCREASESALARY of JOHN, BLUECOLLAR, WHITECOLLAR, JAMES and EMPLOYEE may then be revoked, after which the capabilities to access JOHN, BLUECOLLAR, WHITECOLLAR, JAMES and EMPLOYEE may be passed to *s* again.

12.4.3 Establishment of capabilities

The rules restricting granting and revoking of rights also have implications for the initial protection of entities: If the owner decides to protect an entity with a specific capability, DISCO will first determine which subjects have copies of the old and/or new capability to ensure that these rules will not be violated. These rules may, for example, be violated if the owner of a class were allowed to change the capability for a class and leave some subject in possession of a capability for an instance of the class, but without the new capability for the class itself. A related problem may occur if the owner selects an existing capability to protect an entity: another subject may already possess the capability and suddenly be authorised to access the entity; in practice this would not be a problem on its own, since there will probably be a reason for using an existing capability and a (related) reason why the other subject already possesses the capability. However, this could be a problem if the change will enable another subject to access the entity under circumstances where passing a capability for the entity to this other subject would have violated one of the given rules. DISCO therefore checks, whenever the capability associated with an entity is replaced by another capability, that none of the granting or revocation rules are violated by the facts that

- All those subjects in possession of the old capability, but not of the new capability, will no longer be allowed to access the entity (a capability is thus effectively revoked); and
- All those subjects in possession of the new capability, but not of the old capability, will now effectively be granted the right to access the protected entity.

From the preceding discussion, we recommend that the capability necessary to access an entity is established as soon as possible after the entity is created and not modified afterwards. This is illustrated by our example earlier where we used the CREATE method of a class the establish the protection of an entity.

12.5 Information flow

Although information flow is usually not considered as part of a discretionary access control model, but rather as part of a mandatory access control model, we will briefly mention the possibility to include it in DISCO. The main characteristic of a discretionary access control model is the fact that a user has discretionary power to determine who may access information (or resources) owned by the specific user. In a mandatory access model users have no such power; in fact the model ensures that information does not even flow via an indirect route to a subject not authorised to access it.

One possible reason for including information flow aspects in a discretionary access control model is to ensure that information does not flow unintentionally to a subject not authorised to access it; if the owner does indeed want that subject to access the information, the owner can easily authorise that subject appropriately—without this explicit action from the

- The initial message from the primary accessor has low sensitivity because it does not contain any information;
- A method starts execution with the sensitivity of the message that activated it;
- The sensitivity of the method activation is adjusted whenever it receives a reply after sending a message—the sensitivity is adjusted to the least upper bound of its current sensitivity and the sensitivity of the reply; similar action is taken when a value is read from a variable;
- All messages sent by a method activation has the sensitivity currently associated with the activation; whenever the sensitivity of the method activation is adjusted, all subsequent messages will be sent with the new sensitivity level; and
- If the method sends a reply to its calling method, the reply has the last 'current' sensitivity of the method activation.

Table 12.2: Rules for determining the sensitivity of a message.

owner the owner can rest assured that the information will stay protected and that another (authorised) subject can not pass the information on to the unauthorised subject.

For DISCO we use the parameters from chapter 10 to describe this aspect of the model.

In DISCO the authorisation (parameter X3.1) depends on the primary accessor only; in other words, the authorisation of the (human) subject is not influenced by the path the request follows.

Message sensitivity (parameter X3.2) in DISCO is determined according to the usual rules given in table 12.2—see chapter 9 for details.

DISCO restricts the flow of sensitive information (parameter X3.3) by rejecting messages that will write information to variables that are not protected enough. It may be interesting to consider the implications of using other options for this design parameter.

12.6 Possible future work

An aspect that DISCO did not address is the possibility to view capabilities as objects. The capabilities may be instantiated from classes, where the classes form a hierarchy. If a capability of a given class is required to access an entity, any instance of a subclass will be sufficient to access the entity. Anyone authorised to invoke the CREATE method of such a capability class may create new capabilities, in other words distribute access rights to other subjects. Implicit in the above discussion is the fact that capability classes and capability objects may be protected by capabilities themselves.

12.7 Conclusion

See appendix B for a summary of DISCO.

Two aspects of DISCO are especially noteworthy:

- Restrictions that DISCO imposes on the granting and revocation of rights must be imposed by any similar object-oriented discretionary security model; and
- The ability to include the setting up of initial security attributes for a new object as part of the CREATE method makes it possible to include a significant part of the security policy of a company in the CREATE methods of the various classes; it may be worth investigating to what extend the security policy can be encoded in methods, limiting the freedom of owners to pass capabilities in an *ad hoc* fashion to other subjects.

We make the following specific recommendations for security models similar to DISCO:

- The same capability must be used to protect any inherited facet, from the point where it is defined, through any subclasses where it is redefined, up to any instances where it is available; and
- The decision on which capability must be used to protect any entity must be made as soon as possible after the entity is created; this must be changed as infrequently as possible.

Chapter 13

Epilogue

This final chapter evaluates the state of the research described in this work. It also identifies directions in which this research may be continued.

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

T.S. Eliot
Little Gidding, 5

13.1 Introduction

A piece of work hardly ever reaches a point where it cannot be improved. This work is no different: it can be expanded and refined in many ways. However, after such expansion or refinement, it will probably still be open for improvement. One therefore has to decide at what point to stand back and appraise what has been done. We feel that this work has reached such a point—a point where the work can be submitted for criticism. We therefore submit it with the hope that it will draw comments to enable us to evaluate the current research. May it contribute to the work of others, and may it stimulate our future research.

13.2 Results

Results from this research have been communicated in a number of papers. At present the following papers have been submitted for refereeing:

- "Building a secure database using self-protecting objects", based on chapter 6 (accepted for publication by *Computers & Security*);
- "An object-based version of the Path Context Model", based on chapter 5;
- "A taxonomy for secure object-oriented databases", based on chapters 7 to 11; and
- "DISCO: a discretionary security model for object-oriented databases", based on chapter 12 (accepted for IFIP SEC'92, Singapore).

13.3 Future research

The work described in this thesis may be extended and supported by a number of future research projects. In this section we consider some of the questions that may be addressed by such projects. A number of the issues listed here has been described earlier in the thesis; however, we list such issues again to present a coherent account.

13.3.1 The taxonomy

Chapter 11 mentioned a number of ways in which the taxonomy may be extended—the inclusion of any of the 'new' parameters touched upon there or the identification of other new parameters may be investigated to this end.

In addition, the following issues warrant attention:

- Many of the models covered by the given taxonomy seem complex to implement. However, security models must be simple to implement: Only if simple implementation strategies exist, will it be possible to trust that the implementation accurately reflects the model. Further, the implementation must be economical: if the security model adds too much overhead costs to the operation of the system, the model is not practical. The first research issue is therefore to find implementation strategies that economically implement some of the more complex models (such as SECDB). Examples of implementation strategies for conventional databases may be found in [Laf90].

  Such an investigation of implementation strategies for secure object-oriented databases could also reveal additional insights about possible choices for the various design parameters.

- Most of the work on sensitivity flow use the sensitivity of the variable or object or the sensitivity of the value contained in such a variable to restrict such flows to ensure secrecy. We also used this approach, both in the description of SECDB (chapter 6) and in the taxonomy (chapter 10). However, as mentioned in both those chapters, it is more natural to determine the sensitivity of methods because methods represent the activities individuals have to execute as part of their functions in an organisation. To illustrate the problem, suppose that some (possibly non-sensitive) information is returned by a very sensitive method. If this information is now stored in another location, should

  - The receiving *variable* be marked as sensitive;
  - All *methods* with access to the variable be marked as sensitive; or
  - Is it possible to determine (or approximate) the actual sensitivity of the information using all the methods with access to it as a basis?

  The answer is not obvious. Alternatives should be considered.

- It will be worthwhile to examine individual, existing design parameters. This may be done to discover alternative possible values for such parameters and to study the influence of such values on other parameters.

- Polyinstantiation should be investigated—an initial discussion is contained in section 11.6. It is possible to incorporate polyinstantiation in the axioms for the taxonomy (chapter 7) and consider its influence on all the results obtained with the aid of the taxonomy. 'Polyinstantiation' may have its usual meaning or it may be one of the weaker forms described in chapter 11. It is also possible to investigate (mathematically) what the properties of the weaker forms are.

13.3.2 Discretionary security

Very little has been published on discretionary security; DISCO (chapter 12) is an initial attempt to address this aspect of security. Other aspects of discretionary security that may be investigated, include the following:

- A discretionary security model for object-oriented databases may be formulated where the capabilities themselves are objects. Issues that need addressing in such a model include the role of capability classes, the influence of inheritance between such classes, the methods that a capability object must support and the mechanism to transfer capabilities.

- DISCO indicated how the CREATE method of a class may be used to group a number of 'capability operations' (together with normal operations') in a method. The extent to which the security policy of an organisation may be reflected by such sequences of capability operations is unknown.

  In a certain sense, direct manipulation of capabilities (such as transfer and revoke) correspond to direct manipulation of data; benefits are gained by encapsulating data in objects and only allowing the object's methods to manipulate the data; it is possible that similar benefits may be gained by encapsulating capabilities in objects and providing methods to (exclusively) manipulate the capabilities.

- DISCO uses *existence protection* to interpret protection (X1.2) and identifies a number of restrictions that apply to the manipulation of capabilities; similar, but not identical, restrictions will apply if protection is interpreted differently. It will be enlightening to identify restrictions for other protection interpretations.

13.3.3 Distributed databases

If a database is distributed the security problem is more complex than in the centralised case. Object-oriented databases do map naturally to a distributed environment; however object-orientation on its own is not sufficient to solve the problem (as [Var91] seems to assume). Aspects of network security have to be considered when a model for a secure distributed system is developed.

Models that consider the *access path* (such as SECDB) or the *set of active objects* to determine the *authorisation* (X3.1) have potential benefits for distributed systems, because it may take the fact that a request comes from a remote system into account when it determines whether a request should be allowed to access an entity.

Models that address these issues for secure distributed databases need to be developed.

13.3.4 SECDB

Chapter 6 identified a number of research questions that have to be answered before SECDB will be practical. These are:

- Automated profile generation and automated validation of profiles (for consistency);
- Identification of unnecessary baggage in order to keep baggage as small as possible;
- Implications of multiple inheritance on the model;
- Designing a notation for specifying security constraints (especially in real-world systems); and
- The model should be applicable to distributed systems where requests may come from many sites, often not even directly connected to the database site; note that the distributed systems may operate concurrently, and the effects of this on the model should be investigated.

13.3.5 Security policy

The security policy is a document that prescribes the implementation of computer security in a particular organisation. A number of research projects are under way to investigate the link between such a document and the computer security actually implemented in an organisation—see for example [Pot91]. This work referred to a number of issues that may influence the format of the security policy. We mention some of these:

- A security model such as SECDB requires profiles to function. If the security policy is in an acceptable format, it may be possible to automate the generation of profiles from the security policy;
- DISCO indicated that it may be possible to combine 'primitive' security operations (such as the passing of capabilities) into methods: these methods may represent activities described in the security policy, thus offering a higher level of abstraction for security operations; it may also restrict the extent to which security entities may be manipulated at a primitive level.

Bibliography

[Aho86] AV Aho, R Sethi and JD Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986

[Atk89] M Atkinson *et al*, "Object-oriented Database System Manifesto", *Proceedings of the first international Conference on Deductive and Object-oriented Databases*, Kyoto, Japan, December 1989

[Atw90] T Atwood, "The Object-oriented Database System Manifesto: A Consensus from Academia", *Hotline on Object-oriented Technology*, **1**, 3, January 1990, 6–9

[Aye91] TR Ayers, DK Barry, JD Dolejsi, JR Galarneau and RV Zoeller, "Development of ITASCA^{TM}" *Journal of Object-oriented Programming*, **4**, 4, July/August 1991, 46–49

[Bel76] DE Bell and LJ LaPadula, "Secure computer system: unified exposition and Multics interpretation", *Rep. ESD-TR-75-306*, March 1976, MITRE Corporation

[Bib77] K Biba, "Integrity Considerations for Secure Computer Systems", *US Air Force Electronic Systems Division*, 1977

[Bos89a] WH Boshoff and SH von Solms, "A Path Context Model for Addressing Security in Potentially Non-secure Environments", *Computers & Security*, **8**, 1989, 417–425

[Bos89b] WH Boshoff, *A Path Context Model for Computer Security Phenomena in Potentially Non-Secure Environments*, Ph.D Dissertation, Rand Afrikaans University, 1989

[Bos90] WH Boshoff and SH von Solms, "Application of a Path Context Approach to Computer Security Fundamentals", *Information Age*, **12**, 2, April 1990, 83–90

[Cae90] W Caelli, Private communication

[Dat90] CJ Date, *An Introduction to Database Systems Volume 1, 5th ed.*, Addison-Wesley, 1990

[Dav89] JR Davis, "IBM's data management design", *Patricia Seybold's Office Computing Report*, **12**, 12, December 1989, 1–12

[Den88] DE Denning, "Database Security", pp 1–22 in [Tra88], 1988

[Dit89] KR Dittrich, M Härtig and H Pfefferle, "Discretionary Access Control in Structurally Object-Oriented Database Systems", pp 105–121 in [Lan89], 1989

[Gar90] TD Garvey and TF Lunt, "Multilevel Security for Knowledge Based Systems", *Proceedings of the Sixth Computer Security Applications Conference*, Tuscon, Arizona, December 1990

[Gar91] TD Garvey and TF Lunt, "Multilevel Security for Knowledge Based Systems", *Technical Report SRI-CSL-91-01*, Computer Science Laboratory, SRI International, February 1991

[Gol83] A Goldberg and D Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983

[Hai90] B Hailpern and H Ossher, "Extending Objects to Support Multiple Interfaces and Access Control", *IEEE transactions on Software Engineering*, **16**, 11, November 1990, 1247–1257

[Har76] M Harrison et al, "Protection in Operating Systems", *Communications of the ACM*, **19**, 8, August 1976, 461–471

[Kee89] TF Keefe, WT Tsai and MB Thuraisingham, "SODA: A Secure Object-oriented Database System", *Computers & Security*, **8**, 1989, 517–533

[Kee90] TF Keefe and WT Tsai, "Prototyping the SODA Security Model", pp 211–235 in [Spo90], 1990

[Kim89] W Kim and FH Lochovsky (eds), *Object-oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989

[Kim91] W Kim, "Object-oriented database systems: strengths and weaknesses", *Journal of Object-oriented Programming*, **4**, 4, 1991, 21–29

[Laf90] C Laferriere, "A Discussion of Implementation Strategies for Secure Database Management Systems", *Computers & Security*, **9**, 1990, 235–244

[Lan89] CE Landwehr (ed), *Database Security II: Status and Prospects*, North-Holland, 1989

[Lar90] MM Larrondo-Petrie, E Gudes, H Song and EB Fernandez, "Security Policies in Object-oriented Databases", pp 257–268 in [Spo90], 1990

[Lun89] TD Garvey and TF Lunt, "Secure Knowledge-based Systems", *Technical Report SRI-CSL-90-04*, Computer Science Laboratory, SRI International, August 1989

[Lun90a] TF Lunt, "Multilevel Security for Object-Oriented Database Systems", pp 199–209 in [Spo90], 1990

[Lun90b] TF Lunt, DE Denning, RR Schell, M Heckman and WR Shockley, "The SeaView Security Model", *IEEE Transactions on Software Engineering*, **16**, 6, 1990, pp 1247–1257

[Min87] NH Minsky and D Rozenshtein, "A Law-Based Approach to Object-Oriented Programming", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1987, 482–493

[Miz90] M Mizuno and AE Oldehoeft, "An Access Control Language for Object-Oriented Programming Systems", *Journal of Systems Software*, **13**, 1990, 3–12

[Pfl89] CP Pfleeger, *Security in Computing*, Prentice-Hall, 1989

[Pot91] D Pottas, Ph.D Thesis, Rand Afrikaans University (In preparation)

[Sal74] J Saltzer, "Protection and the Control of Information Sharing in MULTICS", *Communications of the ACM*, **17**, 7, Jul 1974, 388–402

[San90] R Sandhu, "Multilevel Object-oriented Databases", *13th National Computer Security Conference*, Washington DC, 1990, 597–598

[Shi89] JJ Shilling and PF Sweeney, "Three Steps to Views: Extending the Object-Oriented Paradigm" *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1989, 353–361

[Shr88] B Shriver and P Wegner (eds), *Research Directions in Object-Oriented Programming*, MIT Press, 1988

[Sny81] L Snyder, "Formal Models of Capability-Based Protection Systems", *IEEE Transactions on Computers*, **30**, 3, March 1981, 172–181

[Spo89] DL Spooner, "The Impact of Inheritance on Security in Object-Oriented Database Systems", pp 141–150 in [Lan89], 1989

[Spo90] DL Spooner and C Landwehr (eds), *Database Security III: Status and Prospects*, North-Holland, 1990

[Thu89] MB Thuraisingham, "Mandatory Security in Object-Oriented Database Systems", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1989, 203–210

[Tra88] JF Traub *et al* (eds), *Annual Review of Computer Science Volume 3*, Annual Reviews Inc, 1988

[Van70] APJ van der Walt, "Random Context Languages Symposium on Formal Languages", Oberwolfach, West Germany, 1970

[Var91] V Varadharajan and S Black, "Multilevel Security in a Distributed Object-Oriented System" *Computers & Security*, **10**, 1991, 51–67

[Weg88] P Wegner, "The Object-Oriented Classification Paradigm", 479–560 in [Shr88], 1988

[Weg90] P Wegner, "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger*, **1**, 1, 1990, 7–87

Appendix A

Taxonomy: Summary

The taxonomy described in chapters 7 to 11 is summarised in this appendix.

A brief summary of the classification structure proposed in this thesis follows. Options that we considered for each of the design parameters are mentioned.

X1 Labeling semantics

X1.1 Underlying model

We considered explicit security levels, access control lists, capabilities and extensions based on these mechanisms, as well as combinations of these. Other mechanisms are possible.

X1.2 Protection interpretation

Only two possibilities were considered: existence protection and access protection. Other possibilities were mentioned.

X2 Structural labeling

X2.1 Protectable entities

Work in this work was based on the premise that any object or class or any facet of such an object or class may be labeled. Note our interpretation of the term *class*. Weaker forms of the restrictions apply if single-level objects are used.

X2.2 Label instantiation

This work used only one mechanism to label entities: inheritance; in other words, a class is labeled and the labels are inherited by subclasses and instances.

X2.3 Relationship restrictions

The compulsory restrictions are summarised in tables 9.1 and 9.2. Other restrictions a model may require and/or enforce were indicated.

X3 Dynamic labeling

X3.1 Authorisation flow

This work indicated how clearance may depend on the *primary accessor*, the *set of active objects* or the *access path*. The relevant attributes were also identified. In addition to the method a secure database model uses, it thus have to specify up to four additional pieces of information:

1. The format of the clearance attributes ($\Sigma.c$, $a_i.c$ and $M.c$);
2. The format of the sensitivity attribute ($e.l$);

3. For which values of $M.c$ and $e.l$ message M may access entity e; and
4. How are clearances combined, ie define *clear*.

X3.2 Sensitivity flow

A list of rules was given that may be used to sensitivity label messages; see table 10.6 for a summary. Models may use variations of these and also additional rules to label messages.

X3.3 Information flow restrictions

Four possible strategies have been identified to prevent information from being compromised as a result of it flowing through the system, namely

1. Reject it if the sensitivity of the receiving variable is not high enough;
2. Increase the sensitivity of the receiving variable or object if necessary;
3. Modify the sensitivity of the receiving variable up or down (but not lower than a fixed lower limit); or
4. Use polyinstantiation.

Appendix B

Taxonomy: Examples

This appendix illustrates the taxonomy described in chapters 7 to 11 using a number of examples.

B.1 SODA

X1 Labeling semantics

X1.1 Underlying model: Explicit (numeric) levels.

X1.2 Protection interpretation: Access protection.

X2 Structural labeling

X2.1 Protectable entities: Either the instance variables of an object or the entire object.

X2.2 Label instantiation: Inheritance.

X2.3 Relationship restrictions: Nothing is specified explicitly; normal restrictions for access protected models (table 9.2) apply.

X3 Dynamic labeling

X3.1 Authorisation flow: Authorisation based on clearance of primary accessor (table 10.2). Clearance of subjects and sensitivity of entities are integers (range not specified). A subject with clearance $s.c$ may access an entity with sensitivity $e.l$ if $s.c \geq e.l$.

X3.2 Sensitivity flow: The normal rules (table 10.6) apply, except that the production $M \rightarrow px_i$ has the semantic action $M.u := x_i.l'$ where $x_i.l'$ denotes the initial sensitivity of variable x_i.

X3.3 Information flow restrictions: SODA uses polyinstantiation. In addition, SODA rejects messages that are more sensitive than the specified upper limit for the concerned variable—see equation 10.10 on page 128.

B.2 Lunt

X1 Labeling semantics

X1.1 Underlying model: Explicit levels.

X1.2 Protection interpretation: Existence protection.

X2 Structural labeling

X2.1 Protectable entities: Objects and facets of an object (methods and instance variables).

X2.2 Label instantiation: Not specified.

X2.3 Relationship restrictions: Summarised in table 9.3.

X3 Dynamic labeling

X3.1 Authorisation flow: Message clearance is based on the primary accessor according to property 6—see equations 10.8 and 10.9 on page 117 with the accompanying discussion.

X3.2 Sensitivity flow: The sensitivity of a message seems to be equal to the classification level of the subject that sent it.

X3.3 Information flow restrictions: Not specified.

B.3 SECDB

X1 Labeling semantics

X1.1 Underlying model: Path Context Model (PCM).

X1.2 Protection interpretation: Access protection, but axiom 7 does not apply.

X2 Structural labeling

X2.1 Protectable entities: Layers of an object, methods and variables.

X2.2 Label instantiation: Inheritance.

X2.3 Relationship restrictions: Nothing is specified explicitly; normal restrictions for access protected models (table 9.2 on page 107) apply, except those based on axiom 7.

X3 Dynamic labeling

X3.1 Authorisation flow: Authorisation based on complete access path (table 10.4 on page 120). Clearance attributes are *baggage vectors* identifying the object, method, domain (processor) and processor integrity state. Clearance attributes are combined by concatenating them. Sensitivity attributes (*profiles*) are (parts of) a formal grammar. A subject with clearance attribute $s.c$ may access an entity with sensitivity attribute $e.l$ if the string $s.c$ is a word of the language specified by $e.l$.

X3.2 Sensitivity flow: The normal rules (table 10.6 on page 125) apply.

X3.3 Information flow restrictions: Sensitivity of concerned variable is increased.

B.4 DISCO

X1 Labeling semantics

X1.1 Underlying model: Capabilities.

X1.2 Protection interpretation: Existence protection (a subject does not know that an entity exists if the subject is not authorised to access the entity).

X2 Structural labeling

X2.1 Protectable entities: Objects, methods, variables and classes.

X2.2 Label instantiation: The creator (instantiator) initially has all access rights for the entire object.

X2.3 Relationship restrictions: No additional relationship restrictions are specified; the normal restrictions for existence protected models (tables 9.1 and 12.1) apply.

X3 Dynamic labeling

X3.1 Authorisation flow: Authorisation is based on clearance of primary accessor only.

X3.2 Sensitivity flow: The normal rules (table 12.2) apply.

X3.3 Information flow restrictions: Messages that will cause information to be written to less sensitive variables will be rejected.

List of Figures

List of Tables

Index