A multilevel security model for object-oriented database systems

Linda M. Null
Iowa State University
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
A multilevel security model for object-oriented database systems

Null, Linda M., Ph.D.
Iowa State University, 1991
A multilevel security model for object-oriented database systems

by

Linda M. Null

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:
Signature was redacted for privacy.
In Charge of Major Work
Signature was redacted for privacy.
For the Major Department
Signature was redacted for privacy.
For the Graduate College

Members of the Committee:
Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1991

Copyright © Linda M. Null, 1991. All rights reserved.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................. ix

CHAPTER 1. INTRODUCTION ...................................................... 1
  Database Concepts .............................................................. 1
  Security Issues in Database Systems ........................................ 3
  Problem Statement ............................................................. 12
  Main Contribution of this Thesis ............................................ 14
  Overview of this Thesis ...................................................... 15

CHAPTER 2. RELATED WORKS .................................................... 17
  The Relational Model .......................................................... 17
  The Semantic Models ......................................................... 20
    The entity-relationship model and its concepts ....................... 24
    The functional data model ............................................... 26
    The semantic database model ............................................. 26
  The Object-Oriented Model .................................................. 28
    Background ................................................................. 28
    Core concepts ............................................................. 30
    Object-oriented systems vs. object-oriented database systems ..... 32
  A Comparison of Semantic Models to Object-Oriented Models ........ 35
### Summary

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>THE OBJECT-ORIENTED DATA MODEL</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Objects and Object Identity</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Attributes and Methods</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Classes</td>
<td>39</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Instance variables</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Class variables</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Class Hierarchy and Inheritance</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Class hierarchy</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Name conflict resolution</td>
<td>49</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Abstracting superclasses</td>
<td>52</td>
</tr>
<tr>
<td>3.5</td>
<td>Domains of Attributes</td>
<td>54</td>
</tr>
<tr>
<td>3.6</td>
<td>Aggregation Hierarchy and Composite Objects</td>
<td>56</td>
</tr>
<tr>
<td>3.7</td>
<td>Version Hierarchy</td>
<td>60</td>
</tr>
<tr>
<td>3.8</td>
<td>Schema Evolution</td>
<td>65</td>
</tr>
<tr>
<td>3.9</td>
<td>Summary</td>
<td>70</td>
</tr>
</tbody>
</table>

### Chapter 4. THE SECURITY POLICY

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Overview of Existing Approaches</td>
<td>71</td>
</tr>
<tr>
<td>4.1</td>
<td>The Security Properties</td>
<td>75</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Security properties for objects</td>
<td>75</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Security properties for classes</td>
<td>78</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Security properties for methods</td>
<td>80</td>
</tr>
<tr>
<td>4.1.4</td>
<td>The multilevel-user update problem</td>
<td>82</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Security properties for versions</td>
<td>83</td>
</tr>
</tbody>
</table>
The four hierarchies and security properties .......................... 85
Integrating the Security Policy with the Data Model .................. 86
Information flow within the data model ............................... 86
The security policy .................................................. 89
An example ........................................................... 90
Implementing the security policy ...................................... 94

Summary ................................................................. 101

CHAPTER 5. OBJECT-ORIENTED TRANSACTION MANAGEMENT AND PROCESSING .................................................. 103
Introduction to Transactions and Concurrency Control ............. 103
Nested Transactions .................................................... 104
Locking Policy for Object-Oriented Databases ....................... 107
Conventional granularity-hierarchy locking .......................... 108
Class/subclass and class/instance locking ............................ 111
Aggregate/composite object locking .................................. 120
Version locking ......................................................... 125
Lock inheritance ....................................................... 128
Secure Transaction Processing ........................................ 131
Transaction security policy ........................................... 131
Security inheritance via nested transactions .......................... 133
Failure and recovery .................................................. 137
Summary ................................................................. 138
CHAPTER 6. SECURE QUERY PROCESSING IN OBJECT-ORIENTED DATABASES

Introduction to Views ........................................ 140
Secure Views .................................................. 141
Related Work .................................................. 142
Our Approach to Secure Views ................................. 146
Secure Query Processing ....................................... 152
  Rule base .................................................... 154
  Environmental information ................................. 156
Secure Object-Oriented Query Processing ..................... 157
  Factoring mechanism ...................................... 159
  Integrating the rule base into our model .................. 163
Summary ...................................................... 164

CHAPTER 7. CONCLUSION ........................................ 167
Overview ....................................................... 167
Further Research ............................................. 172

REFERENCES .................................................. 174
LIST OF TABLES

Table 2.1: Data model features ........................................ 28
Table 2.2: Comparison of relational to semantic data models .... 28
Table 2.3: Relational concepts vs. object-oriented concepts ...... 32
Table 2.4: Object-oriented systems vs. object-oriented database systems 34

Table 5.1: Lock mode compatibility for hierarchical locking algorithms . 110
Table 5.2: Compatibility matrix for class and instance locking ...... 114
LIST OF FIGURES

Figure 2.1: An example of a relation ........................................... 18
Figure 2.2: IS-A hierarchy .......................................................... 21
Figure 2.3: IS-PART-OF hierarchy .............................................. 22
Figure 2.4: IS-AN-INSTANCE-OF hierarchy ................................. 22
Figure 2.5: An example of the ER model ...................................... 25
Figure 2.6: An example of the FDM model .................................... 27
Figure 2.7: An example of the SDM model .................................... 27

Figure 3.1: Class and instance variables ....................................... 43
Figure 3.2: An example of a class hierarchy and inheritance .......... 45
Figure 3.3: A name conflict with inherited attribute A .................. 50
Figure 3.4: Component and simple references .............................. 57
Figure 3.5: Version hierarchy ..................................................... 61

Figure 4.1: An example to illustrate our security policy .................. 92

Figure 5.1: An example of a class hierarchy ................................. 115
Figure 5.2: A lock hierarchy including intention locks .................... 129
Figure 5.3: A lock hierarchy with all locks ................................. 130

Figure 6.1: A schema graph ...................................................... 148
Figure 6.2: A query graph ........................................ 149
Figure 6.3: A view graph ........................................... 150
Figure 6.4: Two methods for deriving a secure query graph .... 151
Figure 6.5: Representation of C1 with a graph ................... 160
Figure 6.6: Representation of C2 with a graph ................... 161
Figure 6.7: Representation of environmental information ........ 162
ACKNOWLEDGMENTS

This thesis would not exist were it not for the support and guidance of many people. As I cannot mention all of those who helped me through the long and trying period while at graduate school, I will mention a few of the very important ones.

I would like to thank my advisor, Dr. Johnny Wong, for his constant guidance, encouragement, and support. I am forever indebted to him for things too numerous to list here. Never have I worked with someone so willing to help. He read through this dissertation many times and gave up several weekends to help with its preparation. His understanding, patience and special blend of enthusiasm and generosity will forever be remembered and I only hope that one day I will be able to provide the same type of guidance to someone else.

I express my gratitude to Dr. Les Miller for his suggestions, advice, friendship and guidance throughout my stay at Iowa State. I would also like to thank my other committee members, Dr. Arthur Oldehoeft, Dr. Pete Boysen, Dr. Don Piggozi, and Dr. Jim Davis who each provided me with helpful suggestions. Sincere thanks go to my colleagues, Dr. Phil Heeler and Dr. Merry McDonald, who convinced me to set higher goals for myself and made it possible for me to attend graduate school.

I would like to thank my research seminar group for providing many productive discussions of ideas related to my work. Many people put in long hours to help me
assimilate my ideas and to help me think through my concepts.

Many thanks are due to my friends, including Steve and Debbie Nugen, for making life in Ames pleasant and graduate school tolerable. I would also like to thank my very good friend and fellow student, Pinaki Shah, who was always there with constant encouragement and brotherly advice, and who always had faith in me, even when I had lost it in myself.

I would like to express my gratitude to my family, including Christi Jensen, for supporting my decision and for accepting the hardships caused by my extended absence, and Sandy Cox for volunteering her time to care for my horses while I finished my degree. My special thanks to my husband, Jim Jensen, who has been a wonderful source of constant encouragement, total understanding and unwavering faith. As my most valuable source of support, I thank him for enduring my graduate studies and me all these years.
CHAPTER 1. INTRODUCTION

Database Concepts

The database field has seen many changes over the past several years. Although these changes have been significant, one common thread can be traced throughout the different generations of database development: a database must manage large amounts of persistent data in such a way that users can share the data it stores. A database management system (DBMS) is a software system that manages this data efficiently, facilitating the creation and maintenance of a database. The DBMS acts as an interface between the user and the data by allowing the user to deal with the data abstractly, using high level commands that invoke encoded algorithms to manipulate the data. These high level commands comprise the query language (or data manipulation language) that all users must use to access the database.

A database can be thought of as a collection of information existing at two different levels. At one level exists the physical data, stored on disk or tape. At the second level, we have the conceptual database, or the data as the user perceives it and how it applies to the real world. The DBMS must provide a data definition language to allow a user to specify this conceptual database and a data manipulation language to allow the user to access the physical data. Thus, at one end, we have the computer dealing with bits of information, and at the other end, we have the user
dealing with the real world information. The translation between these two extremes is provided by the DBMS.

Using a database has many advantages. Perhaps the biggest advantage is the ability to share information. Individual users do not have to maintain their own collection of data, thus reducing redundancy. Also, since the data is shared, data consistency is improved because one change is seen by all users. But the ability to share gives rise to problems that the database must also address. Since all users share the same database, data integrity must be enforced so that values in the database are protected against accidental or malicious changes. Some type of access control must be enforced to ensure that only authorized users are allowed to access certain bits of data in the database. These security interests often conflict with performance. Database security is discussed in the next section.

A database management system should also support a data model that allows the user to see information in a logical sense rather than a physical sense. A data model is therefore a mathematical abstraction that can be used to conceptualize the data. Ullman [84] defines a data model as a mathematical formalism with two parts. First, it contains a notation for describing data. Second, it contains a set of operations used to manipulate that data.

Originally, file systems were used to store data. To access the information, the user had to know how to navigate from one set of data to another, thus placing more responsibility on the user. Simply storing data does not imply one has a database system. A DBMS allows efficient access to the information that is stored. Chapter 2 summarizes the various data models that have been proposed.

The main research emphasis now is to integrate database technology with object-
oriented concepts. This has been partially brought about by the new technology that allows various types of information to be stored. Originally, databases were designed for business data processing and stored mainly text and numeric data. Presently, with the advent of recent developments in the area of mass storage of data (particularly the digital representation of real world objects), there is a demand to store multimedia data. Multimedia information encompasses traditional data but also includes data in the form of sound, vector graphics, tables, and digitized images. Conventional database systems (such as network, hierarchical, or relational) lack the necessary mechanisms to deal with multimedia data. Constructs need to be provided to define semantic relationships among objects (for example copies or versions). The data model needs to allow for extensibility and evolution of the schema. Complex relationships exist in multimedia systems and also in design databases that cannot be represented in conventional systems. The result has been a push towards using object-oriented systems in the database area. Object-oriented systems are better suited to handle multimedia data and various types of design information than conventional database systems are. This is because many of the semantic relationships that exist in these advanced environments cannot be captured in a conventional system but are inherent in an object-oriented model. Object-oriented database systems reduce the semantic gap between the complex applications and the data storage to support these applications.

Security Issues in Database Systems

A database must provide efficient access to the collection of data it stores and it is often required to guarantee a certain measure of security in regard to this data
as well. The security necessary in database systems can be divided into two different areas, one dealing with the integrity of data in the database, and one dealing with the availability of the data. Integrity of the database refers to the physical integrity of the data, the logical integrity, and the element integrity. Physical integrity deals with protecting the data in the database from external physical problems, such as a power failure or a head crash on a disk. Most database systems provide some method of recovery to protect the physical integrity of the database, such as logging the transactions.

The structure of the database must also be preserved. This logical database integrity deals with the actual values in the database. Modification of one value in the database should not adversely affect other values.

Users should be able to trust the data in the database to be accurate. Element integrity is enforced by the DBMS. By employing the use of field checks, the DBMS tests for appropriate values in certain fields. An example of this might be to have the DBMS check to ensure that numeric data has not been inserted where alphabetic data is required, or that upper case letters have been used instead of lower case. These validity checks are often used at the time the user enters the data.

The second area of security deals with availability. First, concurrent access to data must be arbitrated in the case of two users requesting the same information. This is provided by two common approaches. Locking is used in many systems. In order to maintain consistency, requested objects are locked to ensure their inaccessibility by any other user. Objects can be locked in exclusive mode or in shared mode, depending on the type of access desired. Another method to deal with concurrency control is timestamps. Systems using this method fix a unique timestamp to each transaction.
against the database, resolving conflicting requests for data using timestamp order. Timestamp ordering has an advantage over using locks in that holding locks can result in deadlock, whereas a system using timestamps is deadlock free. However, timestamps result in many aborted transactions and are harder to implement than locking, so the latter is commonly used in commercial systems.

The security notion of availability also encompasses the idea of authorization. The DBMS enforces some type of access control to determine who has authorization to various elements. The idea of ownership is incorporated into access controls so subjects can grant and revoke privileges for objects they own. For access controls to be effective, users must be identified properly and the information specifying the access rights of the users must be protected from unauthorized modification. Access controls are set up in three different ways. One uses an access hierarchy which automatically grants certain rights to privileged subjects. Another incorporates the idea of a capability, or a ticket to access an object, and allows the holder of the ticket authorization to the object for which he holds the capability. Access can also be set up by using an access control list. A database administrator has the job of determining how the access control lists are to be set up and dealing with other questions of policy.

Access control techniques help maintain the integrity of the objects in the database. However, concern must be directed at the dissemination of the information as well. Information flow systems are concerned with how information may flow within the system. Denning [18] introduces a flow control model using a lattice structure. This lattice is an extension of the Bell and LaPadula model [7], which is used to describe the security policy in military systems. The Bell and LaPadula model
assigns security clearances to subjects and security classes to objects and establishes the allowable paths of information flow in a secure system. The security policy is given in terms of two properties. The simple security property states that a subject $S$ has read access to an object $O$ only if the security class of the object is dominated by the security clearance of the subject ($SC(O) \leq SC(S)$). The $\ast -$ property states that a subject $S$ may not have read access to an object $O_1$ and write access to an object $O_2$ unless $SC(O_1) \leq SC(O_2)$. This property is used to prevent write-down of information. It tends to be stronger than necessary because $S$ may not cause information flow from $O_1$ to $O_2$. Many applications require that this condition be satisfied only if $O_2$ is functionally dependent on $O_1$. Others simplify the requirement by saying that a subject $S$ may not write to an object $O$ unless $SC(S) \leq SC(O)$.

Denning's lattice model is an extension of the Bell and LaPadula model. A lattice is a partially ordered set where, for any two elements in the set, there exists a least upper bound and a greatest lower bound. The lattice model represents a generalization of the military model formalized by the Bell and LaPadula model. The military uses the principle of least privilege and limits access by the need-to-know rule. Access is only given to subjects when that access is necessary for the subject to perform a particular job. This is accomplished by assigning not only a security clearance level to each subject and classification labels to all objects, but also by associating a compartment to describe the subject matter of information the user is able to access. Therefore, the security classification of a subject depends on the clearance and the compartment designation. If all possible combinations of security clearances and compartments are considered, a new lattice is created. A subject $S$ is granted access to an object $O$ if $S$ has the appropriate clearance (simple
and ⋆-property must be enforced) and the compartment for \( S \) is contained in the compartment for \( O \).

The area of authorization is divided into two types: mandatory authorization and discretionary authorization. Mandatory authorization (nondiscretionary) policies apply to all objects in the database, regardless of the contents or the owner. Discretionary policies are the choice of the subject owning or managing the object. Many systems employ both policies. In this case, the mandatory policy is enforced first, using the discretionary policy to further limit access to various elements of the database.

Extensive research has been done in the area of database security. Implementation of security measures in information systems is discussed in [16]. Data security in general is reviewed in [22]. The use of trackers to attain unauthorized information is addressed in [24]. Query restriction and modification are also used as a means to enforce security [19, 77]. An authorization mechanism for relational database systems is defined in [37], and the use of “noise addition” is discussed in [56].

The problem of security, however, becomes much more complex when one considers multilevel security. If several users with varying clearance levels are sharing access to data with different security labels, guaranteeing that users never access information to which they have no authorization becomes a very difficult problem. The security classification of a single element may be different from that of another element in the same record. It can also be different from that of another element in the same column (same attribute). Thus security enforcement may need to be implemented for each individual element, instead of for the entire relation or for a tuple. A user may be authorized to see part of a record instead of the whole record.
A user may be able to see one field or another, but not both at the same time. These issues present challenges for researchers in database security. It is difficult to base authorization on the content of a field or on the context in which the field is used.

Significant research has been done in the area of multilevel security. Graubert and Kramer [35] introduce the integrity lock as a method to provide integrity and limited access to a database. Each piece of data consists of the data itself, a sensitivity level, and a checksum to bind the two together. The idea of a front-end process (or guard) has been developed to enhance the security of existing systems with the least number of changes to the system [34]. Denning [20] introduces the idea of filters (similar to the idea of integrity locks and trusted front-end processors) and shows how direct access and Trojan Horse direct release threats can be solved using a trusted filter and cryptographic checksums. Denning extends this idea to a commutative filter [21], a process that interfaces to the user and the database manager. Unlike the original filter, however, the commutative filter reformats the query to capitalize on the efficiency of the DBMS. The query is modified in such a way that sensitive information is never extracted from the database.

A window or view [23, 25] represents an organizational approach to multilevel database security. A view represents a subset of a database, containing exactly the information that a user is entitled to access. The main idea is to separate the database into user views. These individual subsets then guarantee that the user does not access values outside the permitted level (since unauthorized data values are not part of the user's view). Further information on multilevel database security can be found in [26, 29, 62].

Most conventional database systems offer some kind of authorization mechanism
to specify an authorization type on a database entity for a particular user. Authorization types in commercial database systems consist of read, write, create and delete access rights. Relational database systems support the notion of views to accomplish content-based authorization. A query is the basis of access for a user in a relational database.

In an object-oriented environment, authorization becomes more complex. Because objects encapsulate data and methods, and because different hierarchies exist, additional authorization types are necessary. Objects themselves are the unit of access and should therefore be the unit of authorization. The object model is also semantically richer, thus complicating the issues of access even further. Access control and information flow control in a distributed object-oriented system are addressed in [59]. The information flow certification mechanism is a combination of compile-time analysis and run-time certification. An authorization model for object-oriented databases is presented in [68] which is further expanded in [49]. However, this authorization model does not include general issues such as mandatory and discretionary authorizations or content and context-based authorizations. Thuraisingham formulates a mandatory security policy for the object-oriented database system ORION [83]. Security properties are presented for objects, classes, and methods, with no consideration for versions. Keefe [44, 45] describes a mandatory security model for object-oriented systems concepts. Aggregate objects are addressed on a very general level, with no consideration to the problems involved with their use. The concepts of composite objects and copies are not addressed. Versions are mentioned briefly, with no real support for version control.

Thuraisingham describes how security properties can be incorporated into the
object-oriented model ORION. A formal security policy for the database system is presented, including a discussion of other mandatory security issues such as polyinstantiation and handling the inference problem. Polyinstantiation occurs when two users of the database at different security levels see different values for a single entity in the real world. Her security policy also includes rules to assign security constraints and a means to handle these rules.

The secure model SORION (Secure ORION) is used as the basis to formulate a security policy for the database system. SORION evolved from ORION by incorporating security levels for all entities and enforcing security properties that must be satisfied. The paper discusses the security properties for objects, classes, methods, multiple inheritance and aggregate classes and objects. Relationship objects, which represent multimedia information and the link from one object to another, are also included in the security property list.

The discussion on polyinstantiation is brief, but covers the various types of polyinstantiation to be considered in an object-oriented system (such as Object/Value polyinstantiation or Class/Structure polyinstantiation). Several scenarios for polyinstantiation and possible solutions are given.

User inference is addressed lightly, with the author suggesting an inference controller (augmentation of an inference engine with an object-oriented database and rule base to perform query modification). She also suggests that the database and constraints can be expressed in a logic programming language with support for representing and manipulating objects to implement some kind of inference controller. Using an inference engine for database security is also covered in [64, 88].

Another approach used for security in object-oriented systems that concentrates
on the computation model as well as the data model is that taken by Keefe [44, 45]. The SODA (Secure Object-Oriented Database system) project develops a multilevel secure object-oriented database management system using an object-oriented data model. It focuses on security issues affecting the application interface and incorporates what Keefe calls "distributed cooperating-operating objects". An object represents a distributed computation element. Objects cooperate by sending messages and executing methods in response to receiving messages. Performing multilevel updates has long been a difficult problem because of certain restrictions imposed by the security model being used. Many models restrict a subject to write data at a single sensitivity level only (the SeaView Model [25] for example). If a multilevel update is required, several subjects must cooperate. This cooperation is not specified within the model. With the SODA model, multilevel subjects cooperate to allow multilevel updates to be performed within an untrusted application. This model has the following properties:

- It enforces information containment and security label integrity.

- It covers the computational model as well as data classification and access.

- Classification of objects is based on an inheritance lattice.

- The protected data in this model are the instance variables and the objects.

- Access is arbitrated by a TCB (trusted computing base).

- Data classification is based on inheritance. However, this classification specifies a range of sensitivity levels for an object.
• The current classification level of a method activation represents the sensitivity level of data which the activation has read or has had access to.

Both of the aforementioned approaches to security in an object-oriented database system base data classification on inheritance. Object in a class share the same classification constraint. The two methods use the existing structure of the system as the basis for classification. The idea is that using inheritance of constraints ensures that the constraints are only specified when a change must be made.

**Problem Statement**

Despite the fact that the interest and research in the area of object-oriented databases is growing very rapidly, there is currently no standard object-oriented data model - there is not even agreement on what standard object-oriented concepts should be. Chapter 2 introduces some commonly accepted “core” concepts for object-oriented systems, but these are by no means a standard set of concepts.

Although data security strategies are well developed in the field of operating systems and conventional database systems, very little has been done involving security and transaction management in object-oriented systems. It is clear that with the advent of new storage technologies and advanced design systems, the relational systems will no longer be satisfactory for advanced applications such as CAD/CAM, CASE, and multimedia information systems. That is why object-oriented systems are being investigated. The integration of object-oriented concepts within the database environment is a significant advancement in the evolution of database systems. However, this combination also raises a new issue, that of protection of the information in an object-oriented system.
As summarized in the preceding section, much research has been done involving security in relational database systems. Object-oriented systems introduce new challenges that were not present in the relational database world. Object-oriented databases have a much richer structure and support semantic relationships, both of which make the problem of providing adequate security much more complex and interesting. For example, objects can be shared among several different objects. Consider several multimedia documents which all share a common figure. When considering security constraints, one has to allow for the figure to be shared by several different objects but must also provide some access control. Also objects can be composed of other objects. This, too, must be addressed when dealing with security. Objects encapsulate not only their state (attributes) but also their behavior (methods). Access to the methods must also be monitored. This is not a problem in the relational system since methods are not included in the relational model. Therefore, security policies that work in a relational system need to be extended/modified in order to work in an object-oriented system. Because of the hierarchical nature of object composition (see Chapter 3), security constraints must be incorporated differently than in a relational system. Also, the object itself can enforce security, whereas in a relational system, a trusted front-end or some other code is necessary to enforce security.

The existing approaches to security in object-oriented systems base the data classification on the structure of the system. Security levels or constraints are inherited from class to subclass. Consider the following example: Suppose we have a class LIST which has a subclass SORTED-LIST. There may be applications where an instance of LIST might need to be SECRET and an instance of SORTED-LIST might be unclassified. The data itself needs to be classified, not the structure. By
imposing the restriction that a subclass must inherit its security classification label from its class, we cannot have an unclassified instance of a sorted list if we have a secret classification label on the class LIST. We conclude, therefore, that we must secure the data in the database by some other means than just inheriting the security level from the parent class.

Not all databases need to be converted to object-oriented systems. There are certain applications that are best served by using the conventional database systems. For others, however, an object-oriented approach is more natural, and therefore more appealing and more efficient. In particular, CAD/CAM applications and multimedia databases are well suited for an object-oriented approach. In these systems, as well as others, however, transaction processing must be addressed. Often, in a CAD/CAM or multimedia application, long running transactions are common, which may take hours or even days to complete. So the usual method of locking everything dealing with the conventional transaction does not work in these environments. Also, as transactions progress, certain security constraints must continually be enforced, and dynamic modifications in security labels are necessary.

Main Contribution of this Thesis

There are four main contributions in this research. First, a new object-oriented data model is proposed. Second, a security policy based on securing the data without forcing the security to be inherited along the class/subclass hierarchy is presented and then incorporated into the data model, thus describing a complete approach to multilevel security in an object-oriented database environment. Third, a transaction model is described which is used in conjunction with the data model and security
policy to provide secure transaction processing within the object-oriented database. Fourth, secure query processing in our system is addressed.

Overview of this Thesis

We begin the formal investigation in Chapter 2 with a review of important works and developments in the area of database research. This review is divided into seven significant parts: relational databases, entity-relationship concepts, the semantic data model, the functional data model, object-oriented models, a comparison of the different approaches, and a summary.

We have developed an object-oriented data model, which we present in Chapter 3. Some basic object-oriented concepts are covered in more detail, and then the specifics of the model are introduced, including an aggregate object hierarchy and a version hierarchy. We introduce data model specifications for the four hierarchies: class/subclass, class/instance, composite object, and version. We then cover how this new model satisfies the requirements and design objectives in an object-oriented environment.

Our security model is described in Chapter 4. The chapter begins with an overview of existing approaches to object-oriented security. We then introduce security properties for objects, classes, methods, and versions. Our security model is then integrated into the data model introduced in Chapter 3. We introduce a method invocation algorithm, which monitors all information flow within the model. We also provide a formal correctness proof of the invocation algorithm.

Our transaction model is described in Chapter 5. A brief review of nested transactions is presented, and then our application of nested transactions to an object-
oriented setting is described. We present locking protocols for the class/subclass, class/instance, composite object, and version hierarchies. We provide a transaction security policy to ensure secure transaction processing.

Chapter 6 deals with secure query processing. The notions of a secure view and secure query graph are presented in the context of an object-oriented database. Simple, content-based and context-based security constraints are enforced by augmenting a rule base with the database to perform query modification. By retaining environmental information in the rule base as well, user inference is addressed.

Chapter 7 summarizes this thesis. It also presents future work and additional research issues involving object-oriented databases and security.
CHAPTER 2. RELATED WORKS

The desire to access large amounts of data efficiently led to experimentation with the physical implementation of storing the data. The network and hierarchical data models of the 1960s grew out of this research. A network model uses a directed graph (a network) with links. Therefore, all relationships are restricted to be binary, many-one relationships. The hierarchical data model is a network that is a collection of trees where all links point from child to parent. From this point in time, two separate directions were pursued in database research: the relational model and semantic database models.

The Relational Model

Codd's paper in 1970 [14] changed the direction of database research by introducing the relational database system. This system separates the physical implementation from the logical data representation. The relational model supports powerful but simple declarative languages to perform operations on the data. While the query languages for the network and hierarchical model require extra effort on the part of the user by forcing the user to state how to get from one record to the next, the declarative languages based on the relational model require the user to state only how the answer is related to the data.
Relational databases are designed to handle data processing needs such as banking, airline reservations, and inventories. Information is stored in tables (relations). These tables consist of records (tuples), where each record can have several fields (attributes). An example of a relational database can be found in Figure 2.1. Users can query relational databases using languages based on relational algebra.

Although relational databases are in widespread use, they have several shortcomings. The following shortcomings for conventional database systems (relational and past generation systems) are listed in [3, 49]:

1. Many applications such as CAD/CAM design systems and multimedia information systems require manipulation of complex objects. These nested entities cannot be modeled by conventional data models. The relational model requires normalization, thus "flattening" the complex objects. Often an entity must be broken into several components, thus creating several levels of indirection.

2. Conventional database systems support a limited set of atomic types. In particular they cannot support objects like images, sound, and textual documents as found in multimedia environments.

3. Relationships such as generalization and aggregation are not supported in the conventional data model. If users wish to incorporate these ideas, they must do so explicitly because the database itself does not.

<table>
<thead>
<tr>
<th>SS#</th>
<th>Name</th>
<th>Address</th>
<th>BDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-11-1111</td>
<td>Howell D. Weatherbee</td>
<td>12 W. 5th, Austin, Tx 76432</td>
<td>1/23/59</td>
</tr>
<tr>
<td>222-22-2222</td>
<td>Aretha Pineboughs</td>
<td>123 Main, Menlo, CA 23457</td>
<td>4/25/62</td>
</tr>
<tr>
<td>333-33-3333</td>
<td>Patty O'Furniture</td>
<td>324 Oak, Aimes, KS 43295</td>
<td>9/4/55</td>
</tr>
</tbody>
</table>

Figure 2.1: An example of a relation
4. Conventional systems are too slow when it comes to fetching and storing individual fields. This is seen in CAD tools which use record structures from programming languages to get the speed instead of using a relational database. Compute-intensive applications are not suited to the conventional systems available. In a relational system, connections between entities are established using a key. Address translations are required to get to the tuples representing these entities. However, if the entities themselves are stored in memory, pointers can be used. Conventional systems deal with files very efficiently, but this efficiency results in a degradation of performance involving memory computations.

5. Query languages and programming languages must be integrated. Applications programs are written in languages such as COBOL or FORTRAN, yet the query languages used are SQL, QUEL, etc. These languages are vastly different, yet are expected to perform in unison in the database environment. This impedance mismatch led researchers to develop the fourth generation languages.

6. In design environments, users often retrieve an entity and hold that entity for a long period of time. These long running transactions are not handled well in conventional systems. Most commercial databases systems currently use locking as the method of concurrency control. The probability of deadlock is proportional to the length of the transaction raised to the fourth power [51]. Therefore, the longer the transaction, the higher the risk of deadlock in the system. Interactive design environments also lead to the notion of versions and change notification, which conventional database systems do not support.
The Semantic Models

Semantic database modeling gained popularity in the late 1970s and the early 1980s as a means to include richer data structuring capabilities into the database applications and to more naturally model the relationships inherent in data normally stored in a database. With the ever-increasing demand for database systems with richer semantics comes the need for more expressive conceptual data models to meet these demands. The semantic models were introduced originally as design tools: design the schema using a high-level semantic model and then translate into an existing conventional model for implementation. Semantic models are more complex than the relational model and are also more navigational.

Semantic models have a set of generic properties in common. The primary properties are the ability to explicitly represent objects, the use of the two abstractions generalization and aggregation, the ability to express semantic relationships among objects, the use of classification and association relationships and the use of derived schema components.

Generalization is a process that abstracts similar objects into higher-order types. For example, an object of type PERSON might represent a generalization of the objects of the type EMPLOYEE and STUDENT. The object PERSON contains those characteristics common to all people, which includes employees and students. The attributes of PERSON are automatically defined in STUDENT and EMPLOYEE since these are both specific examples of PERSON. STUDENT and EMPLOYEE can each have additional attributes not found in PERSON, thus specializing the definition. Generalization and specialization are often mentioned in conjunction with each other. For example, STUDENT is a specialization of PERSON, while PERSON
is a generalization of both STUDENT and EMPLOYEE. This supertype/subtype relationship is called the IS-A relationship, and the family of relationships is called the IS-A hierarchy. An example of this hierarchy can be found in Figure 2.2.

Aggregation, another property found in semantic data models, exists in the relational model as can be seen when attributes are grouped to form a relation. Semantic models use this property to allow objects to be grouped to form higher level objects. For example, the objects ADDRESS and NAME might be grouped together to form the object PERSON. Thus aggregation provides a method to construct a new type from existing types. The aggregation abstraction is represented by the IS-A hierarchy and the IS-PART-OF hierarchy for aggregate objects. For example, suppose the PERSON type has address as a field, where address is of type ADDRESS, another defined type. Then the address field is part of the PERSON type. Figure 2.3 illustrates this idea.

Classification provides a means to collect similar objects into a higher level object class. Essentially, a description, or template of a class of objects is first presented, and then instances of this class are created. This abstraction is represented by the IS-AN-INSTANCE-OF hierarchy, as depicted in Figure 2.4.

Association (or grouping) is found in many semantic models also. It is used
to build sets of elements of an existing type. For example, the set of instances of EMPLOYEE is an association of PERSON objects as is the set of instances of STUDENT. Association provides the means to define a type whose value will be a set of objects of a particular class or type.

Semantic data models have many advantages, which are enumerated in the following: [41, 67]

1. Economy of Expression and an Increased Separation of Conceptual and Physical Components. Since the semantics exist within the data model itself, users are able to extract information more easily using a semantic data model. For example, users in a relational system must know the attributes of relations and know which ones to join over to do many queries. The user must
perform complicated projections and joins to extract the necessary information because of the inter-relational connections among relations. In the record-oriented models, the user must simulate pointers to traverse from one relation to another. The semantic relationships among relations exist in the user's query. However, in the semantic data model the query is much more compact. Since the data model expresses the semantic relationships, the user is free from specifying these relationships in the query itself. In the semantic models, the attributes themselves may be used as direct conceptual pointers.

2. **Integrity Maintenance.** Since certain objects are connected through relationships, changing one of these related objects immediately impacts the other objects connected to it. There is no need for the user to monitor and maintain intra-object consistency.

3. **Modeling Flexibility.** Since semantic data models use several types of abstraction, they allow the user to model and view the data on many levels. Thus the user has more options when modeling the real world. For example, the concept of specialization allows the user to view objects in a very specific manner, while generalization allows the user to view objects more abstractly.

4. **Modeling Efficiency and the Availability of Convenient Abstraction Mechanisms.** While using a semantic data model, it is unnecessary for the user/designer to implement on a low level. If one object references another, the behavior of the second is encapsulated within the object itself. The schema can be viewed at several levels of abstraction. One can view objects at the most general level – the class level – without looking at the actual structure
of the object. One can also view information about the object itself, isolated by the notion of encapsulation inherent in object-oriented environments. Users can also view objects in terms of other objects (derived components). This is similar to work in language theory involving abstract data types.

5. **Decreased Semantic Overloading of Relationship Types.** In relational models, several different types of relationships must be represented using the same constructs. The semantic models, however, provide many methods for representing data inter-relationships. One of the main objectives of semantic model research has been to provide a family of constructs for representing the kinds of information that the relational model can represent only through constraints.

Several models have emerged, with no particular one gaining general acceptance. The main focus in the semantic data model appears to be on the structural aspects of the models, including such things as class hierarchies, generalization, aggregation, and attributes of and relationships among objects. Secondary importance has been given to the behavioral aspects of these models.

**The entity-relationship model and its concepts**

The entity-relationship (ER) model proposed by Chen [13] is actually a semantic model and is the basis for many of the semantic data models that follow. Schemas based on this model are represented by graphs, similar to the network model, using two primary modeling constructs – the entity and the relationship. An ER schema consists of entities and relationships connecting these entities. It allows for representation of entity sets (abstract types), relationships (aggregation), and attributes
for entities. Attributes are either printable or non-printable. Printable attributes are represented directly in the ER model. A non-printable attribute (one which is non-primitive and consists of another object) must be represented using relationships. These relationships can be one-to-one, many-to-one, one-to-many, or many-to-many. An example of the ER model can be found in Figure 2.5.

The ER model is the first semantic data model to focus on relationships instead of attributes. This model attempts to combine the features found in the network and relational models to provide multiple levels of abstraction, although the only abstraction found in the original model is aggregation. The ER model has been used primarily as a database design tool, prior to implementation in the actual model.
of the DBMS used. This model differs from the later semantic models in that the ER model does not represent IS-A relationships (supertype/subtype relationships). Recent research on extending the ER model has resulted in IS-A relationships being included [6, 80].

**The functional data model**

The functional data model (FDM) [47, 74] is another example of a semantic data model. It is centered around attributes, or functional relationships, unlike the ER model which uses aggregation as the general modeling philosophy. FDM allows objects to be connected directly with attributes without using aggregation and grouping. These attributes can be either single or multi-valued. Unlike the original ER model, FDM supports the IS-A relationship. FDM also includes derived schema components as part of the semantic model. An example of the functional data model can be seen in Figure 2.6.

**The semantic database model**

The Semantic Database Model (SDM) [38] is another example of a semantic data model centered around attributes. However, SDM is richer and more complex than FDM. Most other semantic models provide the constructs to develop more complex constructs, but SDM offers a full set of modeling facilities. Whereas the general modeling philosophy of the ER model is aggregation, and for FDM is the use of attributes, SDM uses both attributes and a type constructor. It uses a grouping constructor and incorporates many modeling constructs into the single abstraction of a class. SDM relies quite heavily on the abstraction of classification as well as
Figure 2.6: An example of the FDM model

that of association, more so than those of aggregation and generalization. SDM also supports derived schema components. Figure 2.7 is an example of the semantic database model.

Table 2.1 summarizes some important features of the above mentioned semantic data models [67]. Table 2.2 provides a comparison of the relational model to the

Figure 2.7: An example of the SDM model
Table 2.1: Data model features

<table>
<thead>
<tr>
<th>Model</th>
<th>Relationship Representation</th>
<th>Standard Abstraction</th>
<th>Network vs. Hierarchy</th>
<th>Derivation/Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER</td>
<td>Independent and Tables</td>
<td>Aggregation</td>
<td>Strong Network</td>
<td>No</td>
</tr>
<tr>
<td>FDM</td>
<td>Functions</td>
<td>Association</td>
<td>No direct support for either</td>
<td>Functional</td>
</tr>
<tr>
<td>SDM</td>
<td>Independent and entity (classes)</td>
<td>Generalization</td>
<td>General hierarchy present</td>
<td>Elaborate and varied</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of relational to semantic data models

<table>
<thead>
<tr>
<th>Model</th>
<th>Query Language</th>
<th>Complex Objects</th>
<th>Generalization</th>
<th>Aggregation</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>Declarative</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Impedance Mismatch</td>
</tr>
<tr>
<td>Semantic</td>
<td>Navigational</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>–</td>
</tr>
</tbody>
</table>

The reader is referred to [41, 67] for further details on semantic databases and the various models developed.

The Object-Oriented Model

Background

About the same time work was progressing on semantic data models, researchers were continuing to develop object-oriented programming languages. Simula-67 is considered to be the first object-oriented language. Since its development, researchers
in the area of programming languages have worked with two options for developing object-oriented programming languages [49]. One option is to extend existing languages with object-oriented capabilities. Examples of languages developed through these extensions include C++ [78, 75], LOOPS [10], Flavors [60, 46], and CLOS [61]. The other option is to develop new languages, such as Smalltalk [32, 33], Traits [17], Trellis/Owl [70] and Eiffel [57]. Many of the ideas present in object-oriented languages can also be found in semantic data modeling issues. In addition to programming languages, object-oriented ideas have presented themselves in the area of artificial intelligence (evident in Minsky's frame idea [58]). The concepts found in object-oriented programming languages have begun to merge with the ideas of semantic data modeling, resulting in the birth of object-oriented database systems. Commercial systems such as ORION [48], GemStone [54], VBASE(now Ontos) [2, 85] are now available. Several industrial research prototypes are also available, including IRIS [28], O2 [52], and JASMIN [55]. University research prototypes include such systems as OZ+ [86], CACTIS [40], ENCORE/ObServer [39], ADVANCE [9], and EXTRA [12]. Before discussing object-oriented database systems, semantic data models and object-oriented programming languages are compared.

Semantic data models and object-oriented models are similar in that they both provide the ability to construct complex data by interrelating entities. Although many of the concepts in semantic data models are similar to those found in object-oriented programming languages, differences exist. Perhaps the main difference in the two approaches is that the semantic models capture the structure of objects whereas object-oriented models capture the behavior as well as the structure of objects. Therefore, when using the object-oriented approach in database systems, a database entity
can locally encapsulate a procedure or function. This gives the database user the ability to express a wider class of derived information than could be accomplished with the semantic data models.

The shortcomings of the relational model and other conventional database systems led researchers to investigate the possibilities of using object-oriented systems instead. People working with object-oriented systems want to add database functionality for advanced applications. As a result, object-oriented database systems are receiving significant attention.

Core concepts

At the present time, no standard object-oriented model exists. Various properties of object-oriented models have been proposed. A very concise description of the main characteristics of an object-oriented data model is given in [3, 5, 49]. These "core" concepts represent the commonly accepted and fundamentally important ideas found in areas involved in object-oriented applications:

1. **Object and Object Identity.** In object-oriented systems, all real-world entities are modeled as objects. Each object has an identity which is separate from its value and is identified by a system-wide unique identifier. This specification of the object in terms of an identifier is in contrast to the value-based specification of the relational system. Therefore, object-oriented systems are navigational in nature and do not support declarative queries (as mentioned in Chapter 1).

2. **Attributes and Methods.** Each object has attributes (instance variables) that make up the private state of the object. The methods of the object consist
of code that manipulates and/or returns the state of an object (the value of the instance variables).

3. **Encapsulation and Message Passing.** The methods and the instance variables mentioned above are encapsulated in the object. This means that they are not visible from outside the object. Thus, messages are required for objects to communicate with each other. Messages and the accompanying parameters constitute the public interface to an object. Only the operations invoked via incoming messages can be performed on the object.

4. **Classes.** Similar objects are grouped together into a class. These objects all have the same instance variables and the same methods. Objects in a class (called instances of that class) respond to the same messages by invoking the corresponding methods. An object may belong to only one class.

5. **Class Hierarchy and Inheritance.** Putting all like objects in a class decreases the amount of redundant information that must be stored. Classes themselves form a hierarchy, adding to the information hiding capability of an object-oriented system. This class hierarchy represents the IS-A (class/subclass) relationship. A subclass of a class represents a specialization of that class; while a superclass of a class denotes a generalization of the class. A class inherits all instance variables and methods from its superclass and may have additional properties specified also. Therefore, a subclass is more specific. If a class inherits from only one superclass, this is an example of single inheritance. If, however, the class is allowed to inherit from more than one superclass, it is called multiple inheritance.
Table 2.3: Relational concepts vs. object-oriented concepts

<table>
<thead>
<tr>
<th>Relational Concepts</th>
<th>Object-Oriented Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>Collection of Classes</td>
</tr>
<tr>
<td>Tuple</td>
<td>Instance of a Class</td>
</tr>
<tr>
<td>DBase Schema</td>
<td>Class Hierarchy</td>
</tr>
<tr>
<td>Record Type</td>
<td>Class</td>
</tr>
<tr>
<td>Procedure Call</td>
<td>Message</td>
</tr>
<tr>
<td>Procedure Code</td>
<td>Method</td>
</tr>
<tr>
<td>Field, Attribute</td>
<td>Instance Variable</td>
</tr>
</tbody>
</table>

Table 2.3 can be used to loosely compare the terms used in a conventional relational system and those terms used in an object-oriented environment.

Object-oriented systems vs. object-oriented database systems

Before discussing the differences between object-oriented systems and object-oriented databases, we should first take a look at the events leading to object-oriented database systems. After realizing that conventional models had serious shortcomings, research in the database field has taken two different branches: extending existing conventional databases to become object-oriented systems, or modifying object-oriented systems to include database functionality. Codd [15] extended his relational data model by specializing the roles of different relations, thus allowing the modeling and manipulation of additional semantic relationships. His new model is called RM/T. The relational model is also extended resulting in the well-known POSTGRES [76]. Other attempts to enhance the relational model include [30, 71, 72, 87]. Proponents of the extended relational approach argue that it is based on familiar database technology with an industry-wide standard.
The other approach ties in with semantic data modeling and object-oriented systems. This object-oriented approach starts with an object-oriented data model and adds the necessary capabilities to perform database functions. The object-oriented approach models more closely the real world, including general data types, complex (nested) objects, and compute-intensive applications. Proponents of the object-oriented approach argue that with the extensions added to the relational model, the end result is almost an object-oriented system (lacking the notions of encapsulation and inheritance). The extended relational systems are also awkward and inefficient. Hence, an object-oriented approach should be used.

The differences between an object-oriented system and an object-oriented database system are discussed in [3]. Following is a list of the concepts missing in an object-oriented system to make it an object-oriented database system:

1. **Set Programming.** In most programming systems (object-oriented systems included) sets are not implemented at the system level. Sets are represented as lists of elements. But a database is a collection of sets. Object-oriented database proponents need to find a way to treat sets in a uniform fashion.

2. **Persistence and reliability.** Automatically managed virtual memory is crucial to the development of object-oriented systems. These systems allow efficient representation and sharing of complex objects via pointers. However, most object-oriented systems do not offer an analogous storage organization for persistent objects. Thus, to keep data from one session to another, the programmer must flatten the rich structures of objects in virtual memory and place them in files or in a conventional database.
Object-oriented systems also provide no protection in regards to hardware or software failure; there are no roll-back features or transaction management features.

3. **Sharing.** The majority of object-oriented systems that exist today are single user. The very nature of a database implies sharing. Thus, object-oriented systems must be extended to multiple users.

4. **Managing large amounts of data.** Since these object-oriented systems run in main or virtual memory, they do not handle large amounts of data very well. They do not use indexing, smart buffer management schemes, clustering of objects on disk, or intelligent query optimization.

Table 2.4 summarizes the differences between an object-oriented system and an object-oriented database system.
A Comparison of Semantic Models to Object-Oriented Models

Most certainly object-oriented modeling is related to semantic modeling. Both provide the necessary means to construct aggregate objects by interrelating lower level objects. Both are concerned with the semantic relationships that exist between objects. However, object-oriented models are different in that they also model the behavior of the objects. This means that an object in the database can have a complex procedure as a property, or attribute, which is locally encapsulated.

Semantic modeling depends heavily on building aggregate objects using abstractions such as aggregation and generalization. These mechanisms are satisfactory for applications similar to those found in the business world. Object-oriented models are aimed at more complex applications involving such things as design databases or multimedia databases. These applications involve much larger objects which must be handled efficiently. The issues introduced by these large objects include transaction management for long running transactions, nested transactions and efficient retrieval of large objects.

Summary

This chapter has provided an overview of existing conventional and semantic data models and covered the core concepts that should be present in an object-oriented data model. A comparison of semantic, relational, and object-oriented data models has been provided. Semantic models clearly provide richer data structuring capabilities than do the conventional systems. The current conventional data models lack the necessary abilities to support data abstraction, inheritance, and constraints.
Semantic data models support these. Object-oriented data models go one step further and support not only the structure of the objects but also the behavior. Chapter 3 will introduce our object-oriented data model.
CHAPTER 3. THE OBJECT-ORIENTED DATA MODEL

As previously mentioned in Chapter 2, no standard data model exists for object-oriented database systems. The core concepts previously presented summarize the commonly accepted and fundamentally important ideas that must exist in an object-oriented model. Additional concepts must be added, however, to capture certain semantic relationships that exist in databases and to specify constraints that may need to be enforced on the data used for advanced applications. For example, the core model does not address aggregate objects or versions. In this chapter, we introduce an object-oriented data model in several steps. In Chapter 4 we introduce the security policy that can be incorporated into the data model.

Objects and Object Identity

As mentioned in Chapter 2, conceptual entities are modeled as objects. Each object has an object identifier which is unique system-wide. Objects in the database are related to other objects in the database through various semantic relationships. An object is represented by a triple,

\[ O = (O_{id}, A, M) \]

where \( O_{id} \) is the system identifier, \( A \) is the set of attribute/value pairs associated with the object, and \( M \) is the set of methods associated with the object.
An object may be:

(1.) a primitive object, the lowest level object (it does not consist of other objects),
or

(2.) composed of other objects (either a composite or aggregate object, or a set object), or

(3.) a version of another object (related using the version-of relationship).

More details on composite objects and versions are introduced later in this chapter.

Attributes and Methods

Associated with each object is a set of attributes which describes properties of the object. The state of the object is determined by the value of its attributes. These attributes or instance variables may be primitive objects, such as integer values or strings, or they may be complex objects, in turn containing other instance variables. In addition, an attribute may represent an entire set of values instead of one single object. For example, if we have a class PERSON with an attribute children, the value for children would actually be a set with elements of type PERSON. We refer to this as a homogeneous set. Ordinarily, elements of a set are of the same type. However, we introduce the notion of a heterogeneous set to allow a set attribute to consist of elements of different types. For example, if we have an attribute CLIENT in a class EMPLOYEE, the value for CLIENT might be a set of clients consisting of individuals (type PERSON), companies (type COMPANY), and schools (type SCHOOL).

Communication with an object must be only through the use of message passing. Each object has associated with it a set of methods which encapsulates the behavior
of the object. Each method is actually executable code that is run when a message arrives at the object for that particular method. When an object receives a message, it invokes the corresponding method for that message and may return an object in response. Messages are represented by the triple:

\[ M = (M_A, P, V) \]

where \( M_A \) is the method name to activate, \( P \) is the formal parameter list, and \( V \) is the return value (which may be \texttt{nil}).

It is possible that the method invocation might need to invoke another method at another object. It would then send a message to this object, which would execute a method also. The methods and the instance variables are not visible from outside the object.

**Classes**

Objects with similar properties are grouped together to form classes. Since classes are logical entities in the database, they are also considered objects. Classes can intuitively be thought of as templates for the objects of interest in the database. A template specifies the instance variables (attributes) that exist for each object in the class and the methods that the object will execute in response to incoming messages. For example, suppose we have a class object PERSON. Attributes in this class might be \texttt{name}, \texttt{social security number} and \texttt{address}. All objects belonging to this class, therefore, must have a name, a social security number, and an address. A class is represented by the following 4-tuple:

\[ C = (C_id, A, M, S) \]
where \( C_{id} \) is the object identifier for the class, \( A \) is the set attributes/value pairs (including class variables), \( M \) is the set of class methods, and \( S \) represents the list of direct superclasses for class \( C \).

As mentioned in the previous section, messages must be sent to objects to perform any operation on the object, whether it be reading the value of an attribute, modifying an attribute value, or actually creating an object. Because similar objects belong to the same class, the methods these objects use are the same and can be stored in the class definition itself, thus reducing redundancy in the code required. It should be emphasized that an object may belong to only one class.

One may wonder how an object can be created if a message must be sent to an object that does not exist yet. This is accomplished by sending the message to create an object to the class that serves as the template for that object. When a create message is received by the class, the class instantiates a new object belonging to that class. This new object is related to the class by the IS-AN-INSTANCE-OF relationship. This newly instantiated object has its own set of instance variables and values but shares the methods with the class.

**Instance variables**

All objects in a class are instances of that class. When speaking of instance variables and classes, one has to consider two different types of variables. One type, which we simply call instance variable, makes up a set of attributes that describes the form that objects in the class must follow. For example, all objects of type PERSON must have a name. This instance variable can be further divided into two categories. If the instance variable represents information that will normally be
changed for each instance of the class, we call this a *value instance variable*. For example, for an instance of class PERSON, we expect to enter a string value for the name, a string for the social-security number, and other more complex information for the address (the address is actually a pointer to another object containing several string values). Certain value instance variables could be shared. If two people in our database lived at the same address, the address pointer could point to the same object which represents the address.

It is also possible to have an instance variable that would represent the default value for any instance variable in an instance which does not have a value assigned. We call this a *default instance variable*. In a database environment, this is particularly useful due to the fact that so many database applications see this type of redundancy. The default value could easily be passed to the instance when the instance is created. The method to create a new instance could “save” the default value and pass it when necessary. An example of this would be if we wanted all objects of type PERSON to contain the default value USA as part of the address. Then as each instance is instantiated, USA is passed as part of the address. If we then instantiate a person and want to change the default from USA to Canada, for example, this can be done by explicitly sending the value of Canada for that part of the address instead of letting the default set the value.

There are two ways to implement the notion of a default instance variable. One way is simply to have the default value passed to the instance when it is created. For example, USA could be passed as the value of *country*. Each instance would then have an individual value for *country*. If the default value is changed, all instances created prior to the new value would use the old value unless explicitly updated.
The other way is to store the default value in the class and have all instances point to this value in a shared fashion. If the default value is changed (whether it be by someone modifying the class or by allowing any instance using the default to modify the value), all other instances see the change. The old value of the default no longer exists. If an instance is to have some value other than the default, then this instance will not share the value, but will create and point to a new value. This allows the maximum of sharing but still allows individual variation.

Class variables

The other kind of instance variable we shall call a class variable. This is an instance variable that exists in the class template itself. The idea of a class variable can further be broken into two types. The first type, which we call an aggregate class variable, represents some type of aggregate property about all of the instances. For example, suppose that class PERSON has a counter to keep track of how many instances have been created. This counter is accessible by all instances of the class, and resides in the class template. Another example is some statistic that resides in the class. Suppose class EMPLOYEE has an aggregate class variable to keep track of the average salary which changes as salaries change and employees are added and deleted. In either case, if the value is modified, all instances of the class see the new value.

An alternative approach for handling aggregate class variables would be to treat them as dynamic class variables. The values are not stored. Instead, the value is calculated whenever a method is invoked that accesses any aggregate class variable. The decision to treat the aggregate class variable statically or dynamically depends
on the application. If the database environment is such that a significant number of queries reference aggregate class variables, the computation time will be quite large. In this case, it would be more desirable to store the values for immediate access.

The second type of class variable, which we call a shared class variable, is an attribute which resides in the class and is shared by all instances. For example, if we knew all people in our database were from the USA, we could make this a shared class attribute instead of a default instance variable. This would reduce storage since the value is always the same for each instance created. If the shared class variable is modified, all instances see the new value. Figure 3.1 illustrates the concepts of class variables and instance variables. The value of USA is a default instance variable that is passed to the Aretha Pineboughs instance of this class. *Number of people* and *average age* are aggregate class variables.

The idea of a class is very important, especially in the database environment. It acts as a grouping constructor and relates all objects to a class with the IS-AN-
INSTANCE-OF relationship. This group is often the target of a query. Without classes, queries would have to be directed at all objects in the database. However, classes allow us to narrow the domain of queries by sending them only to those groups of objects that the query needs to access in order to provide the necessary response. Classes also hold the methods and class variables, thus reducing storage, which is especially important in a database environment. Since classes act as templates for instances, some integrity checking can also be performed while instantiating objects and initializing their instance variables.

Class Hierarchy and Inheritance

Class hierarchy

It is often desirable to create a class which is more specific than one that already exists. The new class has additional properties (instance variables and methods) that the original class does not have. This new class is called a subclass of the original class, and the original class is called a superclass of the new class. This class/subclass relationship is reflected in the class hierarchy (the IS-A hierarchy). This hierarchy reflects generalization and specialization. The subclass is a specialization of the class, because it contains everything the class does with additional properties which make it more specific than the class. The class is a generalization of all subclasses because it defines the similarities that exist among all of the subclasses and acts as a grouping constructor based on these similarities. The IS-A hierarchy depicts the structure or relationships among classes. For example, consider the class PERSON introduced earlier. We can now create a subclass of PERSON called EMPLOYEE that contains additional instance variables salary and department and additional methods to ac-
cess these new attributes. Figure 3.2 illustrates this idea. In this figure, aggregate classes are represented by a circle with a $+$, sets by a circle with a $\ast$, and primitive (printable) attributes with a flattened oval. The solid edges to attributes represent ones defined at that class, and the dashed edges indicate inherited attributes.

EMPLOYEE inherits all of the attributes and methods that are found in class PERSON. Inheritance is a reusability mechanism which allows classes and instances of these classes to share behavior. Essentially, inheritance allows us to define new classes that inherit properties from existing ones. If the new subclass inherits all attributes, this represents full inheritance, whereas if the subclass is allowed to inherit
only certain attributes, it is partial inheritance. In our model, we consider only full inheritance.

There are two further types of inheritance, which are classified as to which class the subclass can inherit from. The first type, single inheritance, allows a subclass to inherit instance variables and methods from only one parent class, although the subclass may add instance variables and methods at its own level. The class EMPLOYEE did just this. EMPLOYEE has the attributes name, social security number, and address (inherited from PERSON), but it also has the additional attributes salary and department. The second type of inheritance, multiple inheritance, is a natural extension of simple inheritance. Multiple inheritance allows a subclass to inherit from more than one parent class. For example, if we have subclass EMPLOYEE of PERSON and subclass STUDENT of PERSON, we may create a new subclass STUDENT_EMP, which is a subclass of both EMPLOYEE and STUDENT. Therefore, it will inherit the attributes from both classes.

It should be noted that a subclass can inherit an attribute or method from a class and make additional modifications to either at its own level. The subclass can change the name of an attribute, or the characteristics of an attribute. If, for example, a subclass inherits an address attribute which allows for three lines on the address (name, street address, city/state/zipcode), it might modify the zipcode from five digits to nine, or it may add a field to include business or organization as part of the address, now allowing for four lines in the address. The subclass can also modify the methods it inherits. It can rename the method or it can modify the code for the method. A subclass may supply additional code so that the method has increased functionality at the subclass level. For example, a method to display
certain attributes in a class might be modified to display additional attributes found in the subclass (such as modifying a method to display an extra line in the address or a nine digit zipcode).

Class variables as well as instance variables are inherited. Aggregate class variables inherited by a subclass represent a finer granularity than those in the class. For example, if class PERSON had an aggregate class variable called *average-age*, this variable represents the average age for all instances (direct and logical) of class PERSON. If a subclass EMPLOYEE is now created, it also has an aggregate class variable called *average-age*, which represents the average age of all people who are employees. Shared class variables are handled in a similar fashion. The subclass will inherit not only the variables, but also the value of certain class variables. Default instance variables must have the values passed to the subclass so the value is available when instances of the subclass are created. Both default instance variables and shared class variables may have the value modified by the subclass at the time of inheritance.

If one visualizes the class hierarchy, single inheritance results in a tree. Multiple inheritance, on the other hand, results in a lattice-like structure. Since a subclass inherits properties from its parent class, and this parent class inherits properties from its parent class, it follows that a class actually inherits properties from all of its ancestors.

Now that the class hierarchy has been introduced, we can formalize the data model by defining the following data model specifications (DMSs):

DMS1. All objects are either primitive objects, set objects or aggregate objects.

DMS2. All classes belong to the IS-A hierarchy (the class hierarchy) which defines a
partial order on the classes. The IS-A relationship is a relationship which can be one-to-one, one-to-many, many-to-one, or many-to-many. The latter two allow for multiple inheritance.

DMS3. If C is a class and C' is a subclass of C, C' is related to C in one of two ways:

(a) If C' is an immediate descendant of C, we say C' is a direct subclass of C. (i.e., there are no subclasses defined between C and C'). C is a direct superclass of C'.

(b) If there exists some C'' such that C'' is a direct subclass of C and C' is a subclass of C'' we say C' is a general subclass of C, and C is a general superclass of C'.

DMS4. If C' is a subclass of C, and I' is an instance of C', then I' is said to be a logical instance of C. When referring to the instances of C, both instances and logical instances are included.

DMS5. If S is the set of attributes and methods of class C, and C' is a subclass of C, then S' is the set of attributes and methods of class C' where either:

(a) S' is a superset of S and the set $N = S' - S$ (where '-' is set difference) represents all of the new attributes and methods defined explicitly in C' which are not inherited from C, or

(b) $S' = S - R + N$, where R is the set of attributes and methods from C which are redefined in C' and N is the set of new and modified attributes and methods defined in C' (where '-' denotes set difference and '+' denotes set union).
DMS6. If DV is a default instance variable of class C with value v, and C' is a subclass of C, then there exists a default instance variable DV' of class C' with value v' such that one of the two conditions holds:

(a) v' = v if C' inherits the default value from C
(b) v' = v'' if C' modifies the value of v' and v'' is the new modified value

DMS7. If AV is an aggregate class variable of C, and C' is a subclass of C, then there exists an aggregate class variable AV' for C' such that AV' reflects the same aggregate property for C' that AV represents for C.

DMS8. If SV is a shared class variable of C with value v, and C' is a subclass of C, then there exists a shared class variable SV' of C' with value v' such that one of the two conditions holds:

(a) v' = v if C' inherits the value
(b) v' = v'' if C' reassigns the new value v''

Name conflict resolution

Although multiple inheritance may be desirable in many situations, it raises the issue of name conflicts. For example, suppose that a class inherits attribute A from each of its superclasses. (see Figure 3.3 ). If attribute A is defined differently in each superclass, which A will the new class inherit?

This name conflict must be resolved in a deterministic manner. One possibility is to force the attribute to be inherited from the class which existed first. This can be determined in various ways, one of which is to use the numerical value of
the object identifier to establish which class was created first. Another method is to utilize the fact that classes keep a list of their superclasses. The class can be forced to inherit from the first class in this list when a name conflict arises, thus eliminating the problem. These two methods are defaults in that the system picks a default superclass for the class to inherit from. It may be desirable to allow a user to alter the default. For example, a user may be able to edit the listing of superclasses to change the order, thus affecting which superclass is used as the default class for inheritance. It is also desirable to allow the user, in certain cases, to pick which superclass the attributes will be inherited from. This can be done by specifying the superclass to inherit from at the time the new class is created. The user should also be able to allow inheritance of instance variables with the same names from more than one superclass by renaming the instance variable from one class. For example, a new subclass can inherit attribute A from two parent classes, renaming one of them A1, so it ends up with attributes A and A1. This can, however, present a problem if an
inherited method from one of the classes needs to reference A (now renamed A1). It will then reference A (from the other parent class), which may not be the appropriate variable to use. One way to deal with this would be to implement a “method minder” which would, upon any changes being made to an attribute in the subclass or any renaming of attributes in the subclass, give a list of methods involving these altered attributes to the user. The user would then have the option of modifying the methods so they would be compatible with the new changes or deleting the methods.

This leads to the following data model specification:

DMS9. Let C1, C2, ..., Cn be superclasses of C'. Let P be the name of an attribute (method) which exists in each of C1, C2, ..., Cn. Then C' inherits P in one of the following ways:

(a) C' inherits P from only one of C1, C2, ..., Cn. The name of the attribute (method) remains the same. In this case, the parent class to inherit from can be specified in one of several ways:

i. Let O1, O2, ..., On be the object identifiers for C1, C2, ..., Cn respectively. Then C' inherits P from the class identified by the min(O1, O2, ..., On).

ii. Let L be a list of superclasses for C'. Then C' inherits P from the first class in the list L containing the attribute P.

iii. When class C' is instantiated, the message to instantiate contains the property/class pair (P, C) specifying, for each attribute and method P the class C it is to be inherited from.
(b) $C'$ inherits $P$ from more than one of $C_1, C_2, \ldots, C_n$ but renames individual $P$'s so that there is no name conflict.

Another name conflict arises, irrespective of whether multiple inheritance is used or not. This conflict arises between a class and its subclass. Suppose that the class has an attribute called $A$ and the subclass also defines an attribute called $A$. This conflict can be resolved by giving precedence to the subclass definition, similar to how a programming language such as Pascal gives precedence to local variables. If the subclass redefines $A$, the same problem discussed above can occur. The method minder again is a possible solution. The following data model specification involves name conflicts in subclasses:

DMS10. If $C$ is a class with attribute (method) $P$ and $C'$ is a subclass of $C$ with attribute (method) $P$, then either:

(a) $C'$ inherits $P$ from $C$, or

(b) $C'$ redefines $P$ locally

Abstracting superclasses

Objects in an object-oriented database have various relationships which have been discussed. If the designer is aware of these relationships during the design phase, many of these relationships can be exploited. For example, STUDENT is related to PERSON in that all students are people, so the commonalities between the two are extracted and placed in the class PERSON. The subclass STUDENT is then created and inherits all the attributes of PERSON.
Often time users find themselves working with objects that are not along the same immediate derivation chain of the class hierarchy, but which, nonetheless have certain commonalities. Therefore, we propose the addition of *superclass generalization* to the data model. This construct allows us to abstract out the commonalities that exist among objects from different classes or simply to create a new superclass for one class.

To motivate the need for this addition to the data model, consider the following example. Suppose we are dealing with a class SHAPE, which has subclasses POLYGON and ELLIPSE. Down the class hierarchy from POLYGON we eventually find class RECTANGLE. Descended from ELLIPSE we have a class CIRCLE. In class RECTANGLE, we find the attributes *height*, *width*, and *position* (representing the shape's position on the screen for example). In class CIRCLE, we have the attributes *radius* and *position*. We may want the option of referring to instances of both classes as a group. If we abstract out the commonalities, we see that a superclass containing the attributes *gheight*, *gwidth*, and *position* describes both classes accurately (a circle has height and width). Therefore, we can create the superclass GSHAPE containing these common attributes. The two classes CIRCLE and RECTANGLE can now inherit from this new class if so desired, or the class can be used as a grouping constructor to allow a user to group objects descended from different branches of the class hierarchy.

Often times, after the design phase, we find we can make modifications to improve our data model. Superclass generalization expands the allowable modifications we can perform. Suppose we have a complicated method $M$ that exists in a class $C_1$. If, in another branch of the tree for class $C_2$, we find we have duplicated the code for
M, we can create a superclass for C1 and C2 containing the code for M, and allow each to inherit the method. The superclass that is added must, of course, be inserted into the class hierarchy (which means associating it via the IS-A relationship to some other class). The position of insertion of this superclass is application dependent and must be determined at the time of insertion, by the user. We now have the following additional data model specification:

DMS11. If C1,C2,...,Cn are classes such that no Ci, 1 ≤ i ≤ n, is a subclass of Cj, 1 ≤ j ≤ n, i ≠ j, then a class C can be inserted as the direct superclass for C1,C2,...,Cn. C must be added to the class hierarchy in such a way that there exists at least one class C' where C is a direct subclass of C'. (Note that a class always exists since we can go to the root class of the hierarchy if need be.) Class C' is a superclass of the classes C1,C2,...,Cn.

Domains of Attributes

It was mentioned previously that an object can belong to only one class. The value of an instance variable is itself an object. It can be a primitive object, or it can be a complex object, but it belongs to only one class. The class it belongs to is called its domain class. If the domain is a primitive class, then the value of the instance variable is a primitive type (e.g. a string or an integer). If the domain of the instance variable is another non-primitive class, then the value of the instance variable is actually a set of values – those values making up the instance variables of the objects in the domain. For example, take the class PERSON mentioned previously. One of the instance variables is address. Suppose address points to an object which is defined to have instance variables street, city, state, and zipcode. The domains of this
object representing the address are string, string, string, and integer, respectively—all primitives.

When the domain of an instance variable is a non-primitive class, all of the subclasses of this domain class are also part of the domain. For example, suppose the address attribute of PERSON was to be of type ADDRESS. If HOME-ADDRESS and BUSINESS-ADDRESS are subclasses of type ADDRESS, then address can be an instance of ADDRESS, HOME-ADDRESS, or BUSINESS-ADDRESS. This is formalized by the following data model specification:

DMS12. If A is an attribute of class C with domain class D, then the domain for A is the subtree rooted at class D. If a class D' is created as a subclass of D, D' becomes part of the domain of A.

When querying the database, one has to think of the domain in a fashion similar to the above. Consider an example where neighbor is an attribute of some object and it is of type PERSON. When querying for a neighbor, the instances of PERSON need to be inspected, as well as the instances of EMPLOYEE and STUDENT; since the domain of PERSON includes STUDENT and EMPLOYEE. Therefore, when querying against a particular class, the query should be directed to that class as well as to all of the subclasses of that class. If a query is directed to a complex object, then the query is recursively employed on the component parts as well. For example, if a query is directed to an instance of type VEHICLE, asking for cost information, the query is also directed at any component parts, such as the engine, the chassis, or the doors.
Aggregation Hierarchy and Composite Objects

Because the domain of an attribute of a class can be another class, this presents us with a new hierarchy, which we call the aggregate class hierarchy. Class PERSON is an aggregate class, with a component class pointed to by the address. When the domain of a class is a non-primitive class, the value for the attribute is actually the object identifier of some instance of some class (or a set of identifiers). Therefore, the value of address is actually the object identifier of some instance of the class ADDRESS. We call this a reference. If one considers all references, we have an aggregate class hierarchy. It is often desirable to consider only a subset of the aggregation hierarchy, namely those objects which exist in the IS-PART-OF hierarchy (which consists of composite or component objects). Note that the class hierarchy (IS-A) has no cycles but the aggregate class hierarchy (class composition hierarchy) may have cycles.

The following example illustrates the concepts introduced. Suppose we have a database consisting of memos from various departments, and we have a subclass CS-MEMO representing the memos from the computer science department. Each instance is a memo sent from this department. Each memo instance references an object called csinfo to obtain appropriate letterhead information. Each instance also has a reference to body (the actual text of the memo). There is a fundamental difference between the csinfo object and the body object – the concept of ownership. Essentially, an instance of CS-MEMO uses the csinfo object to obtain information, but it owns (or is composed of) the body object. Both are references, but the latter reference illustrates what we mean by the IS-PART-OF relationship.

We call the first type simply a reference to an object, and we define the second
type to be a component reference. These component objects can be owned exclusively by one other object (such as the body of a memo belonging to only one memo), or they can be owned by several objects (suppose a particular paragraph object of body is a component object owned by several different memos). We call the first type an exclusive component object reference and the second type a shared component object reference. If an object has an exclusive composite object reference, it is still possible to have other simple references. Consider an example with an object VEHICLE, consisting of several objects, one of which is the component object engine. There is an exclusive component object reference to engine since the engine can be part of only one car. However, there may be another reference to engine, for example, by an object ENG-INVENTORY which includes the engine as part of the inventory. One might argue at this point that engine is actually part of the inventory also. The engine is "related to" the vehicle in a much different manner. The engine is part of VEHICLE, but is merely an element in an aggregate of ENG-INVENTORY. Figure 3.4 shows a component reference and a simple reference for this example. This yields the following DMSs:

DMS13. If O is an object with attribute A, where A is a non-primitive object, then

O references A. All references are either simple or component references. Com-
ponent references augment the IS-PART-OF semantics and are either exclusive or shared.

DMS14. If an object has simple and component references, it is called an aggregate object. Objects related via component references form a composite object. Composite objects then form a subset of aggregate objects.

This introduces the notion of dependency. The idea of dependent and independent objects is introduced in [5, 49]. An object is dependent if its existence depends on the object referencing it. A dependent object cannot be created if its owner does not exist. For example, the door of a vehicle is owned by one specific vehicle. Furthermore, if the composite object (vehicle) is deleted, all of its dependent objects (door) must also be deleted. If an object has both a dependent and an independent reference to it, it is deleted if the object with the dependent reference is deleted, resulting in a dangling reference for the object with the independent reference.

This approach to dependent and independent objects, however, raises some questions. Should the door really be deleted if the car is deleted? Suppose the door exists in an inventory list. The key to answering this question is to first answer when was the door created. Did it exist before the car (as part of the inventory list) or was it created when the car was created? If it existed before the car was created, it should exist if the car is deleted since its existence does not depend on the car.

Consider another example. Suppose we go back to our memo database, and we have created a memo consisting of a heading, a body, and a closing, where the body consists of three paragraphs. By definition, these paragraphs are dependent objects. Now suppose another memo is created that is similar to the first in that the middle paragraph is the same (therefore the two memos share this paragraph). If the first
memo is deleted, the paragraph is deleted (according to the aforementioned rules for dependent objects).

We feel that to define a dependent object, it is important to consider the reason an object is created. Therefore one approach defines a dependent object as an object with an exclusive component reference, which was created when its parent referencing object was created. This means it does not exist until its parent exists and it can only be owned by one object.

Another approach requires that a reference count be kept. Any reference to an object, regardless of the type of reference, increments the count by one. If the reference count is greater than one, the object becomes independent. If the reference count is one, then the object is defined as a dependent object, and its existence depends on the single object that references it. We choose this latter definition, thus giving the following DMS:

DMS15. A referenced object $O$ is either independent or dependent. If the object has more than one reference to it, it is defined to be independent. Otherwise, it is a dependent object. Deleting an object which references an independent object only reduces the number of references to the object by one, whereas deleting an object which references a dependent object results in the deletion of the dependent object as well.

It should be noted that the aggregation hierarchy does not correlate to the class hierarchy (nor does the IS-PART-OF object hierarchy correlate to the class hierarchy). The aggregation hierarchy simply outlines the composition of the objects, specifying the domains for the attributes, and the IS-PART-OF hierarchy reflects the part-of relationship among objects in the database. Certain problems brought
about by the introduction of the aggregation hierarchy must be addressed. One such problem is magnified by the prospect of having a multimedia database. Sometimes, in defining or representing an aggregate object, the component parts need a particular order enforced on them. With our employee object, there was a set of values, and no special order needed to be specified. However, consider the class MEMO, with attributes heading, body, and closing. Suppose body is an aggregate object composed of several objects of type paragraph. For the memo to make sense, a particular order must be imposed on the paragraphs. One solution would be to include an “order” number with each object of type paragraph indicating in which order each is used to become the body of the memo. Another would be to include pointers to create a type of “linked list” to get the proper ordering of objects. For example, heading could point to paragraph1, then paragraph1 could point to paragraph2, etc.

Version Hierarchy

The original core model does not handle composite and aggregate objects, nor does it address the issue of versions. Versions are an important feature in several data-intensive application environments, such as design databases or multimedia databases. Users need the option of examining various versions before selecting the object of choice. For example, in a design database, often the specification for a design object is given, and several versions supporting this specification are created. The specification can change, however, resulting in a new version of the specification object. Then new versions of this specification might be created. We adapt our method of dealing with versions from [5] and [49].

A class can be versionable or non-versionable. Versionable objects are instances
of a versionable class. This can result in a version chain, a version hierarchy, or a version graph. A version chain results when only one version can be derived from an existing version. Thus, there is a linear order imposed on the versions of the object. The other extreme is the version graph. The version graph allows any number of new versions to be derived from any version at any time (the derived-from relationship) and allows a version to be a descendant of more than one older version (the version-of relationship). It should be noted that most proposed models for versions restrict the graph to be a hierarchy. Figure 3.5 illustrates a version hierarchy (VERSION-OF). This yields the following data model specification:

DMS16. A versionable object is the root of a version hierarchy. Descendent objects are all related to the versionable object by the version-of relationship. Newer versions can be derived from older versions.
The version graph represents a history of evolution of the object, and this information can be stored in the *generic object*. If the class is versionable, when an instance of the class is created, a generic object is created along with the instance. This generic object is a data structure for the version-derivation hierarchy, but has a unique object identifier. The generic object and all instances belong to the same class, with the generic object containing a special flag to differentiate it from the actual instances. [49] cites the following as a list of items maintained by the generic object:

1. **Version Count.** This represents the number of existing versions of the versionable object.

2. **Next-version Number.** This is the version number that will be assigned to the next version of the versionable object that is created. It is incremented each time a new version is created.

3. **Default Version Number.** Some references to objects may be *dynamically* bound, that is, determined at run-time. The intent is to allow some default version to be used in this situation. The default version number is the number of the version to be used as this default.

4. **User Default Switch.** This is directly related to the default version number, in that it indicated whether the user has specified the default version or not.

5. **Version-derivation Hierarchy.** This is a tree of version descriptors, one for each version of the versionable object. It includes:
   
   - the version number of the version
• the object identifier of the version

• the children, or list of references to the version descriptors of all versions directly derived from the version

There are three types of versions: transient versions, working versions, and released versions [5, 49]:

1. A transient version has several properties:

   • A transient version can be updated by the user who created it.
   • A transient version can be deleted by the user who created it.
   • A new transient version can be derived from an existing transient version. The existing one is then “promoted” to a working version.

2. A working version has the following properties:

   • A working version is considered stable and cannot be updated.
   • A working version can be deleted by its owner.
   • A transient version can be derived from a working version.
   • A transient version can be “promoted” to a working version. This promotion can be user specified or system determined.
   • A working version can be “demoted” to a transient version by the user.

3. A released version has the following properties:

   • A released version cannot be demoted.
   • A released version cannot be updated.
• A released version cannot be deleted.

• A working version can be “promoted” to a released version by the user.

These three different types of versions represent the history of a version. When it is first created, many modifications may occur and it is not stable. But when the version has reached a reasonably stable state, it can be shared, or it can become a working version. When the object reaches its final state, it becomes a released version and cannot be modified or deleted by itself. Deletion in the version hierarchy must be considered carefully. There are three objects to consider:

• The versionable object

• A non-leaf version object

• A leaf version object

If the versionable object is deleted, then all versions are deleted, and no record is maintained of the objects. If a non-leaf version object is deleted, a record of this object in the history is maintained in order to deal with any versions derived from it. If a leaf version object is deleted, no extra steps are required.

Previously we mentioned the idea of default versions. If a reference is made to a particular version number, the use of a default is not necessary. However, it may be that some object referencing this versionable object does not request a particular version – it only knows that it needs the object itself. Since this is a version hierarchy and not a version chain, using the most recently modified version does not determine a unique default version. Therefore, the user can determine how the default version is to be determined. One method is to use the version with the latest timestamp.
If the user default switch is off, and the user has not specified which version to use in a dynamically bound situation, then the system needs some method of assigning the default version number. Again, the system could simply use the version with the latest timestamp.

**Schema Evolution**

Suppose that the database schema is defined (the class definitions and inheritance have been specified) and several instances of the database have been created. Now suppose a user wishes to modify a class definition. The ability to change the class is called *schema evolution*. There are several changes that could occur, with each change giving rise to several different issues that must be addressed. There are two categories of changes to consider. The first category of changes involves those that change the class definition, such as modifying or adding instance or class variables or even methods. The second type of schema change involves changing the relationships among the classes, such as adding or deleting classes. The following list enumerates the possible changes in each of the categories [5].

1. Changes to the class definition

   a. Add an instance variable or class variable
   b. Delete an instance variable or class variable
   c. Rename or modify an instance variable or class variable
   d. Change the domain of an instance variable or class variable
   e. Reassign a new parent to an inherited instance variable
f. Add a method
g. Delete a method
h. Rename or modify a method
i. Reassign the parent(s) of an inherited method

2. Changes to the relationships among the classes
   a. Drop an existing class
   b. Make an existing class a superclass of some other existing class
c. Remove a class as a superclass of an existing class
d. Make a new class the superclass of an existing class(es)

The taxonomy of schema evolution and certain invariants of schema evolution are presented in [5]. We summarize the properties of the class hierarchy that any changes to the class definitions and to the structure of the class hierarchy must preserve.

**Class Hierarchy Invariant.**

The class hierarchy must remain a rooted and connected directed acyclic graph.

**Distinct Name Invariant.**

All instance variables and methods of a class, whether locally defined or inherited, must have distinct names.

**Distinct Identity Invariant.**

All instance variables and methods of a class have distinct origins.
Full Inheritance Invariant.

A class must inherit all instance variables and methods from each of its superclasses.

Domain Compatibility Invariant.

If an instance variable \( V_2 \) of a class \( C \) is inherited from an instance variable \( V_1 \) of a superclass of \( C \), then the domain of \( V_2 \) must either be the same as that of \( V_1 \) or a subclass of \( V_1 \).

Each of the aforementioned changes results in a potential problem. Take, for example, changing a class by simply adding an instance variable. Care must be taken to name this instance variable with a unique name. Suppose that we add a new attribute and name it \( A \). No other instance variables in the class are named \( A \). But suppose a subclass exists which has an instance variable \( A \). This is easily resolved by giving precedence to the subclass (or most local) definition. What about deleting an instance variable? Any method requiring that instance variable will no longer be useful. Deleting (as well as inserting) an instance variable has a huge impact because it not only affects the class itself, but every descendent of the class. Some method of change notification must be used to propagate the "news" of the deletion of this variable. This could be done by invoking a method any time this type of change is performed that would notify the existing subclasses to any changes being made. If this is desired, the class should contain a list of the direct subclasses in addition to the list of direct superclasses to make the change notification more efficient.

Similar problems arise if instance variables are renamed or given different domains. If we wish to change the parent that a class inherits a particular attribute from, this introduces problems. Different methods may need to be inherited along
with the new attribute, depending on its type. Changing the default value of an instance variable does not cause the same type of problems. However, adding a shared variable or deleting one can result in similar problems to those mentioned above.

Changes to methods also change the class definition. If a method is added, deleted or renamed, again notification of this change must be propagated to all ancestors. Care must be taken when adding a new method to ensure a unique method name. If a method is recoded, since methods are stored in the class and inherited by the subclasses, this schema change is not as hard to handle as the others. If the subclasses truly inherit the code, and the code is stored only in the class, then the changes will be enforced when any subclass invokes the method.

The second type of schema change involves changing the relationships among the classes. Dropping a class involves dropping all instances of the class. It is not necessary to drop all subclasses. The superclass of the class that was dropped can become the superclass of the subclasses. Of course, the inherited methods and attributes will not be the same, thus forcing change notification down the hierarchy.

We now present the following additional data model specifications to enforce the above invariants. (Note that DMS9 and DMS10 partially address these schema modifications.)

DMS17. When either an instance variable, class variable, or method is changed in a class C, the changes must be propagated to all subclasses of C that have not redefined the instance variable or method. If an instance variable or method is added to the class, then the change is propagated to all subclasses as long as no name conflicts are introduced as a result of the modification.
DMS18. If a class $C$ is made a superclass of class $C_1$, then $C$ is added to the list of superclasses of $C_1$ as the last superclass. If class $C$ is newly added to the hierarchy, a list of superclasses must be specified for it also. If not, it becomes a direct subclass of the root of the class hierarchy.

DMS19. If a class $C$ is removed from the hierarchy, and $C$ is the only superclass of $C_1$, then $C_1$ is made a direct subclass of all of the superclasses of $C$.

DMS17 is included to ensure that all instance variables and methods of a class have distinct names. DMS18 forces any class that is added to maintain the class hierarchy structure, namely a rooted and connected directed acyclic graph. DMS18 also enforces the class hierarchy invariant.

Allowing the class definitions to be modified gives rise to the following question – should we be allowed to keep the old definitions and have the new definitions as versions of the originals? If the application warrants doing so, then this is a very reasonable thing to manage. *Schema versioning* allows the schema to be versioned just like any other object might be versioned. Therefore, users have different conceptual schemas of the database depending on which schema version they choose. One conceptual schema may contain an attribute that has been removed from a different schema.

If one user chooses to modify the schema, this change will not be propagated to all users if schema versioning is supported. In a single-version schema environment, if one user drops an attribute, for example, all values for that attribute are forever lost. Even if, at a later time, it is decided that the attribute should not have been deleted and is again added, any values associated with the attribute at the earlier time cannot be recovered.
Summary

Our data model has been presented in this chapter. This model satisfies the requirements for an object-oriented environment and addresses several design issues. First, this model supports the same properties as the semantic data models with the additional power of handling composite objects (the aggregate hierarchy and IS-PART-OF hierarchy) and versions (the VERSION-OF hierarchy). The data model is easily extensible – adding new classes is as easy as sending a message. The notions of generalization and specialization are supported by the class/subclass (IS-A) hierarchy. Sharing is allowed via shared references to objects. Component objects can be ordered if this order is part of the definition of the object they comprise. This data model is also flexible. Objects can be versioned (VERSION-OF hierarchy) and the schema can be modified. Storage is reduced by allowing shared variables and default values.

This data model allows homogeneous as well as heterogeneous sets. It combines the notion of versions and aggregate objects with existing core concepts for a complete approach. Superclass generalization is introduced as a means to allow the user more freedom in design. Value and default instance variables are defined for the model, as well as shared and aggregate class variables. The data model specifications provide a concise and formal set of rules for our object-oriented data model. In the next chapter, we present our security policy for object-oriented databases that can be incorporated into our data model.
CHAPTER 4. THE SECURITY POLICY

Should security in an object-oriented database be inherited? All of the previous approaches to object-oriented database security promote the use of inheritance via the class hierarchy to protect the data efficiently. Inheriting security based on the class structure is sometimes restrictive. A security policy for object-oriented database systems is presented in this chapter. In our model, we concentrate on securing the data values, but also allow the security to be inherited through the structure. Mandatory and discretionary security controls are addressed by this model. We present an alternative approach to security by placing the majority of the security checking within the object itself instead of inheriting security from the class. Classes can derive security classification constraints from their instances and logical instances.

Overview of Existing Approaches

Recall from Chapter 1 that in the object-oriented environment, the issue of security becomes more complex than in a conventional system. Objects encapsulate data and methods, and several different hierarchies exist, thus necessitating additional access types. The richness of the data model complicates security even further.

Previous research [44, 45, 83] done in object-oriented database security was discussed in Chapter 1. Past research emphasizes the inheritance of security from the
class hierarchy structure and the instance hierarchy. The following constraints are placed on classes and objects:

- The security levels of the instances of a class must dominate the security level of the class.
- The security level of a subclass must dominate the security level of the superclass.

An alternative approach is suggested to model security in an object-oriented database. We propose individual security labels on all primitive, or non-aggregate, objects. The security label for a class need not be inherited from its superclass (unless specifically requested by the user) but can be derived from its instances and logical instances. Our policy will enforce mandatory as well as discretionary access controls and information flow.

Thuraisingham [83] introduces the SORION (Secure ORION) model, a mandatory security policy integrated into the ORION object-oriented database system. In addition to including concepts such as polyinstantiation and support for handling the inference problem, her security policy includes rules to assign security constraints and how these rules are to be handled. However, the discussion does not include advanced concepts such as versions.

Objects in the SORION model also have several restrictions. Some of these restrictions are:

- The security level of a set object must dominate the security level of the individual elements of the set.
• The security levels of the instance variables of an object must be the same as that of the object.

• The security levels of the instances of a class must dominate the security level of the class.

• The security level of a class must dominate the security level of the superclass.

In SORION, no class can exist with a security level less than that of its superclass. Also, an object cannot have instance variables with varying security classification labels. In an object-oriented system, many attributes are semantically joined because they are attributes of the same object. For example, we could have a PERSON object with attributes name, SS#, address, and salary. Suppose we want to keep the salary attribute confidential. We would have to make the entire PERSON object confidential in the above model, which may be unnecessarily restrictive.

Keefe [44, 45] presents another mandatory security policy for object-oriented systems with the classification policy again based on inheritance. The SODA (Secure Object-oriented Database system) project develops a multilevel secure object-oriented database management system using an object-oriented data model. It focuses on security issues affecting the application interface and incorporates distributed cooperating-operating objects.

Both of the previous multilevel security policies proposed for object-oriented systems are based on inheriting the security classification labels or ranges. Consider the following example. In an object-oriented system, subclasses are specializations of the classes. They inherit the attributes from parent classes and add additional attributes of their own. Suppose we have a class LIST that represents an unordered
list of elements. Now we create a subclass **ORDERED_LIST** which is a specialization of **LIST**. Imposing the security classification labels on **LIST** and **ORDERED_LIST** by following the rule that the security label of the subclass must dominate that of the class requires that if **LIST** is unclassified then **ORDERED_LIST** must be unclassified, confidential, secret, or top secret. But it is quite possible that one might wish to use a **LIST** for some secret purpose and an **ORDERED_LIST** for an unclassified application. This leads us to the question: Is the security classification of an object in an object-oriented database determined by the class/subclass hierarchy? Or is it, instead, independent of this hierarchy?

Our policy ensures security by attaching a security classification label to all non-aggregate as well as aggregate objects in the database and checking allowable access at the time a request is made for the data. The classification label of an object may be static, not allowing for any changes, or dynamic, allowing for changes or upgrades within an allowable range.

In this chapter we detail the security policy to be incorporated into the data model discussed in the previous chapter. We attach security classification labels to all object instances and classes. Methods accessing these objects are required to check for authorization based on a set of rules. All users have a security clearance which consists of a security clearance level and a compartment. These security clearances, as well as the security classifications, form a lattice. Read access to an object \( O \) by a user \( U \) is allowed only if:

1. The security clearance of the user \( U \) dominates the security classification level of the object \( O \), that is, \( L_U \geq L_O \), and
2. The compartment of the user is contained in the compartment of the object $O$, that is, 
$$C(U) \subseteq C(O).$$

From this point on, we will refer to the security clearance of the user as $(L_{Sclear})$, inferring the clearance and the compartment. The classification label of the object will be denoted as $(L_O)$.

The Security Properties

Following are the security properties for our policy. We provide these properties for objects in general, classes, methods and versions. Each subsection deals with one of these areas.

Security properties for objects

The term "object" in an object-oriented environment refers to essentially everything. We, however, will restrict the term object in this section to instances of classes and attributes of those instances. Because security has been incorporated into the model, one more type of object needs to be introduced. This new object is called a polyinstantiated set in [45]. This is a set with different values existing at different security levels. The security label for each element in the set must be unique. Polyinstantiation has been used to handle the multiparty update problem. When low-level users attempt to update or overwrite higher level data which they cannot see, if the write is allowed, then the data cannot be protected, and if the write is not allowed, a covert channel is introduced. By allowing unique values with varying security classification levels to co-exist in this set, a low-level user can add an element to the
set at the user's level without interfering with the higher level data. Of course, this means several values exist for one object, and the problem now becomes deciding which value is the "correct" one. One possibility is to gain access to the value with the highest allowable security level and accept that as the correct value.

The security properties for objects are:

O1. All non-aggregate objects (simple or primitive objects) must explicitly have a security classification label assigned to them, that is, for all objects \( O \) in the database, \( L_O = L \), where \( L \) is an existing security class.

O2. The classification label assigned to an aggregate object is the greatest lower bound of the labels of the attributes of that object. If \( O_a \) is an aggregate object consisting of \( \{O_1, O_2, \ldots, O_n\} \), where each \( O_i, 1 \leq i \leq n \), can be either a primitive object or an aggregate object, then \( L_{O_a} = \text{glb}(L_{O_1}, L_{O_2}, \ldots, L_{O_n}) \).

If the creator of the aggregate object wishes to explicitly assign a security label to the aggregate object itself, this forces all attributes of the object to be assigned the same security classification label, that is, \( L_{O_a} = L \Rightarrow L_{O_i} = L \), for \( 1 \leq i \leq n \).

O3. If an object is a set object, then the security label of the object is the greatest lower bound of the security classification labels of all the elements of the set. That is, if \( O_s \) is a set object consisting of \( \{O_1, O_2, \ldots, O_n\} \), where each \( O_i, 1 \leq i \leq n \), is either a primitive object or an aggregate object, then \( L_{O_s} = \text{glb}(L_{O_1}, L_{O_2}, \ldots, L_{O_n}) \).

O4. A negative authorization list may be stored in the object by its creator to deny access to this object by other users. (For example, once it has been
determined that the security clearance of the user requesting read access to the data dominates the classification of the data, the negative authorization list in the object is checked. If the user’s identity appears in this list, read access to the data is denied.) More formally, let object $O$ with $L_O = L$ contain negative authorization list $N$, where $N$ consists of a list of (authorization, user identity) pairs. If a user $U$ with user identity $= U_{id}$ and security clearance $L_{Sclear} \geq L_O$ wishes to gain read access to $O$, $N$ is checked. If the pair $(R, U_{id})$ exists in $N$ (where $R$ is read access), $U$ is denied read access to $O$. If the user $U$ wishes to gain write access, and $L_{Sclear} \leq L_O$, then access is denied if the pair $(W, U_{id})$ exists in $N$ (where $W$ is write access).

The greatest lower bound is used to help when querying the database. If a query from an unclassified user is directed to an aggregate object with a classification of secret, this object can be cut from the query view because there are no attributes that this user can see, where the query view is the view representing a subset of the data in the database the user is authorized to access. On the other hand, if the label on the aggregate object is unclassified, there must exist some attributes at the unclassified level. The user is prohibited from seeing the attributes at higher levels when the method is invoked requesting this particular information (we discuss this in the section on methods).

The idea of the negative authorization list is quite straightforward. When creating an object, if the creator wishes to further limit access to this object (in addition to clearances and classifications), entries are put into the negative authorization list of that particular object. This list contains the identity of users and types of accesses not permitted. For example, a user could be placed on this negative authorization list.
for one type of access, but not for another. These authorization lists are somewhat similar to a distributed access control matrix. This allows us to incorporate discretionary as well as mandatory access controls in our security policy. Mandatory access is controlled by the security clearance of the user and the classification label assigned to the object in the database. All accesses must be authorized. Furthermore, access may be additionally limited. Even if a user is authorized to see an object based on the clearance/classification comparison, discretionary controls may disallow the access based on a need-to-know principle. This is done using the negative authorization lists.

Security properties for classes

Classes serve two purposes: they act as templates so that instances can inherit attributes and methods and they act as grouping constructors to collect similar instances. Classes also contain class variables of two kinds: aggregate and shared. The security properties for classes are:

C1. If a class is not given an explicit security label when created, the security label of the class is the greatest lower bound of all instances and logical instances of the class. That is, if, for class $C$, $L_C$ is not assigned at the time of creation, then $L_C = \text{glb}(L_{I_1}, L_{I_2}, \ldots, L_{I_n})$, where each $I_i, 1 \leq i \leq n$, is an instance or logical instance of $C$.

C2. If so desired, the user can assign a security label to a class at the time of creation. All instances of this class must then have the same security label. The security levels of all subclasses must also be explicit and must dominate that of the class. More formally, if, for a class $C$, $L_C$ is explicitly defined to be
equal to $L$, then for all $I_i$ an instance of $C$, $L_{I_i} = L$, and for all subclasses $C_i$ of $C$, $L_{C_i}$ must be explicit such that $L_{C_i} \geq L_C$.

C3. Any aggregate class variable that might exist in the class will have as its security classification label the least upper bound of the security classification associated with any instance that contributes to its value. That is, if $V_a$ is an aggregate class variable of class $C$, then $L_{V_a} = lub(L_{O_1}, L_{O_2}, \ldots, L_{O_n})$, where each $O_i, 1 \leq i \leq n$ contributes to the value of $V_a$.

C4. Any shared class variable must have its security classification label determined a priori (at the time of creation of the class). Since this is a variable to be shared by all instances of the class, regardless of the instance’s security classification, a label must be assigned at the time the shared variable is created. If a user attempts to access this shared variable and does not have the proper authorization, access will be denied. That is, if $V_s$ is a shared class variable such that $L_{V_s} = L$, and $U$ is a user such that $ZU = Z_1$, and $L_1$ does not dominate $L$, i.e. $(L_1 \leq L)$, then $U$ will be denied read access to $V_s$.

C5. A negative authorization list may be stored in the class by its creator to deny access to this class and its variables by other users. Let class $C$ with $L_C = L$ contain negative authorization list $N$, where $N$ consists of a list of (authorization, user identity) pairs. If a user $U$ with user identity $= U_id$ and security clearance $L_{Sclear} \geq L_C$ wishes to gain read access to $C$, $N$ is checked. If the pair $(R, U_id)$ exists in $N$, $U$ is denied read access to $C$. If the user $U$ wishes to gain write access, and $L_{Sclear} \leq L_C$, then access is denied if the pair $(W, U_id)$ exists in $N$. 
Property C3 requires that the least upper bound of all contributors to an aggregate class variable be assigned to the class variable. This is because of inference problems with aggregate variables. Information flows from each of the contributors to the aggregate variable. Therefore, the least upper bound is used to ensure that information cannot flow to an unauthorized user. Property C5 allows discretionary access controls to be applied to classes as well as objects (from Property O4).

We note that our security properties address a superset of previous research on database security. Recall that former work advocates inheriting the security from class to subclass and class to instance. Because we use the greatest lower bound of all logical instances to determine the security level of a class, the class security level will certainly be dominated by that of any subclasses or instances. The two properties listed earlier dealing with class security levels and inheritance can be obtained by using our policy. Our approach to security allows more freedom in the database design. Recall the example earlier involving the list and the sorted list. Previous research demands a security level to be attached to the class before instances are created, thus limiting the possible instances that can belong to the class. Our policy, however, derives the security level from the instances themselves.

Security properties for methods

Next we address the security properties dealing with methods. Since access to any object in the database must be via message passing that invokes a method, secure execution of all methods is of extreme importance. Methods enforce the security policy that we propose in this section. A method executes within a range of security levels determined by the current security classification level of the method which
invoked it ($L_{\text{caller}}$) and the clearance of the user for whom the methods have been activated ($L_{\text{clear}}$). Method activations must satisfy the following properties: [45]:

**M1.** The user's login method begins execution at the system's lowest level, or

$$L_{\text{current}} = \text{UNCLASSIFIED}.$$  

**M2.** A method may read the value of a labeled object $O$ with sensitivity level $L_O$ if $L_O \leq L_{\text{clear}}$. An unreadable object returns $nil$.

**M3.** A method may create and store a labeled object with sensitivity level $L_O = L_{\text{current}}$. If the object is constrained to have the security range $[L_{\text{bottom}}, L_{\text{top}}]$, then the object can be stored if $L_{\text{bottom}} \leq L_{\text{clear}}$ and $L_{\text{current}} \leq L_{\text{top}}$. Otherwise the update is rejected.

**M4.** As a method executes, its current classification label changes dynamically. A method starts initially with $L_{\text{current}} = L_{\text{caller}}$. If the method accesses an object $O$ such that $L_{\text{current}} < L_O \leq L_{\text{clear}}$, the current classification level of the method will be assigned that of the object, or $L_{\text{current}} = L_O$. The classification level changes dynamically to reflect the classification of objects the method has had direct or indirect (through other method activations) access to.

**M5.** A method's security classification label can never exceed that of the clearance of the user activating the method (either directly or indirectly, via another method). That is $L_{\text{current}} \leq L_{\text{clear}}$ for every method invocation at all times.
M6. An object $O$ returned upon method completion is labeled with the classification of the method at the time the object is returned, or $L_O = L_{\text{Scurrent}}$.

To illustrate these properties, consider the following example. Suppose a method $M1$ begins execution on behalf of a secret user. $M1$, having accessed no data in the database at this point, is executing at the system low, or unclassified, level. If $M1$ needs to read a confidential piece of information, the read is allowed, and the method's execution changes to confidential. If $M1$ attempts to read a top secret piece of information, the read is not executed and $\text{nil}$ is returned.

Suppose now that $M1$ calls another method $M2$. $M2$ begins operating at the confidential level. $M2$ now attempts to write to an object constrained to the range [secret, top secret]. The property $L_{\text{bottom}} \leq L_{\text{Sclear}}$ (secret $\leq$ secret) ensures the integrity of the data item by allowing only authorized users, and the condition $L_{\text{Scurrent}} \leq L_{\text{top}}$ (confidential $\leq$ top secret) ensures that the object does not end up with a value beyond the specified range. So the write by $M2$ is allowed. If $M2$ now reads a secret piece of information, the read is allowed. Now suppose $M2$ terminates and passes a value to $M1$. The value passed is at the secret level, and $M1$ now operates at a secret level. If this value is now returned to the user, it is labeled secret.

The multilevel-user update problem

Recall that polyinstantiation has been used to solve the multilevel-user update problem. If several users with varying clearances are updating the database, polyinstantiation allows each user to update without interfering with other updates already performed by other users with different security clearances. Additional specifications
must be provided if polyinstantiation is implemented in the database. Each logical object might be represented by several different physical objects, differing only in the security classification level that is assigned. Two new properties must be added to the properties specified for methods. These properties correspond to M2 and M3 as follows:

M2'. (for polyinstantiated sets) A method may read the values of all labeled objects in the polyinstantiated set with sensitivity level $L_O$ where $L_O \leq L_{\text{clear}}$. $\text{Nil}$ is returned when no objects are readable.

M3'. (for polyinstantiated sets) A method may create and add a labeled object to a polyinstantiated set with $L_O = L_{\text{current}}$. If the object is constrained to have the security range $[L_{\text{bottom}}, L_{\text{top}}]$, then the object can be stored if $L_{\text{bottom}} \leq L_{\text{clear}}$ and $L_{\text{current}} \leq L_{\text{top}}$. Otherwise the update is rejected.

If a method $M1$ reads from a polyinstantiated set, any values in the set with a classification of secret or below are returned to the user. If $M1$ writes to the set, the update can be performed if the user’s clearance falls within a specified range for the object. If the range constraint is satisfied or if no range is specified, the update is performed. This is accomplished by either updating an existing value at that level or by adding a new one, if none exist at that level.

Security properties for versions

Recall that objects in the version hierarchy are related by one of two different ways: One object can be a version of another, and one version can be derived from
another. The versionable object has several objects related to it by the version-of relationship. Several of these versions may be derived from existing versions, with minor or major modifications. We propose the following security properties for versionable objects and their versions:

V1. Every versionable object and any existing version must have a security classification level assigned to it, or for all objects $O_v$ in the database, such that $O_v$ is a versionable object or a version of an object, $L_{O_v} = L$, where $L$ is an existing security class.

V2. The security classification of the versionable object is assigned the greatest lower bound of all version instances. If $O_v$ is a versionable object with versions $O_1, O_2, \ldots, O_n$, where each $O_i, 1 \leq i \leq n$ is related to $O_v$ by the "version-of" relationship, then $L_{O_v} = \text{glb}(L_{O_1}, L_{O_2}, \ldots, L_{O_n})$.

V3. If a version $O_i$ of a versionable object $O$ is related to versions $O_m, \ldots, O_n$ by the "derived-from" relationship, then the security classification level of $O_i$ must dominate all of those for $O_m, \ldots, O_n$. That is, $L_{O_i} \geq \text{lub}(L_{O_m}, \ldots, L_{O_n})$.

The greatest lower bound is again used to help when querying the database. If a query from an unclassified user is directed to a versionable object with a classification of secret, this object can be cut from the query view because there are no versions that this user can see. On the other hand, if the label on the versionable object is unclassified, there must exist at least one version at the unclassified level. The user is prohibited from seeing the versions at higher levels when the method requesting this particular information is invoked.
The four hierarchies and security properties

We introduced four hierarchies in the previous chapter, namely:

- Class/subclass Hierarchy (IS-A)
- Class/instance Hierarchy (IS-AN-INSTANCE-OF)
- Composite Object Hierarchy (IS-PART-OF)
- Version Hierarchy (VERSION-OF)

The above properties for objects, classes, methods and versions encompass all necessary constraints for these four hierarchies. The relationship between a class and its subclass is covered by the properties that classes and objects must adhere to. This is true for the class-instance hierarchy also. Composite objects need not be treated any differently than other objects. When a component object \( O_c \) becomes part of a composite object \( O \), \( O \) has a pointer to \( O_c \). The security level assigned to the attribute in \( O \) corresponding to \( O_c \) is the same as the security level of \( O_c \). From a security standpoint, an attribute is an attribute – it does not matter whether the attribute is a primitive object or a complex object, as long as there is a security level associated with the attribute to enforce access. Therefore, the security properties for objects address the necessary issues dealing with composite objects as well. We have elaborated on separate security properties for versions, although the similarity between the security properties for versions and those for objects can be seen (since versions are objects). We now integrate our security policy with the data model.
Integrating the Security Policy with the Data Model

Security properties for objects, classes and methods have been introduced in the preceding section. Our data model has been presented in Chapter 3. This section integrates the security properties into the data model to develop a formal and precise model of object-oriented database security.

Information flow within the data model

Before we can formalize our security policy, we must first discuss the concept of information flow within our model. After we understand exactly how information flows can occur, we will proceed with the presentation of the security policy.

Since we are working within an object-oriented system which supports encapsulation, information flow is rather easy to define. There are basically five types of information flow that we must address:

1. Flow from class to subclass upon subclass creation
2. Flow from class to instance upon object instantiation
3. Flow from object to object via message passing and returned values
4. Flow from instance to version instance upon version instance creation
5. Flow from instance to class in the case of aggregate class variables

Flows of types (1), (2) and (4) can be collectively termed "new-object" flow (as the result of a create message to define a new object). Information flow of the first three types is discussed in [42]. Forward flow is information carried through the
parameters when a message is sent to an object. **Backward** flow occurs when a value is returned after a method activation. **Transitive** flow occurs when there is a flow from one object $O_1$ to another object $O_2$ via a third object $O_3$. In other words, information flows from one object into another and then passes from the latter to a third object. These are all examples of **direct** information flow. If information passes from one object to a third object by passing **through** a second object (but without changing the state of the second object) it is termed **indirect** information flow.

Flows of type (5) occur when classes have aggregate class variables whose values depend on the instances of the class. For example, if a class variable represents the average of the salaries of all instances, then information flows from each instance to the aggregate class variable to determine the value. In other words, the aggregate class variable must know each of the values contained in the instance in order to determine the value of the aggregate it is to store. We term this **aggregation** flow.

Flows from components of aggregate and composite objects can be termed flow from object to object, and are addressed by flows of type (3). Object-to-object information flow can be the result of forward flow, backward flow, or transitive flow, all examples of direct flow. However, indirect flow can also occur between component objects if an intermediate object passes information from one component object to another (or from the root to a component object and vice versa).

We now discuss the different types of information flow prevalent in the four hierarchies we incorporated into our data model. Information flow from a class to a subclass along the class/subclass hierarchy has been termed **inheritance** flow [42]. When a subclass is created, information about the structure and behavior of objects in the class is passed from the class to the subclass. Since any class has read access
to any of its ancestors, there is an information flow along the hierarchy from the root class on downwards to the subclasses. However, this is information flow regarding the *structure* of the hierarchy. If it is necessary to protect the structure and behavior of the class (similar to protecting the attributes in a relation in a conventional database), then it is necessary to assign a security classification label to every class as it is created and to require the security classification of the subclass to dominate the security classification of the class. However, in most cases it is the data and the values themselves that must be protected. For this reason, it is not necessary to assign the classification of a class at its creation. The user is allowed to assign a security classification level to the class if so desired. The security constraints imposed on classes are enforced, thus leaving the option open to protect the structure and behavior.

The class-instance hierarchy represents a similar situation to that of the class/subclass hierarchy. In [42], the authors discuss this *class-instance* flow and make note of the fact that all of the instance objects have implicit read access of the structure and behavior information of the class object. Again, if it is the data and values that must be protected instead of the structure, this flow is not significant. However, if the structure must be protected, then a constraint can be imposed on the database requiring the security classification level of any instance of a class to dominate the level of the class. However, information not only flows from the class to the instances, but also in reverse. When class aggregate objects exist, information flows from the instances to the class variable. Therefore, the classification of the aggregate class variable must dominate all of the classification levels of the corresponding attributes.
in the instances. That is why we assign the classification level of the least upper bound of all instance classifications to the aggregate class variable.

When dealing with aggregate objects, information flow must also be monitored. When one object is a part of another, then the latter has implicit read access to the former. However, access to any information must still be via messages, so this information flow is an example of the object to object flow via message passing. If a new object is instantiated, it represents an example of information flow using the method which creates new objects.

We must also concern ourselves with the version hierarchy. When one version is derived from another, the new version shares certain parts and must have read access to the older version to derive this information. Therefore, information flows from the older version to the newer one. Within the version hierarchy, therefore, we require that the security classification of the newer version must dominate the older one.

The security policy

Using the security properties for objects, classes, methods, and versions that were presented earlier in this chapter, we propose the following security policy:

S1. Users and objects are assigned appropriate security clearances and classifications.

S2. Users have access to objects only by invoking methods. Methods invoked on behalf of a user begin execution at system low (unclassified) and dynamically change to reflect the security classification of any object they have accessed, either directly or indirectly. The security classification of the method is bounded above by the security clearance of the user originating the invocation.
S3. When a method is invoked, the method is passed several parameters. Included in this list are the security clearance of the user originating the method invocation and the user identity. Before the method executes, two things must be checked:

(a) The user's security clearance is compared to the security classification of the attribute of the object the user wishes to access. According to the security properties for methods, if there is a read or write security violation, the method execution is terminated.

(b) If a successful comparison is performed in the previous step, the negative authorization lists are checked. When a user invokes a method to access a particular object, access begins at the root of the class hierarchy and continues downward until the object is found. If, anywhere along this path, a negative authorization list with the user's identity is encountered for that particular method activation, the method execution is terminated. If the path is free all the way to the object, the final step is to check the authorization list of the object itself. If the user's identity is not in that list, access is allowed.

An example

Figure 4.1 illustrates a database schema. PERSON is a class, with no explicit security classification. The attributes of PERSON include name, SS#, and address (an aggregate object). PERSON has two subclasses: EMPLOYEE and CUSTOMER. EMPLOYEE has additional attributes salary and manager (which is of type EMPLOYEE). CUSTOMER also has an additional attribute agent. EMPLOYEE has
subclasses APOLLO-PROJECT, SATURN-PROJECT, and MISC. Class MISC has subclasses CLERICAL and R&D. The R&D class has an attribute proposal, which is a set of PROPOSALS. This is a set because different values exist at different security levels. This is an example of a polyinstantiated set. The security label for each element in the set is unique. A query against this set will result in the element with the highest security classification possible being returned. The clearance of the user must dominate the classification of the element which is returned. Thus, if the user's clearance is \( L_{\text{clear}} \) and the set consists of elements \( \{E_1, E_2, \ldots, E_n\} \), then \( E_i, 1 \leq i \leq n \), is returned with the highest classification \( L_E \) such that \( L_E \leq L_{\text{clear}} \) (provided the discretionary access allows for this). Elements are added to this set if no element exists at that level, and elements replace or overwrite elements in the set if an element already exists at that level. An explicit security label of Secret has been placed on the class APOLLO-PROJECT.

Suppose that these classes are now instantiated. SATURN-PROJECT has all top secret instances (which, according to our property for derivation of the class security level assigns the greatest lower bound, or top secret, to the class SATURN-PROJECT itself). Class APOLLO-PROJECT can only have secret instances since the restriction was placed on the class instead of deriving the level from the instances. Class CLERICAL has unclassified and confidential instances and R&D has unclassified and secret instances. R&D contains a negative authorization for user John Doe to access any names in the class R&D, while one instance of CLERICAL (at the unclassified level) contains a negative authorization for John Doe to access its name attribute.
Figure 4.1: An example to illustrate our security policy
If user John Doe (with an unclassified clearance level) poses a query to the database to retrieve all names of employees, the query is evaluated as follows:

1. The query is directed to the EMPLOYEE class.

2. EMPLOYEE class sends the query to APOLLO-PROJECT, SATURN-PROJECT, and MISC, which, in turn, forwards the query to CLERICAL and R&D.

3. When the query arrives at APOLLO-PROJECT, the class security level is secret, which implies this user cannot access any of the instances of this class.

4. The query directed to SATURN-PROJECT sees the class level (the greatest lower bound of all instances) of top secret and knows there exist no instances this user might possibly access.

5. The query directed to CLERICAL sees a class label of unclassified and then queries each individual instance. The instances with confidential labels cannot be accessed by user John Doe. This narrows the query space to the unclassified instances. One instance has negative authorization, so the query on this part of the database returns all instances at the unclassified level except this one object with negative authorization.

6. The query against R&D sees John Doe has an acceptable clearance, but terminates the query execution when John Doe’s name is found in the negative authorization list as not being allowed to read the names. None of the instances are checked, since John Doe is restricted from the entire class.
Implementing the security policy

The idea of a message filter is presented in [42]. Essentially, the message filter is a system element responsible for intercepting all messages passed in the system and implementing the security policies. It is the message filter that determines whether or not a message should be sent to its intended destination. We apply this same idea, suggesting that the message filter can be implemented at each object (as in a virtual message filter). Therefore, no message is denied arrival at an object. Only when the appropriate method is invoked will any flow violation be detected and proper steps taken to thwart this security problem.

We now present a Method Invocation Algorithm. This algorithm must be executed by all objects when messages arrive. It is unlike the message filtering algorithm in [42] because it is not an algorithm that filters messages. Instead, all messages are delivered to intended objects, and each object must determine whether to invoke the method.

The METHOD INVOCATION ALGORITHM

Before presenting the algorithm, we review notation used previously and introduce some new notation:

- \( L_O \) is the security classification level of the object \( O \)
- \( L_{\text{current}} \) is the security classification level of the current method execution
- \( L_{\text{caller}} \) is the security classification level of the method making the current call (to invoke a method)
- \( L_{\text{clear}} \) is the security clearance level of the user who originated the messages
• $L_C$ is the security classification level of the class

• $U_{Id}$ is the user identification

• $M_A$ is the name of the method to activate

• $P$ is the formal parameter list, if any, in the message

• $V$ is the value, if any, that is returned to the calling method

• $N_C$ is the negative authorization list present at the class the object belongs to

• $N_O$ is the negative authorization list present at the object receiving the message

Messages are 6-tuples with the following format:

$$\text{Message} = (M_A, P, V, U_{Id}, L_{S\text{clear}}, L_{S\text{current}})$$

The algorithm is as follows:

\begin{verbatim}
CASE $M_A$ of

READ : If $L_{S\text{clear}} \geq L_C$ AND

$U_{Id}$ is NOT on the list $N_C$ AND

$L_{S\text{clear}} \geq L_O$ AND

$U_ID$ is NOT on the list $N_O$ then

Invoke $M_A$ (a READ operation)

$L_{S\text{current}} \leftarrow L_O$

Else

$V \leftarrow \text{nil}$

abort $M_A$

\end{verbatim}
WRITE: If \( U_{Id} \) is on the list \( N_C \) OR
\[
(\text{If } L_{bottom} \leq L_O \leq L_{top} \text{ (object has a range)) AND} \\
((L_{Sclear} < L_{bottom}) \text{ OR } (L_{top} < L_{Scurrent}))) \text{ then}
\]
\[ V \leftarrow \text{nil}\]
abort \( M_A \)
Else
Invoke \( M_A \) (a WRITE operation)
\[ L_O \leftarrow L_{Scurrent}\]

OTHERWISE: If \( U_{Id} \) is on the list \( N_C \) then
\[ V \leftarrow \text{nil}\]
abort \( M_A \)
Else
Invoke \( M_A \)

We must note several things. First, when writing an object, if the object's classification level is constrained to be in a specific range, then if the method trying to write this object has a security level out of this range, the operation will be aborted. This is handled in the second and third line dealing with the WRITE case in the algorithm. If the user's identification is not on the class negative authorization list and the object's classification level is either not restricted to a certain range or the restrictions are met, then the write will go ahead and take place. The last thing we note about this algorithm is that it is somewhat recursive. If the method is not a read or a write, then it will be invoked. When it is invoked and makes calls to other methods, this algorithm must be followed for each call to a method.
Eventually, methods read or write values. At this primitive level, the reads and writes are detected by the algorithm and the proper steps are taken to ensure secure operation.

Even though we think of sending messages to instance objects, the messages in reality are passed to the class, since the class contains the actual code for method activations. (This is somewhat of an implementation issue, but object-oriented systems make use of this inheritance and do not repeat the method code for each object.) To implement this algorithm, the code for the algorithm could be attached to each class during class instantiation by the underlying kernel or by some trusted process responsible for class creation. This Trusted Security Stub would be unknown to the user.

If all objects follow the security properties specified in the previous section and adhere to the method invocation algorithm, this leads to the following secure information flow (SIF) theorem:

**Theorem 4.1 (SIF Theorem):** Given an object-oriented database with class objects, instance objects and methods, if all entities in the database adhere to the security policy for object-oriented databases and follow the class security properties, the object security properties, and the method security properties respectively (as outlined in the method invocation algorithm), then no information flow violation can occur.

**Proof:** We consider all flows *into* an object and all flows *out* of an object and prove that, based on the existence of only these two types of flows, no illegal flows can occur within our model.
Case 1. Flows into an object

1.1 Parameters within a message flow into an object (direct forward flow)

Because the method sending the message must adhere to the method security properties, if it is sending parameters, the current method activation level \( L_{\text{Current}} \) must dominate the security level of all parameters in the message being sent. If the parameters are at a higher classification than the object receiving the message, the first conclusion might be that this is a security violation. The only way a violation can occur is if the object retains the value (i.e., stores it somewhere) and allows an unauthorized user access at a later time. This implies a write operation is performed. If a write operation is attempted, one of two things will occur. If the security classification level of the attribute being changed can be upgraded to \( L_{\text{Current}} \), the upgrade is done and no violation has occurred. Anyone requesting this information must now have a security clearance level which dominates the upgraded level. If the level of the attribute being changed cannot be upgraded, the method will abort, and, again, no violation has occurred.

1.2 An object is returned in response to a message (direct backward flow)

Recall that any object returned upon method completion is labeled with the classification of the method at the time the object is returned, or \( L_O = L_{\text{Current}} \). If an object is returned in response to a message, then to ensure no flow violation occurs, two conditions must hold:

i. \( L_O \leq L_{\text{clear}} \) (to check for flow to the user), and
ii. If the object invoking the method to return the value also stores the value, then either the method performing the write assigns the security level of the returned object to any attributes/objects being written or the write is unsuccessful.

The method invoked to read and return the value must abide by the security properties for methods which state that the method can only access the desired value (object) if $L_O \leq L_{S\text{clear}}$. Therefore, condition (1) is satisfied. If the invoking method also stores the value, it must follow the security properties for methods which state that the security level of the object being written is set equal to that of the method, or $L_O = L_{S\text{current}}$. If the security level cannot be modified to the new level, the update is rejected. Therefore, there is no way for information to flow illegally into the invoking object.

### 1.3 Indirect Flow between Objects

Methods which are invoked on behalf of a user can be pictured as a tree of method invocations. The user’s login method constitutes the root node. Messages sent by this method invoke other methods, which make up the next level of the tree. Each node represents a method, while each node’s children represent the methods it invokes by passing messages to various objects. This method invocation tree must satisfy the following property: The security classification level of a child method node must dominate that of its parent node. That is, as we trace from the root to the leaf nodes, security classification levels must never decrease. All method invocations assume the classification level of the calling method that invoked
them. For illegal indirect information flow to occur, information must pass through one object and into another, where the information at the latter is improperly labeled (that is, the method invoked at the latter object is invoked at the incorrect security level). Following the security properties imposed on methods, this can never occur. Therefore, indirect illegal flows are prevented by our policy.

Case 2. Flows out of an object

The only way an illegal flow out of an object can occur is when a value is “released” with an improper security classification level. This can never happen since all values must be retrieved by a method and every method must dynamically change its security classification level to reflect the level of any values it has had direct or indirect access to, limited by the clearance of the user. Therefore, any values flowing out of an object must be properly labeled and, combined with the argument above for flow into an object, no flow violation can occur.

Since all illegal flows into and out of an object are contained by our policy, no illegal flows can occur. \( QED \)

Essentially what we have is a system with dynamically bound security levels on objects. Therefore, information flow from one object to another is largely allowed (unless the security range constraint on the object receiving the flow will not allow the security classification to upgrade to the necessary level). It is the flow from the database objects to the user that is restricted, since the user’s clearance must be static to provide the necessary security.
The above argument does not cover the possibility of allowing polyinstantiation. If objects can be polyinstantiated, the proof is actually more straightforward. Essentially, polyinstantiation allows every write to be performed (there is no security range on the object being created or modified). Flow to the user is checked in exactly the same way as if polyinstantiation was not being used. Note, however, that using polyinstantiation does not ensure the user has the "correct" or most current value for the value requested.

Summary

We have presented an alternative model for security in object-oriented database systems. Instead of forcing a class to inherit its security classification label from its superclass, we allow it to derive this label from its instances. The class derives its security classification label by using the greatest lower bound of its instances and logical instances. This approach encompasses previous work done in this area, and also allows more freedom in class/subclass design and object instantiation.

Security properties for classes, objects, methods, and versions are introduced and integrated into the data model from Chapter 3. The security policy is presented along with a method invocation algorithm that we prove to be correct. Mandatory security is enforced when methods are invoked by checking the security clearance of the user and comparing this to the classification label of the object or objects the user wishes to access. Discretionary access is also addressed by allowing objects and classes to contain negative authorization lists. These lists are checked when a method is invoked on behalf of a user. The user's identity and clearance must be passed to the method upon invocation. The security classification level of a method can change
dynamically, to reflect the level of information the method has had access to, either directly or indirectly via another method. This material has been addressed in [66].

We have presented a security model to protect the stored information. Derived security labels for classes are mainly used for query processing to limit the domain of the query graph, a topic which we discuss in Chapter 6.
CHAPTER 5. OBJECT-ORIENTED TRANSACTION MANAGEMENT AND PROCESSING

Introduction to Transactions and Concurrency Control

A transaction is a sequence of operations (or the execution of a program) performed against a database. Related activities on certain data elements are grouped together to form a logical unit of consistency and recovery. In conventional database systems, a transaction consists of a series of reads and writes executed on the database. A transaction containing only read operations is called a query, while a transaction containing any write operations is called an update transaction. If a transaction finishes normally, we say that it is committed. Until a transaction is committed, it can be aborted, either receiving the abort request from the transaction control mechanism or issuing the abort itself if it cannot complete correctly.

It is important that a transaction appears to execute without intervening steps of other transactions. However, it is also important to allow as much concurrency as possible to increase performance. Therefore, there are certain properties that transactions must satisfy. The user can assume that each transaction executes atomically, as if no other transaction executes concurrently. This property of atomicity allows the sequence of reads and writes for a transaction to be viewed as one operation. Any values from the intermediate steps are not visible from outside the transaction.
Transactions also ensure that the database goes from one consistent state to another consistent state. This consistency property does not require that the database be in a consistent state during transaction execution. The persistence or durability property states that if a transaction succeeds, all writes are recorded and that any committed changes made by a transaction will never be lost, even if the system crashes. If a transaction fails, all writes from that transaction are aborted.

Another important concept of transactions is that of serializability. This simply means that the effect of executing more than one transaction concurrently is equivalent to executing this same group of transactions one at a time, in some serial order. Concurrent execution allows an increase in throughput and performance. However, it may also lead to violations of existing consistency constraints. Several mechanisms have been proposed for concurrency control which are based on serializability. One way of guaranteeing serializability is to use locks [36, 27]. Another method is to use timestamps [73, 81, 69].

**Nested Transactions**

Atomic transactions have been used in conventional systems to guarantee reliable concurrent processing during read and write accesses to data. *Nested transactions* are an extension to the notion of traditional transactions which apply transaction synchronization and recovery mechanisms within transactions. These transactions can have child transactions nested within them. This allows the top level transaction which contains complex operations to be decomposed into more primitive operations. The parent transactions coordinate and supervise the children. All manipulation of data objects is done via the children. (Only leaf nodes can perform the actual
primitive operations on the data elements.) The children are required to start after the parent and terminate before it. Traditional transactions are thus expanded to a tree of transactions, with the root transaction at the top with all of the children of the root synchronized among themselves. Database consistency is assured by requiring serializability at every level in this nested hierarchy. Safe concurrency within transactions is guaranteed (which is especially useful in a distributed system). Nested transactions also provide a finer grained recovery [63]. Nested transactions provide a mechanism for simple and safe composition of transaction routines that can execute concurrently, thus increasing modularity. In addition, the success or failure of a subtransaction is independent of the success or failure of any siblings. If a nested transaction does fail, the parent need only retry that child. If other child transactions have committed successfully and recovery is necessary, checkpointing at each child transaction is finer grained than if the entire parent transaction had to be rerun.

Svobodova [79] has implemented a system based on [69] which uses multiple versions of data and timestamps to implement nested transactions which can tolerate failures of subtransactions. Moss [63] has taken a different approach based on two-phase locking. Because locks are easier to implement and most database systems currently in use today employ locks instead of timestamps, we base our secure transaction processing on locks and the method introduced by Moss.

In the locking method, a lock is associated with certain data items. Each transaction, before performing a read or write operation on a data item, must lock the item first. The transaction must release any locks it holds before it completes. A transaction locks the data element to ensure that the element is inaccessible to other conflicting transactions while the transaction holding the lock is accessing the data
element. There are essentially two types of locks, read locks and write locks. If two locks can be applied to the same data element concurrently, they are said to be compatible. For example, two or more transactions can simultaneously read a data element with no adverse effects. On the other hand, if one transaction is reading a data element, another transaction cannot be updating the same element, so a read lock and a write lock are not compatible. Sometimes locks can be converted from one type to another. If a transaction with a lock on a data item requests a lock on the same data item, but in a new mode, then the lock can be converted. Conversion is possible if the original lock can be replaced by the new lock without violating any compatibility.

Moss [63] presents new locking and state restoration algorithms for nested transactions. Following are the read-write locking rules for nested transactions:

1. A transaction may read an object iff it holds the corresponding lock in either read or write mode.

2. A transaction may write (modify) an object iff it holds the corresponding lock in write mode.

3. A transaction may hold a lock in read mode iff all holders of the lock in write mode are ancestors of the requesting transaction.

4. A transaction may hold a lock in write mode iff all holders of the lock in any mode are ancestors of the requesting transaction.

5. When a transaction commits, its parent (if any) holds the locks of the committing transaction. If the parent already holds the lock, it will begin to hold the
lock in the mode which is more exclusive of the modes of the parent and the child. When a transaction aborts, its locks are discarded after unwinding the transaction’s effects.

This last item, *lock inheritance*, ensures that the parent transactions keep the effects of the child transactions “hidden” until the root transaction commits.

Locking has certain advantages and disadvantages. The main advantage of using locking is that the serialization order need not be assumed a priori, but instead depends on the progress of the transaction and its execution. The main disadvantage of locking is the possibility of deadlock.

Several efficient algorithms exist for hierarchical locking, which relates very closely to object-oriented systems because of the inherent hierarchical nature of the organization of objects. Hierarchical locking is addressed in the following section.

**Locking Policy for Object-Oriented Databases**

Object-oriented database systems offer more of a challenge when dealing with concurrency control. The unit of access is an object; hence, it should also be the unit of concurrency. In a relational database, the relation is the unit of access. These conventional systems do not lock at the tuple level. An object in an object-oriented system has a unique identifier and can therefore be accessed individually. Since objects belong to classes, the locking protocol for object-oriented database systems involving classes must be richer (more complex) than those protocols existing for relations. The classes hold not only the template information for instances, but also the class variables. Therefore classes reflect certain information about all of their instances. The class hierarchy and aggregation hierarchy also demand sophisticated
locking protocols. For example, if a query is directed against a class (which is a domain of an attribute), it is recursively applied to the subclasses as well. Some type of lock is therefore necessary to ensure no changes are made to any of the subclasses (or instances of the subclasses) for the duration of the query. If a query is directed at a composite object, there must be assurances that the component parts are not modified during the execution of the query.

Object-oriented operations differ from conventional transactions in other areas also. Firstly, object-oriented transactions tend to be long duration (possibly days or months), whereas conventional transactions are very short lived – often less than a few seconds. Secondly, object-oriented operations involve more detailed and often more complex actions (such as in a multimedia database when a document involving text, graphics, sound, and possibly images is addressed). Conventional transactions usually perform a simple series of reads and writes.

**Conventional granularity-hierarchy locking**

The problem involved with lock granularity is to select the optimal size of the lockable unit. For example, if a transaction must access three data elements, it must perform six lock operations (three to lock for access, and three to unlock). This becomes very noticeable if the transaction must access several thousand elements. Therefore, it is essential to find the optimal lockable unit of data.

Unfortunately, there is a tradeoff between maximizing the concurrency and minimizing the locking overhead. The concept of hierarchical locking is proposed by Gray [36] who advocates the use of lockable *granules*. Originally, a conventional database was viewed as a hierarchy of lockable granules (for example, the local database as
Hierarchical locking algorithms employ two types of lock modes: *basic locks* (the read and write locks defined above) and *intention locks*. A basic lock on a coarse granule *explicitly* locks that granule while *implicitly* locking all proper descendants which are finer granules of the coarse granule. If a transaction sets a basic lock on a given vertex of the granularity hierarchy, no other transaction should obtain an incompatible lock on any of its descendants. The intention lock assures this by implying that, once an intention lock is set at a given level of granularity, basic locking is enforced at all lower levels. The mode of the intention lock is determined by the mode of the basic lock. Three intention locks are introduced in [36]:

- *intention read lock*, denoted by IR;
- *intention write lock*, denoted by IW;
- *intention mixed lock*, denoted by RIW, which represents a combination of the basic read lock and the intention write lock

These intention locks are used to indicate that access will be directed at descendants of the given element. For example, an intention read lock IR set on a given granule by a transaction means that the transaction intends to read some of the descendants of the granule. When a transaction sets an intention write lock IW on a granule, this indicates that the transaction intends to write and/or read some of the descendants of the granule. The RIW intention mixed lock implies that the transaction intends to read this particular granule and write and/or read some of the descendants. The compatibility of the intention and basic lock modes is indicated in Table 5.1 [36].
Table 5.1: Lock mode compatibility for hierarchical locking algorithms

<table>
<thead>
<tr>
<th></th>
<th>IR</th>
<th>IW</th>
<th>R</th>
<th>RIW</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>IW</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>R</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>RIW</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>W</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

A hierarchical locking algorithm using the concept of intention locks is presented in [36, 8]. This algorithm is intended for tree granularity hierarchies and represents the protocol a transaction must follow to lock a granule. Locks must be set from the root down. A lock on a granule $x$ itself is an explicit lock on $x$, while locks on proper ancestors of $x$ are implicit locks for $x$. According to this algorithm, a scheduler sets and releases locks for each transaction as follows:

1. If $x$ is not the root, then to set an R lock or an IR lock on $x$, a transaction must have an IR or IW lock on $x$'s parent.

2. If $x$ is not the root, then to set a W lock or IW lock on $x$, a transaction must have an IW lock on $x$'s parent.

3. To read (or write) $x$, a transaction must own an R or W (or W) lock on some ancestor of $x$.

4. A transaction may not release an intention lock on a data item $x$, if it is currently holding a lock on any child of $x$. 
If all transactions obey the above protocol, and if a transaction owns an explicit or implicit lock on a granule, then no other transaction owns a conflicting explicit or implicit lock on that granule [8].

Although this protocol ensures no two transactions ever own conflicting locks, it is not sufficient for serializability. The above protocol governs how locks should be set and released. More details are necessary to determine when the locks should be set and released. Therefore, to ensure serializability, two-phase locking must also be used [8]. A transaction must acquire all of its locks before it can release any locks. Once a transaction has released a lock, it may not set any new ones.

Database elements in a conventional system have to be "organized" in such a way that hierarchical locking can be applied, but in an object-oriented database system, several hierarchies already exist. In fact, two of the hierarchies can be used in viewing the data in a hierarchical format – the class/subclass hierarchy and the class/instance hierarchy. By combining these two, we can apply the techniques of hierarchical locking. The aggregate class hierarchy can also be involved if our data consists of composite objects. We examine each of these separately in the next two sections.

Class/subclass and class/instance locking

Let us first consider the class/subclass (IS-A) and the class/instance (IS-AN-INSTANCE-OF) hierarchies. We note that although many similarities exist between conventional hierarchical locking and these two hierarchies, we point out some differences. The original hierarchical locking algorithm is intended for use on a tree structure. The class/instance hierarchy is such a structure, but the class/subclass
hierarchy may not be because of multiple inheritance. By allowing multiple inheritance, the structure representing the classes and subclasses becomes a directed acyclic graph (dag). The major difference that exists is that in the conventional hierarchical locking, locking a node implies locking all descendants. This is because the descendants of a granule “make up” the granule itself. For example, if a file is divided into records, then locking the file should (and does) lock the individual records. However, locking a class to read the definition should not have any effect on the instances of the class. We should be able, in fact, to modify the values of attributes of an instance of a class when a read lock is present on the class itself.

We can do this because the class serves two purposes: that of a template for instances of the class and subclasses, and that of a grouping constructor. If we are reading a class definition, why shouldn't we be allowed to modify a subclass or an instance of that class? We, therefore, present a new locking protocol to better fit the object-oriented data model. To the best of our knowledge, locking along the class/subclass hierarchy, class-instance hierarchy and composite object hierarchy has been reported only by Kim [49]. We have found no reports addressing locking requirements for the version hierarchy.

We have combined the class/subclass and class-instance hierarchies because many of the same locks can be applied to both. Before we can present our lock compatibility matrix for locks in our object-oriented data model, we present the locks which we employ as they apply to classes and instances.

- R lock: An R lock on an instance means the values of the instance variables are locked in R mode. An R lock on a class means the class variables and class definition are locked in R mode. An R lock on a class does not implicitly
read lock all descendants. Note that this is not a problem when aggregate class variables are present if they are calculated dynamically. When an aggregate class variable is requested in a query, a method is activated that queries all of the instances and produces the necessary aggregate value. To do this, the method locks all of the instances it references. Anyone wishing to change the value of any of these instances must put a W lock on that particular instance, which conflicts with the read lock the method holds to calculate the aggregate variable.

- W lock: A W lock on an instance means the values of the instance variables are locked in W mode. A W lock on a class means the class variables, class definition, all subclasses and all instances are locked in W mode.

Note that instances can only have R or W locks placed on them.

- R* lock: An R* lock on a class means that the class definition and variables are locked in R mode, and all instances of the class and all subclasses of the class are implicitly locked in R mode.

- IR lock: An IR lock on a class means some descendant, either an instance or a subclass will be explicitly locked in R mode or that a subclass will be explicitly locked in R* mode.

- IW lock: An IW lock on a class means some descendant, either a subclass or an instance, will be explicitly locked in W mode.

- R*IW lock: An R*IW lock on a class means the class is locked in R* mode and some instance of either this locked class or a subclass of this locked class will
be locked in W mode. Owning an R*IW lock on a class is logically the same as owning an R* lock and an IW lock on the class. This is useful if a transaction must read every instance of a class and modify some of them.

We present our new compatibility matrix for class and instance locking in the object-oriented data model in Table 5.2.

A locking protocol used in conjunction with the above matrix is necessary. However, if we simply apply the protocol as it has been discussed earlier ([8, 36]), we run into conflicting locks. The difference between applying this protocol to transactions (which was proven correct by Bernstein [8]) and applying it to the class/subclass hierarchy lies in the notion of multiple inheritance. To illustrate why the protocol as presented so far is not sufficient to handle the hierarchy, consider the following example. Figure 5.1 shows a class/subclass hierarchy in which the class E has two superclasses, C and I. If a transaction $T_1$ wants to read C (with an R* lock), $T_1$ must hold an IR or IW lock on the predecessor of C, or A. Suppose it holds an IR lock on A. Note that $T_1$ has now put implicit R locks on subclasses D, E, F and G. Now a transaction $T_2$ wants to write I, so it sets an IW lock on H and a W lock on I. $T_2$ implicitly locks the classes E, F, and G with a W lock. These two transactions

---

### Table 5.2: Compatibility matrix for class and instance locking

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>W</th>
<th>IR</th>
<th>IW</th>
<th>R*</th>
<th>R*IW</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>W</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>IR</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IW</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>R*</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>R*IW</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

---
have followed the protocol, but have conflicting locks. $T_1$ has implicit read locks on $E,F,$ and $G$ and $T_2$ has implicit write locks on $E,F,$ and $G$.

Kim [49] solves this problem by requiring that whenever a transaction sets an explicit $R$ or $W$ lock on a class $C$, it must set explicit $R$ or $W$ locks, respectively, on those subclasses of $C$ which have more than one superclass. In the above example, this means that when $T_1$ sets an $R$ lock on $C$, it must also set an explicit $R$ lock on $E$. Then if another transaction, $T_2$, tries to set a $W$ lock on the class $I$, the request for the lock will be denied. This is because $E$ is a subclass of $I$ and has more than one parent which means $T_2$ must place an explicit lock on $E$ also. But $E$ is already locked in a conflicting mode by $T_1$.

We choose to resolve these conflicting locks in a slightly different manner than Kim [49]. We follow the protocol suggested by Bernstein [8] and require an additional property. That is, to implicitly write lock a node, a transaction must explicitly or
implicitly write lock all parents of the node. Therefore, when \( T_2 \) requests a write lock on \( I \), implicit locks are placed on all descendant nodes. Since \( E \) has more than one parent, \( T_2 \) must have an explicit or implicit lock on \( C \) also. Since \( T_1 \) already has \( C \) locked in read mode, \( T_2 \) is not granted the write lock on \( I \). Both our protocol and Kim's protocol require traversal of the hierarchy from the lock point down (the granule which is actually locked). Kim's protocol, however, requires explicit locks on any subclasses with more than one parent, while ours only requires that it have some type of lock (implicit or explicit).

The locks must be defined to be compatible with certain locks and not compatible with other locks to allow for maximum concurrency yet ensure that no conflicting accesses occur. Clearly a write operation conflicts with everything. We must also address the problem that arises when a write is performed on a class definition, since this affects all subclasses and instances. However, when reading a class definition, we should allow the instances and subclasses to be accessed by other transactions. We have chosen to use \( R^* \) as the lock that will lock everything from the locked granule down which implies that access to any object (whether it be a subclass or an instance) below the locked granule should be denied. We now present our protocol for class hierarchy locking, which includes the class/subclass and class/instance hierarchies:

The Class Hierarchy Locking Protocol (or CHL Protocol):

1. The first object requested by \( T_i \) to lock is the root class of the class hierarchy.

2. If \( x \) is not the root class, then \( T_i \) can set an IR, \( R^* \) or R lock on \( x \) if it is holding an IR or IW lock on some parent of \( x \). That is, to set an explicit R or \( R^* \) lock
on a lockable object \( x \), \( T_i \) must first set an IR lock or IW lock on all ancestors along any one ancestor chain of the lockable object \( x \).

3. \( T_i \) can set an IW, R*IW or W lock on a non-root object \( x \) if it is holding an IW or R*IW lock on all of \( x \)'s parents. That is, to set an explicit W lock on a lockable object \( x \), \( T_i \) must first set an IW or R*IW lock on all ancestors along each ancestor chain of the lockable object \( x \).

4. All locks set by \( T_i \) must be released either when \( T_i \) finishes execution or in leaf-to-root order during \( T_i \)'s execution. This implies that a transaction may not release an intention lock on a class \( x \) if it is currently holding a lock on any child of \( x \).

5. To set an implicit W lock on a class \( x \), a transaction \( T_i \) must have an implicit or explicit W lock on all of \( x \)'s parents.

The goal of the CHL Protocol is to ensure that transactions never hold conflicting locks (explicit or implicit) on the same class or instance of a class. We present the following theorem for the CHL Protocol:

**Theorem 5.1** Assume that all transactions obey the Class Hierarchy Locking Protocol with respect to a given class hierarchy. If \( T_i \) owns an explicit or implicit lock on an object \( x \), then no other transaction owns a conflicting implicit or explicit lock on \( x \).

*Proof:* Assume that transactions \( T_i \) and \( T_j \) own conflicting locks on an object \( x \). There are seven cases we must consider involving conflicting locks:
<table>
<thead>
<tr>
<th>Transaction</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>implicit $r$ lock (R)</td>
</tr>
<tr>
<td>$T_j$</td>
<td>implicit $r$ lock (R)</td>
</tr>
<tr>
<td></td>
<td>explicit $r$ lock (R)</td>
</tr>
<tr>
<td></td>
<td>explicit $r$ lock (R)</td>
</tr>
<tr>
<td></td>
<td>implicit $w$ lock (W)</td>
</tr>
<tr>
<td></td>
<td>implicit $w$ lock (W)</td>
</tr>
<tr>
<td></td>
<td>explicit $w$ lock (W)</td>
</tr>
</tbody>
</table>

**Case 1.** By the definition of implicit read lock, $T_i$ must own an R* lock on $y$, where $y$ is some ancestor of $x$. However, $T_j$, to own an explicit write lock on $x$ (by rule 3), must own an IW or R*IW lock on every ancestor of $x$ along every path from the root class to $x$, including $y$. But this is impossible because R* conflicts with either IW or R*IW.

**Case 2.** By definition, $T_i$ must own an R* lock on $y$, where $y$ is some ancestor of $x$. By definition of an implicit write lock, $T_j$ must own an explicit W lock on an ancestor of $x$. By application of Rule 5, $T_j$ must own an explicit W lock on $z$, where $z$ is on the same ancestor chain as $y$. There are three cases concerning $y$ and $z$:

1. $y = z$. This case is impossible since R* and W locks conflict.
2. $y$ is an ancestor of $z$. To place the R* lock on $y$, $T_i$ must have IR or IW locks on all ancestors along this chain, including $z$. But a W lock conflicts with IR and with IW, so this case is impossible.
3. $z$ is an ancestor of $y$. To place the W lock on $z$, $T_j$ must place IW locks on all ancestors, including $y$. But IW and R* conflict, so this is impossible.
Case 3. The modes R and W and the modes R* and W conflict immediately. It is therefore impossible.

Case 4. By Rule 2, $T_i$ must have IR or IW locks on all ancestors along one ancestor chain, particularly one of $x$'s parents, call it $y$. $T_i$ therefore holds IR or IW locks to the root along this ancestor chain including $y$. By Rule 5, $T_j$ must have implicit or explicit W locks on all parents of $x$, including $y$. If the lock that $T_j$ holds on $y$ is an explicit W lock, there is a conflict because W conflicts with either IR or IW. If the lock that $T_j$ holds on $y$ is an implicit W lock, this implies that $T_j$ must hold implicit or explicit W locks along every ancestor chain of $y$, including the same chain that $T_i$ is holding IR or IW locks. The very definition of implicit W lock implies a W lock along this chain, but W conflicts with either IR or IW. So this case is not possible.

Case 5. By definition of implicit W lock, $T_i$ must have a W lock on some ancestor of $x$, call it $y$. $T_j$ has an explicit W lock on $x$, which implies IW on all ancestors along all ancestor chains, including $y$. But W and IW conflict, so it is impossible.

Case 6. If $x$ is a class, then by rule 5, for $T_i$ to have an implicit W lock on $x$, it must own implicit or explicit W locks on all parents of $x$. This is true for $T_j$ also. But these locks conflict (see Case 4), so the scenario is not possible. If $x$ is an instance, then $T_i$ must hold either an implicit W lock on $x$'s parent class, call it $y$, or an explicit W lock on $y$. The same is true for $T_j$. We have four cases:

1. $T_i$ has an implicit W lock on $y$ and $T_j$ has an implicit W lock on $y$. We have already shown in the first part of Case 6 that this is not possible.

2. $T_i$ has an implicit W lock on $y$ and $T_j$ has an explicit W lock on $y$. This was shown impossible in Case 5.
3. $T_i$ has an explicit W lock on $y$ and $T_j$ has an implicit W lock on $y$. This was shown impossible in Case 5.

4. $T_i$ has an explicit W lock on $y$ and $T_j$ also has an explicit W lock on $y$. These locks conflict immediately, so the situation cannot occur.

Case 7. From the compatibility matrix, W lock and W lock conflict immediately, making the case impossible.

We assumed that two transactions held conflicting locks and have shown that this is not possible. Therefore, no two transactions can ever hold conflicting locks on objects in the class/subclass or class-instance hierarchy. \[QED\]

Aggregate/composite object locking

Many applications that employ object-oriented models require the ability to manipulate complex objects, that is, objects that are composed of other objects. Composite and aggregate objects have been discussed in Chapter 3. In this section, we extend the locking protocol to ensure that concurrent access to an aggregate or composite object does not result in conflicting locks.

Very little research is available for locking in object-oriented database systems, and even less on protocols that cover the aggregate object hierarchy. One protocol is introduced by Kim [49] that addresses locking via the composite object hierarchy. The concept of implicit locking is applied toward the composite hierarchy to reduce the number of explicit locks that must be set and maintained in the system lock table. Several new locking modes are introduced to make the protocol correct. This
particular protocol is quite restrictive in that if there are any readers (or writers) coming through the composite class hierarchy, there cannot be any direct readers or writers accessing the instances of component classes. This is because locks are placed on the class objects that the component objects belong to. Also this protocol may not be suitable for long-duration transactions, since all instances of a component class are affected. Furthermore, the algorithm is very complex, difficult to implement, and time consuming.

We take a slightly different approach that increases concurrency while increasing the number of explicit locks only minimally. We feel that this protocol is more suitable for long duration transactions since it does not implicitly lock all instances of the component object class.

Essentially, we extend the hierarchical locking algorithm reviewed earlier to apply to composite objects. Recall from Chapter 3 that composite and aggregate objects can have dependent references that may be exclusive or shared. A problem arises when two or more objects reference the same component part. Our protocol is quite straightforward and requires no additional locking types.

If a transaction wishes to lock a composite object, it must follow the steps outlined below in the protocol we now present for composite objects:

The Composite Object Locking Protocol (or COL Protocol):

1. To read or write a composite object, a transaction must own an explicit lock on the root composite object itself. To do this, the transaction must follow the class hierarchy locking protocol presented earlier. That is, an intention lock must be placed on the parent (and, depending on the type of lock, up one or
all of the ancestor chains).

2. The desired component object referenced by the root object must have a matching lock explicitly placed on it also. This requires intention locks to be placed on the class object corresponding to the independent object being referenced. The transaction must follow the class hierarchy locking protocol to place EACH of these explicit locks on any desired component objects.

3. Once the explicit lock has been placed on the desired component object (referenced object), the lock on the root object can be released.

The goal of the COL Protocol is to ensure that transactions never hold conflicting locks (explicit or implicit) on the same component object instance which is part of the aggregate object hierarchy and also part of the class-instance hierarchy. We present the following theorem for the COL Protocol:

**Theorem 5.2** Assume that all transactions obey the Composite Object Locking Protocol with respect to a given class composition hierarchy. If $T_i$ owns an explicit or implicit lock on an object $x$, then no other transaction owns a conflicting implicit or explicit lock on $x$.

*Proof:* The proof is similar to that for the CHL Protocol. Assume that transactions $T_i$ and $T_j$ own conflicting locks on an component object $x$. There are seven cases we must consider involving conflicting locks:
transaction $T_i$  
transaction $T_j$

1. implicit $r$ lock  explicit $w$ lock  
2. implicit $r$ lock  implicit $w$ lock  
3. explicit $r$ lock  explicit $w$ lock  
4. explicit $r$ lock  implicit $w$ lock  
5. implicit $w$ lock  explicit $w$ lock  
6. implicit $w$ lock  implicit $w$ lock  
7. explicit $w$ lock  explicit $w$ lock  

Case 1. By the definition of implicit read lock, $T_i$ must own an $R^*$ lock on $y$, where $y$ is some ancestor of $x$ (on the class/subclass or class-instance hierarchy). However, $T_j$, to own an explicit write lock on $x$ (by rule 3 of the CHL Protocol), must own a $W$ lock for some ancestor of $x$ along every path from the root class to $x$, including $y$. But this is impossible because an $R^*$ and $W$ lock conflict.

Case 2. If $T_i$ and $T_j$ each own implicit locks on $x$, then the locks must have been placed via the class/subclass or class-instance hierarchy since implicit locks are not placed on component parts via the class composition hierarchy. We have already shown that this case is impossible in the proof for the CHL Protocol.

Case 3. The modes $R$ and $W$ conflict immediately. It is therefore impossible.

Case 4. $T_i$ must have IR or IW locks on all ancestors along one ancestor chain, particularly one of $x'$s parents, call it $y$. $T_j$ must have implicit or explicit $W$ locks on all parents of $x$, including $y$. But IR and explicit $W$ locks conflict. Therefore, $T_j$ and $T_i$ cannot own these locks simultaneously, making the situation an impossibility.

Case 5. By definition of implicit $W$ lock, $T_i$ must have a $W$ lock on some ancestor of $x$, call it $y$, along the class/subclass or class-instance hierarchy. $T_i$ must own implicit
or explicit W locks on all parents of \( x \). \( T_j \) must have W lock on \( x \), which implies IW on all parents. But W and IW conflict, so it is impossible.

**Case 6.** Implicit write locks can only be placed on \( x \) via the class/subclass or class/instance hierarchy. We have already shown in Case 6 of the CHL Protocol that this situation is not possible.

**Case 7.** From the compatibility matrix, W lock and W lock conflict immediately, making the case impossible.

We assumed that two transactions held conflicting locks and have shown that this is not possible. Therefore, no two transactions can ever hold conflicting locks on component objects. \[ QED \]

Consider the following example: We have several documents that consist of paragraphs. Documents D1 and D2 both share paragraph P. To lock D1 for read access, the transaction must lock the object class D1 belongs to with an intention read lock (which requires intention read locks from the root down), and then lock D1 itself with a read lock. At this time, desired component parts must be locked in the corresponding mode. Therefore, the transaction must lock paragraph P by placing a read lock on it also. This requires an intention read lock on the class to which P belongs. Now, if another transaction comes along and wishes to modify D2, a write lock must be placed on the object D2, which requires intention write locks from the root to the class object for D2, along every ancestor chain. A write lock must also be placed on desired component parts, or paragraph P. But the first transaction is already holding a read lock on P, and a write lock is not compatible. Therefore, the second transaction will block until the first is done.
With the protocol presented in [49], all instances in the class object for paragraph are locked implicitly. Access is severely restricted, especially if we are dealing with a long duration transaction. Our protocol is more efficient for long-duration transactions because only the referenced object has to be locked.

Version locking

Recall from Chapter 3 that versions introduce two new relationships, namely the "derived-from" relationship and the "version-of" relationship. A new version may be derived from some existing version and is a version of some versionable object. Within the version-of relationship, locking is done as within the class/subclass hierarchy or aggregate/composite object hierarchy. For example, if we have a versionable object and several versions of that object exist, locking version 1 has no affect on what can be done with version 2, provided they are not related by the derived-from relationship. In essence, these two versions are objects in the database, related only because they are versions of the same versionable object.

However, if two versions are related with the derived-from relationship, then we must take this into consideration when designing a locking protocol. These two versions may share a component or aggregate object. To illustrate, suppose that version 2 is derived from version 1, and both share object \( O \), which remains unchanged from version 1 to version 2. Then if we want to modify version 1, we should lock out access to version 2. However, modification to version 2 can also affect version 1. Locking version 2 and then changing object \( O \) affects version 1. This results in the following protocol for version locking (Version Locking Protocol):
The Version Locking Protocol (or VL Protocol):

1. Only R, IR, W, and IW locks apply in the version hierarchy. R* locks are not placed on version objects.

2. A transaction \( T_i \) wishing to lock the versionable object simply follows the class hierarchy locking protocol (as presented earlier.) The versionable object is treated as any object in the database. The versionable object must be locked before any versions can be locked.

3. To lock a version object \( O_v \), which is not related to any other version objects by the "derived-from" relationship, in either read or write mode, an explicit read or write lock must be placed on the version object.

4. To lock a version object \( O_v \), which is related to other objects by the "derived-from" relationship, intention locks must be placed on the appropriate "relatives". To lock a version object \( O_v \) in read mode, a transaction \( T_i \) must first place intention read or intention write locks on all ancestor objects related to \( O \) via the "derived-from" relationship. To lock a version object \( O_v \) in write mode, a transaction \( T_i \) must first place intention write locks on all ancestor objects related to \( O_v \) via the "derived-from" relationship.

Intuitively, the above can be expressed as follows. When reading a version object related to other version objects by the "derived-from" relationship, notification of this reading must be passed up the hierarchy to all ancestors to make sure no writing is done at any of the upper levels. When writing a version object of this type, we must make sure implicit write locks are enforced on all lower version objects derived from the one with the explicit write lock, as well as intention write locks on all ancestors.
The goal of the VL Protocol is to ensure that transactions never hold conflicting locks (explicit or implicit) on version instances of a class which are related by the "derived-from" relationship. We present the following theorem for the VL Protocol:

**Theorem 5.3** Assume that all transactions obey the Version Locking Protocol with respect to a given version hierarchy. If $T_i$ owns an explicit or implicit lock on a version object $x$, then no other transaction owns a conflicting implicit or explicit lock on $x$.

**Proof:** This proof is again similar to those for class hierarchy and composite object locking. Assume that transactions $T_i$ and $T_j$ own conflicting locks on an version object $x$. Since $R^*$ locks are not applied in the version hierarchy, Case 1 and Case 2 (involving implicit read locks) do not exist. Therefore, we must consider five cases involving conflicting locks:

- **transaction $T_i$**
- **transaction $T_j$**

1. explicit r lock explicit w lock
2. explicit r lock implicit w lock
3. implicit w lock explicit w lock
4. implicit w lock implicit w lock
5. explicit w lock explicit w lock

**Case 1.** The modes R and W conflict immediately. It is therefore impossible.

**Case 2.** $T_i$ must have IR or IW locks on all ancestor version objects related to $x$ by the "derived-from" relationship, including a version $v$. $T_j$ must have implicit or explicit W locks on all ancestors of $x$ related to $x$ by the "derived-from' relationship, including $v$. But IR and explicit W locks conflict. Therefore, $T_j$ and $T_i$ cannot own these locks simultaneously, making the situation an impossibility.
Case 3. By definition of implicit W lock, $T_i$ must have a W lock on some ancestor of $x$ (related to $x$ by the "derived-from" relationship), call it $y$. $T_i$ must own implicit or explicit W locks on all such ancestors $x$. $T_j$ must have a W lock on $x$, which implies an IW on all ancestors, including $y$. But W and IW conflict, so it is impossible.

Case 4. For $T_i$ to have an implicit W lock on $x$, it must own implicit or explicit W locks on all ancestors of $x$ related by the "derived-from" relationship. This is true for $T_j$ also. But these locks conflict, so the scenario is not possible.

Case 5. From the compatibility matrix, W lock and W lock conflict immediately, making the case impossible.

We assumed that two transactions held conflicting locks and have shown that this is not possible. Therefore, no two transactions can ever hold conflicting locks on version objects related by the "derived-from" relationship in the version hierarchy if they follow the VL Protocol. $QED$

Lock inheritance

We now return to the environment of nested transactions to address our new locking protocol. Moss [63] presents rules for nested transactions and lock inheritance. When a transaction commits, its parent (if any) holds the locks of the committing transaction. If the parent already holds a lock, it will begin to hold the lock in the mode which is more exclusive of the modes of the parent and the child. We must now determine the definition of "more exclusive" in the context of the locks used in object-oriented systems.

The work done by Moss does not address the addition of intention read and
intention write locks. To the best of our knowledge, there has been no work done on developing a hierarchy which includes intention locks. We have also introduced a new lock, the R* lock which must be inserted in the hierarchy. Clearly a W lock is more exclusive than an R lock. The addition of IR, IW, R*, and R*IW require a new hierarchy to define “more exclusive”. The five locks (IR, IW, RIW, R, and W) that Bernstein [8] presents can be ordered according to the hierarchy found in Figure 5.2.

The W lock is still the most exclusive. An RIW lock is more exclusive than an IW lock or an R lock. IW and R locks are equal in the hierarchy. An IR lock is the lowest lock, and is dominated by all others. To illustrate how this hierarchy works, consider a transaction T that creates two subtransactions T1 and T2. Suppose T1 holds an IR lock on a granule x, and T2 holds an IW lock on the same granule x. When T1 commits, T inherits the IR lock on x. When T2 commits, T inherits the IW lock. Since T already has an IR lock on the granule x, it will hold the lock on x which is more exclusive. Therefore, T will hold an IW lock on x.

Figure 5.2 includes only the modes introduced by Bernstein. The new locks R* and R*IW must be incorporated into the hierarchy as well. Figure 5.3 illustrates
Figure 5.3: A lock hierarchy with all locks

how these locks are incorporated into the hierarchy. This hierarchy mandates how locks are to be inherited by parent transactions in our object-oriented database environment. The following example illustrates the application of this hierarchy.

Consider a transaction $T$ with three subtransactions, $T_1$, $T_2$, and $T_3$. Suppose that $T_1$ acquires an IR lock on granule $x$, $T_2$ acquires an $R^*$ lock on $x$, and $T_3$ obtains an $R^*W$ lock on $x$. As each subtransaction completes, $T$ inherits the lock according to the hierarchy in Figure 5.3. When $T_1$ completes, $T$ inherits the IR lock on $x$. When $T_2$ completes, $T$ compares the IR lock to the $R^*$ lock and begins to hold an $R^*$ lock on $x$, since $R^*$ is more exclusive. When $T_3$ completes, $T$ compares the $R^*$ lock it is now holding on $x$ to the $R^*W$ lock that $T_3$ was holding. Since $R^*W$ is more exclusive, $T$ begins holding an $R^*W$ lock on $x$.

By always ensuring that the parent transaction holds the most exclusive lock of its children on any given granule, we ensure that no outside transaction can interfere with the actions that the children have performed. It may be the case that certain subtransactions will need to be aborted and their operations on the database must be
undone. By giving the subtransaction's lock to the parent, the parent can keep the effects of the subtransaction hidden from the rest of the transactions in the database until either the top-level transaction commits or aborts.

Secure Transaction Processing

Transparent concurrency control and recovery are two necessary elements when dealing in a database environment. Transactions provide this transparency. In a secure database scenario, it is also important that the transactions execute securely. Very little research has been done in this area. This section discusses transaction security and presents an approach that uses nested transactions.

Transaction security policy

Database accesses by transactions are governed by the security policy. Transactions execute on behalf of the user, and as the transactions execute, methods are invoked in response to messages received from the transactions. We differentiate between a transaction (as a set of operations executed on behalf of the user) and a method, which can simply be called an action. As the transaction executes, messages are sent to objects in the database, which in turn respond by executing methods (or performing actions). The security policy presented in Chapter 4 is enforced at the action level. This enforces the security constraints as methods are invoked. Refer to the section on security methods from Chapter 4.

These properties must now be put into the context of transactions as follows, where $M_2'$, $M_3'$, $M_5'$, and $M_6'$ are the same as $M_2$, $M_3$, $M_5$ and $M_6$ in the section on security properties for methods from Chapter 4.
M1/. A method first called by a transaction executing on behalf of the user begins execution at the security level of the transaction which called it, or $LT_{current} = L_{T_{caller}}$, where $T_{caller}$ is the calling transaction.

M2/. A method may read the value of a labeled object $O$ with sensitivity level $L_O$ if $L_O \leq L_{S_{clear}}$. An unreadable object returns \textit{nil}.

M3/. A method may create and store a labeled object with sensitivity level $L_O = L_{S_{current}}$. If the object is constrained to have the security range $[L_{bottom}, L_{top}]$, then the object can be stored if $L_{bottom} \leq L_{S_{clear}}$ and $L_{S_{current}} \leq L_{top}$. Otherwise the update is rejected.

M4/. As a method executes, its current classification label changes dynamically. A method starts initially with $L_{S_{current}} = L_{T_{caller}}$ if called directly by a transaction and $L_{S_{current}} = L_{S_{caller}}$ if invoked by another method. If the method accesses an object $O$ such that $L_{T_S_{current}} < L_O \leq L_{S_{clear}}$, the current classification level of the method will be assigned that of the object, or $L_{T_S_{current}} = L_O$. The classification level changes dynamically to reflect the classification of objects the method has had direct or indirect (through other method activations) access to.

M5/. A method's security classification label can never exceed that of the clearance of the user activating the method (either directly or indirectly, via another method). That is $L_{S_{current}} \leq L_{S_{clear}}$ for every method invocation at all times.

M6/. An object $O$ returned upon method completion is labeled with the classifica-
tion of the method at the time the object is returned, or $LO = L_{current}$. The classification changes dynamically to reflect the classification of objects the method has accessed, either directly or indirectly (through other method activations).

**Security inheritance via nested transactions**

As a transaction executes, it can be divided into subtransactions. A problem arises with information flow. If some subtransactions execute at a high level and then finish, information can be transferred to the parent transaction. If a new subtransaction is then executed to write data at lower level, then unauthorized information flow could occur.

**EXAMPLE**: Consider the following scenario: Transaction $T$ has subtransactions $T_1, T_2$ and $T_3$. These three subtransactions are responsible for making the actual accesses to the database. Suppose $T_1$ and $T_2$ both begin execution and $T_1$ finishes and returns a result before $T_3$ begins. If the result returned is at the secret level, and $T_3$ must modify data at the unclassified level, there is a possibility of an information flow violation.

The idea of nested transactions can be used to increase concurrency. The hierarchical structure of the nested transactions can also be used to inherit the security of the methods executed on behalf of the subtransactions. As each transaction executes, it sends messages to different objects, which in turn execute the methods associated with the messages. As the method execution finishes, a security label is associated with the method to indicate the security level that particular method has had access
to, either directly or indirectly via other method calls. A transaction might invoke several different methods. When the methods are finished, the transaction assumes the security level of the method that it first invoked.

Incorporating this nesting into our transaction security policy and using the hierarchical structure provided by subtransactions yields the following transaction security policy:

TS1. A parent transaction begins execution at the lowest possible level (system LOW), or $L_{TS\text{current}} = UNCLASSIFIED$.

TS2. A parent transaction is passed the clearance of the user, $L_{Sclear}$, to act as an upper bound on its security range, or $UNCLASSIFIED \leq L_{TS\text{current}} \leq L_{Sclear}$.

TS3. When a child transaction is created, it is passed the current security level of the parent transaction, $L_{TS\text{current}}$, and the upper limit, $L_{Sclear}$.

TS4. As a leaf node child performs the actual calls on the database (invoking methods), it passes its current security classification and its upper limit to the method(s) it invokes. The method(s) will then use this range and apply the security constraints applicable to methods. Each method invoked will pass back its final security level, $L_{M\text{current}}$, to the transaction which called it.

TS5. If, at any time, a method cannot complete because of a security violation, the calling transaction will abort.

TS6. When a subtransaction commits, it must pass its current security level to its parent. The parent then assumes the maximum security level of the (child,
current) pair. If $LSS_{\text{current}}$ is the current security level of the subtransaction, then $LTS_{\text{current}} = \max(LSS_{\text{current}}, LTS_{\text{current}})$.

TS7. If at any time, a subtransaction must abort, the parent need not abort. This allows the parent transaction to retry later or invoke a different child transaction to complete the necessary work.

**EXAMPLE:**

Consider a transaction $T$ with subtransactions $T_1$ and $T_2$. Suppose $T$ is initiated by a user with a clearance of secret. $T$ begins at an unclassified level. It creates $T_1$, which also begins at unclassified, but carries the upper security bound of secret (i.e., $LS_{\text{clear}} = \text{secret}$). $T_1$ now invokes a method in the database to access data. The method begins execution at level unclassified also. As the method executes, it accesses confidential data, which is allowable because $LS_{\text{clear}} \geq \text{confidential}$. When the method is done, the classification of confidential is passed back to $T_1$. This transaction is now finished, so it passes this security classification back to $T$, which sets $LTS_{\text{current}} = \text{confidential}$. If $T_2$ is created after $T_1$ finishes, $T_2$ must begin with a level of confidential. If the methods that are invoked by $T_2$ try to read top secret data or write unclassified data, the subtransaction $T_2$ will abort.

The transaction security policy leads to the following theorem:

**Theorem 5.4** If all transactions follow the transaction security policy, there will be no unauthorized flows of information.

*Proof:* To guarantee no unauthorized information from the database flows to the user, our policy must satisfy two conditions:
1. It must ensure proper classification of all values

2. It must prohibit direct flow to the user

The transaction security policy ensures all information is properly labeled. Any access to secure data requires the method performing the access to upgrade its security level to that of the accessed information after the access. All transactions must assume the security level of any method executed on their behalf. All parent transactions assume the maximum security level of their children. At any given time, the security level of a transaction reflects the maximum level of security of data it has had access to. Therefore, if a transaction invokes a method to write a value, the method is invoked at the level of the transaction. This implies the write must be performed, labeling the object being written with the proper security level (because methods must adhere to the method security policy), or the write is not performed at all. Since all write operations are performed at the level of the method that executes them, it is not possible, following this transaction security policy, to mislabel information, which would allow unauthorized flow to the user.

Direct flow of unauthorized information to the user is also prohibited by requiring the clearance of the user to be passed to every transaction and all methods invoked by transactions. The clearance is used as an upper bound for all accesses in the database. Therefore, no direct access to unauthorized information is possible, thus stopping any direct flow to the user. Our policy satisfies the two specified conditions and therefore all flows are authorized.

QED
Failure and recovery

As mentioned, when a child aborts, the parent need not abort. Actually, if a subtransaction aborts, it can be retried. There are different reasons for a transaction to abort. A hardware failure can result in aborted transactions. Hardware failures can result from an actual system failure, a link failure, or a network partition. System failures result from system crashes and disk failures. If any of these situations occur, the transaction will need to be rerun, after the database is recovered. Another possibility is that the user running the transaction signals the transaction to abort. This might happen if the user has decided the information is no longer necessary, the transaction needs to be modified, or the user simply wants to terminate the transaction. If the transaction needs to be modified, then the modification can be performed and the transaction can be rerun. The third possibility results when there is a problem with locking, causing a timeout and an aborted transaction. If the abortion is caused by a locking problem, resubmitting the transaction often results in a successful execution.

The above scenarios cover common causes for aborting transactions. In a secure environment, however, one more possibility arises. A security violation can cause a transaction to abort also. However, if the aborted transaction is the result of a violation of the security constraints imposed on the system, resubmittals will never lead to a committed transaction. We can retry the transaction with the possibility of success if we are willing to accept less precision in the result. For example, suppose that a transaction requests a launch date and time from a database. The date and time together (9/4/91 0800:00) are top secret. A resubmission of the request might be less demanding in that it only needs the date (9/4/91), which might be secret.
If an unclassified user is requesting this data, this is still a security violation. A resubmission might request only the month and year (9/91), which still might be confidential. If the year by itself (1991) is unclassified, a request for this information will be granted. Therefore, a modified retry can yield successful results, thereby averting the need for the transaction to abort. In fact, TS7 is motivated by the previous discussion on failure and recovery.

Summary

This chapter has provided an approach for handling transaction management and processing by using nested transactions in an object-oriented database. A locking policy based on granularity locking is presented, which applies directly to the hierarchical nature of an object-oriented database. Class hierarchy locking is covered, as well as aggregate or composite locking.

We have introduced a lock hierarchy to govern the inheritance of locks from the subtransactions to the parent transaction. This lock hierarchy includes not only the read (R) and write (W) locks discussed by Moss, but also the intention read (IR), intention write (IW) and read-intention write (RIW) locks used in hierarchical locking schemes. To better fit the object-oriented environment, we defined the R lock as a lock which locks only the class definitions, methods and variables, whereas our new lock, the R* lock, locks the class plus all subclasses and instances – essentially everything in the class/subclass and class/instance hierarchies from the locked class down. By introducing this new lock, we provide more opportunity for concurrency.

We have presented a protocol for secure transaction processing based on locking and nested transactions. Transactions inherit security levels from their child trans-
actions, indicating the flow of information in the system. Transactions must adhere to a security policy to ensure no security violations of information flow occur during execution. These ideas have been presented in [65].

The next chapter deals with query processing in an object-oriented database. The idea of a query graph is presented, along with the notion of a view graph. Secure query processing is then addressed.
CHAPTER 6. SECURE QUERY PROCESSING IN OBJECT-ORIENTED DATABASES

Introduction to Views

The main purpose of a database management system is to manage and maintain the collection of data, thus freeing the user from this complicated yet tedious task. Whatever means the DBMS uses to do this, there are two main requirements – it must handle data retrieval efficiently and conveniently and it must also hide enough of the complexity to allow people to use the database without requiring them to have significant knowledge about how the data is stored and the different operations used for managing the data. The DBMS meets these requirements by defining several levels of abstraction at which the user can view the database: [50]

1. **Physical level.** This is the lowest level of abstraction and describes how the data are actually stored. The complex low-level data structures used at this level are described in detail.

2. **Conceptual level.** This is the level above the physical level which describes what data are actually stored in the database and any relationships that exist among data. The entire database is described in terms of smaller, relatively simple structures. This is the level that database administrators use, because
they must determine what information is to be kept in the database and how it should be inter-related.

3. **View level.** This level represents the highest level of abstraction, but describes only a part of the database. Most users need access to only a small portion of the data stored in the database. To simplify how these users interact with the system, the view mechanism is defined. A user's view is simply a subset of the database which that particular user has interest in.

A major task of the database management system is to provide multiple users with multiple abstract views of the data. Views can then be considered "virtual relations" for each user. Any query by the user is directed to this virtual relation and all data retrieved comes from this subset of the database.

**Secure Views**

The concept of a view provides a mechanism to design a personalized subset of the database which is often used for access control. Views can be used to hide information a user should not see. Views are employed, then, to simplify using the database. However, if a user is restricted to this view only, security can be enforced within the database environment. In relational databases, security is addressed at two levels:

1. **Relation.** User authorization is checked at the relation level, permitting or denying access to particular relations that exist in the database.

2. **View.** User authorization is checked at the view level, permitting or denying access to particular views that exist in the database.
By combining the above two mechanisms, a user's access can be limited to a precise set of data the user needs. For example, a user may be denied direct access to an entire relation, but may be permitted access to a view containing some of the data in the relation.

Related Work

A significant amount of research has been done on using views for security in relational database systems ([23, 25, 26, 29, 62]). Because views are defined arbitrarily, they have been proposed as a way of handling many of the issues involved with secure query processing, such as context- and content-dependent classification of data, dynamic classification of data and users, inference and aggregation.

Basic view concepts for a multilevel-secure relational database model are presented in [23]. This model allows for the specification of derived data in the database schema so that any relationships between the stored and derived data (inferences) can be expressed formally. The difference between a view that retrieves or updates data (called an access view) and a view that classifies the data (called a classification view) is explicitly presented. A view is defined to be a mapping (multivalued function) from a database (set of relations) to a relation (or set thereof). The elements used to compute the resulting relations are called the view source. This source consists of all elements which are named in the view mapping and whose tuples are selected by various conditions in the mapping. The term "view" in this paper refers to the specification or mapping function, whereas often times the term "view" refers to the actual data returned when a view specification is applied to the database. Each view is defined by a view specification or definition which has an access class. If the access
class of the view specification is not dominated by the access class of the user, then this particular user cannot apply this particular view. Views serve two purposes, one of labeling new data (these views are called classification constraints), and one for accessing data (these views are called access views). These “mapping functions”, or views, are then used to label any new data with an access class and to actually access the data.

Another idea on using views is found in [25, 53]. Both of these references address the SeaView Security Model. This is a formal security policy model that uses basic view concepts for a secure multilevel relational database system. Views are used for specifying instances of multilevel relations at different access classes, for discretionary access control, for assigning labels to new data (through classification constraints) and for sanitization (a method allowing subsets of aggregate information to be released which ensures that mandatory security is enforced). The model has two layers, one which corresponds to a security kernel or reference monitor that enforces mandatory security (the Mandatory Access Control or MAC layer), and the second which defines multilevel relations and formalizes policies for labeling new and derived data, data consistency, discretionary security, and transaction consistency (the trusted computer base or TCB layer).

The majority of research involved with using views to enforce security has been aimed at relational database systems. Very little work has been done in the area of views in object-oriented databases. Views can provide object-oriented database systems with the same types of uses as are found in relational systems. That is, views can be used to provide certain subsets of instances for particular users, thus providing a useful partition of the data. Authorizations can be enforced by placing
only certain instances in user views. Users can experiment with various changes to the data or schema within their views, without making permanent changes to the contents of the database [49].

Abiteboul and Bonner [1] introduce a sophisticated view mechanism for object-oriented database systems to overcome some of the difficulties encountered when trying to integrate databases or restructure data. They present a view mechanism that allows a user to restructure the class hierarchy and modify the behavior and structure of objects. New virtual classes can be introduced into the class hierarchy and can be populated by existing objects or by creating new imaginary objects.

In [1], views are equated with queries. This view model is presented in the context of the O2 data model [4]. The authors blur the distinction between attributes (stored values) and methods (behaviors), thus treating attributes and methods as equals. These virtual attributes allow the view mechanism to split complex attributes into several simpler ones or to restructure several attributes or objects simultaneously. For example, the attributes Home and Office, both consisting of an address and a phone number can be restructured into the attributes Addresses and Telephones, consisting of the home and office phone number.

Importing and hiding are also applied. A view imports its data from other relations by using an import statement. A hide command allows certain data that have been imported to be hidden from users. For example, by using the following statement:

\[
\text{hide attribute Salary in class Employee}
\]

the salary is hidden in this class and any other subclasses that are involved in calculating the view.
New classes, called virtual classes essentially reflect the view that a user will have of the data in the database. The virtual class's population, position in the class hierarchy, and behavior of its objects are all defined. The population is defined by stating what the view should contain, and the position and behavior are system determined. The flexibility of the view mechanism is enhanced by introducing the notions of behavioral and parameterized classes. Behavioral classes are those that are grouped according to behavior, and parameterized classes are those which define classes based on parameters that can change and thus yield new classes.

These virtual classes are created by using a create view command. The classes are populated with existing object instances by several different methods:

1. Specialization: using a query to specify the immediate instances of a class that should belong to the new virtual class

2. Generalization: specifying the subclasses that the virtual class should include (which automatically includes the instances of these subclasses), and

3. Behavioral generalization: grouping classes with similar behaviors and including all instances of these classes

The system must then determine where this new virtual class is to be placed in the class hierarchy. But because of the way virtual classes are populated, there are two basic mechanisms by which a virtual class can acquire attributes: from its superclass (the "standard" object-oriented way), and by inheriting attributes that are common to all the objects in the class.

The above methods result in actual "real" objects in the virtual class. That is, the objects actually exist in the database, in other classes. However, it is possible for
the virtual class to be populated by creating new, imaginary objects. These objects exist only in the view, and not in the database itself. An example of this is to create a view which takes a database of people and creates a view of families. A particular family is not an actual object in the database, but will be treated as such by this particular view. These imaginary objects must also be given object identifiers so that they can be addressed by the system. If virtual classes are recalculated each time a query is made, assigning the same object identifier to the virtual and imaginary classes becomes an issue. It is resolved by picking core attributes (similar to keys in a relational system) and putting the object identifier/core attribute pairs in a table. When the virtual classes are recalculated, the core attributes are used to scan the table and pick the same object identifier.

**Our Approach to Secure Views**

The ideas presented in [1] lack certain notions which are important in an object-oriented database environment, particularly our query processing model. We modify a few of the notions and add further enhancements to better suit our model.

1. We do distinguish between an attribute and a method. Many times a method will not only calculate a value (which is simply the computed attribute that [1] refers to), but will also perform a procedure that has lasting effects on the data. For example, a method that simply calculates the average of a list of values could be treated as an attribute. But consider the method that performs a spelling check on every paragraph of a document in a multimedia database. It doesn't calculate or return a single value that can be deemed an attribute.
It makes modifications to existing objects in the database. Therefore, we do not blur the distinction between attributes and methods.

2. A query and a view are not equivalent. We expand on this difference and treat a view as THE subset of the database the user is allowed to access. The user will be given a view, and it is this “virtual database” that the user must apply queries to.

3. Because our view represents a virtual database, much can be gained if the virtual classes are placed in cache at the local site instead of recalculated each time. This way, the user can make several queries without recalculating the virtual class every time. This saves system and communication resources and speeds up response time.

Integrating our security policy with the notions of virtual and imaginary classes enables us to formulate a method for implementing secure views. The idea of a schema graph for a class C is presented in [49]. This is a graph of the class/subclass hierarchy (IS-A) and the aggregate class hierarchy rooted at class C. A schema graph for the employee example presented earlier is given in Figure 6.1.

A query graph is also introduced, which is a subgraph of the schema graph, to describe a domain for the query. It represents, essentially, those classes and attributes that the query must use to execute. In our example, suppose we formulate a query to find the salaries of all employees in the advertising department. Figure 6.2 illustrates the query graph for this query.

We use our security policy to implement secure views by forming a view graph. A view graph built on top of a query graph is essentially a subset of the query
graph. The query graph is reduced in size by using the security properties above and pruning the tree whenever possible. For example, the fact that SATURN-PROJECT in Figure 6.2 has no instances with levels below top secret (thus placing a derived level on the class SATURN-PROJECT of top secret) allows us to prune this particular branch from the view graph. Only those instances with classification labels dominated by the user’s clearance are in the view graph. A view graph will exist for every user, so the discretionary access controls found in the negative authorization lists can be applied to these view graphs also. The view graph is a dynamically changing entity and reflects modifications to the database and to the user's clearance. A view graph is illustrated in Figure 6.3.
We have presented a scenario of first establishing a schema graph, then a query graph, and finally a view graph. What this implies is a query evaluated against the entire database and then the security enforced on the resulting query graph. The query graph depicts that portion of the database involved with the query. The view graph further decreases the number of objects to be dealt with by the query by removing any objects the user (posing the query) cannot see.

When the graphs are derived in this particular order (schema → query → view), what we are doing is making the query graph secure, resulting in a *secure query graph*. However, some administrators might prefer a secure view be derived first, so the query is posed to a smaller subset of the database. In this scenario, the schema
Figure 6.3: A view graph

What we have now established is that a secure view can first be generated for a particular user or set of users. This view can be retained in the database by creating a virtual or imaginary class to act as the root of the view. This new class can then be added to the class hierarchy, making the user's view "permanent" in the sense that the class will remain part of the class hierarchy while the user is querying the database. Now queries can be directed at this subset of the database.

Therefore, a secure query graph can be derived in one of the two following orders:
Figure 6.4: Two methods for deriving a secure query graph

1. Schema → Query → View

2. Schema → View → Query

Figure 6.4 illustrates this idea.

The method employed to gain optimal performance depends on the application and the types of user access and queries. With the first case, the query must be evaluated against the entire database, which is very time consuming. If many different users with different clearances are going to make the same queries against a database, this is the method of choice. Since the query is the same and only the user view changes, when the query graph is calculated first, the calculation of unique view graphs for different users is simplified because the size of the database has been reduced. An example of this situation is when several people with varying classification levels were going to query a database and ask for employee salaries. The query graph is formulated first, removing all objects and attributes not relevant to the query. Then, a secure query graph is derived as each user queries the database.

However, if one user makes several different queries on the data, the second method would be more appropriate. This method requires the entire database to be surveyed to find the view graph for a particular user. Once the view graph has been determined, new query graphs can be calculated using the view graph as the basis.
instead. An example of this situation is when a user is making many different queries against an employee database, such as a query for names, then one for salaries, then one for addresses, etc. In this situation, the user's view graph would be formulated first, followed by finding the proper subset of the graph for each query. To summarize, if the database is accessed frequently by the same user, then the second option is the more desirable since many queries can be applied to the same view graph. However, if many users access the database, often posing the same query, then deriving the query graph first is more appropriate and efficient.

Secure Query Processing

When a query is posed against a multilevel secure database, several different problems must be addressed. Since the classification in a database can be at varying granularities, we must ensure that those users authorized to see various granules can actually access those granules, and un-authorized users must be prohibited from accessing these granules. Access to these data elements is often based on more than just user clearance and discretionary access lists. Often data is classified according to context, content, time and aggregation. Furthermore, it is often possible for users to infer unauthorized information based on information they are allowed to access. Secure query processing is an area that attempts to deal with these problems.

Secure query processing can be implemented by posing normal queries to the database and checking all responses generated by the DBMS. If the response will violate any security constraints, the response is not released. A modification of this approach yields yet another secure query processing method. The response can be
"filtered", thus removing all information that causes a security violation and only releasing that information which the user is authorized to see [21].

Another approach to secure query processing is to use query modification [19, 77, 83, 43], which can enforce integrity constraints as well as provide view mechanisms. The normal functions of the database are kept separate from the security issues using this approach. Query modification is the process by which a query is modified before it is evaluated against the database. The original user query is replaced by a query that will perform the desired function. Consider an example query to list all salaries and employee names. If this query is posed by an unclassified user and all salaries are classified as secret, then the query would be modified to list all of the names only.

This particular example can be handled by the security policy already enforced in our environment if every salary has a classification label of secret. That is, an unclassified user could not read a secret salary because the method acting on behalf of the user would not be allowed to access the required information. However, the problem arises with other security constraints.

Security constraints can be classified into three groups:

1. **Simple Constraints.** These are the security constraints that assign classification levels to the entire database as well as to individual attributes. An example of a simple constraint is: All employee salaries are secret.

2. **Content-Based Constraints.** Using this type of constraint, a classification level is assigned to an element based on its content. An example of a content-based constraint is: Employee names are classified as secret if the employee's salary is greater than 100K.
3. Context-Based Constraints. These constraints classify relationships between data. An example of a context-based constraint is: Names and salaries taken together are secret. This could be modified further to allow one or the other of these two attributes to be released at the unclassified level, but once either is released at the unclassified level, the other becomes classified at the secret level.

Simple and content-based constraints apply to actual objects in the database, and a classification label can be assigned to the object in question. This way, the classification is actually stored with the object. However, context-based constraints address the relationships between objects, and the classification label cannot be stored with the object. These constraints must consider environmental information (such as what has been released to whom) as well as the values and attributes themselves. In doing so, user inference is addressed. Query modification is the process by which a query is modified according to the relevant security constraints so that when the modified query is posed, the response will not violate any of these security constraints.

Rule base

Extracting the appropriate constraints to perform the query modification is a major issue in query modification. One solution is to append a rule base to the database to represent the constraints and environmental histories necessary to guard against user inference [11, 31, 43, 82]. A rule base also allows for more complex mandatory security checks than simply comparing the user’s clearance to the classification of the object. This is very useful in the event that any content and context-based constraints are being applied to the data.
As described by [43, 82], the constraints can be represented as rules by using logic. As an example consider a database storing employee information. Attributes for each employee object include Name, SS#, Salary and Department. Department is a reference to another database object with attributes Dept#, Project, Dname and Mgr. Suppose the following security constraints are enforced on this database:

C1. The name attribute of the Emp object where the Salary is greater than 100K is secret.

C2. Name and Salary taken together are secret.

C3. Name where Project in Dept is Stars is secret.

Let Emp(X,Y,Z,W) represent the employee object and attributes and Dept(P,Q,R,S) represent the department object and attributes, and let Emp.Dept(P,Q,R,S) represent the department attribute for each employee object. Then these constraints can be represented in logic form as follows:

R1. \( \text{Emp}(X,Y,Z,W) \land Z > 100K \rightarrow \text{Level}(Y,\text{Secret}) \)

R2. \( \text{Emp}(X,Y,Z,W) \rightarrow \text{Level}([Y,Z],\text{Secret}) \)

R3. \( \text{Emp}(X,Y,Z,W) \land \text{Emp.Dept}(P,Q,R,S) \land Q=\text{Stars} \rightarrow \text{Level}(Y, \text{Secret}) \)

Rules 1 and 3 are content-based constraints and Rule 2 is a context-based constraint. The rules R1 - R3 illustrate how constraints can be represented in a rule base [43].
Environmental information

Many security mechanisms in use with database systems today do not maintain any environmental state information. The mechanisms address the external schemas that describe the views for the database, the conceptual schemas that describe the relations and the attributes, the internal schemas that describe the physical files, and the security and integrity constraints. However there are no provisions for including and maintaining environmental information to discourage user inference.

For example, R2 specifies that the Name and Salary should not be released at the same time. If an unclassified user poses a query for the names, all unclassified user names will be released (according to our example, that would be all names of employees earning 100K or less). Since the user did not ask for salaries also, there is no violation of security constraints. Suppose now that the same user poses a query asking for salaries. Combining the past and current query can lead to the user inferring unauthorized information. Therefore, the information released in the past must somehow be combined with the current query and current constraints in the database to stop this inference from occurring.

If we allow the rule base to be modified dynamically, we can add rules that reflect the fact that certain information has been released to various users, thus storing the environmental information in the rule base as well as the constraints. This is done only when the release of information affects the constraints enforced on the database. For example, R2 can be expressed as follows:


There are now five rules that are part of the rule base to be used when the query is processed. Suppose a query is posed by an unclassified user to obtain all names. All names of employees earning 100K or less will be released, and the following rule will be added to the rule base to record this environmental state information:

\[ \text{R6. } \text{Emp}(X,Y,Z,W) \rightarrow \text{Release}(Y, \text{Unclassified}) \]

As queries for employee names are posed to the database, the system will search the rule base for all rules that classify the names with a sensitivity level higher than the clearance of the user. The premises in this group of rules are conjuncted and the result is the modified query that will be posed to the database in place of the original user query.

This strategy is somewhat conservative. This is best illustrated by the preceding example. When R6 is added to the rule base, it implies that all names were released at the unclassified level. Although this rule is not completely accurate, it is a very efficient method of recording the necessary environmental state information.

Secure Object-Oriented Query Processing

The idea of integrating a rule base into the database works very well in an object-oriented environment. The rule base complements our current security policy very well. Combining the idea of a virtual class as the user view and a rule base to modify this view creates an even more versatile security mechanism. We now describe how these ideas apply in our secure, object-oriented environment.

Recall that the database is represented by a schema graph. From this schema graph we derive a secure query graph, either by finding the query graph and then
applying the security constraints or by applying the security constraints first and then deriving the query graph from this. The rule base can be used in this initial derivation. The user is now associated with a secure query graph. As queries are posed by the user, these queries are evaluated against this graph. As results to queries are returned to the user, the rule base is dynamically updated to reflect this release of information. The secure query graph can also be updated (pruned) as necessary. One large advantage can immediately be seen as compared to the secure query processing proposed in [43]. The strategy in [43] places rules in the rule base that error on the conservative side (i.e., the preceding example which considers a rule implying ALL names have been released at the unclassified level). By pruning the secure query graph carefully, this rule can now be more accurate, while still maintaining the efficiency required by the database.

Dynamically modifying the secure query graph could be time consuming, depending on the application of the database. The use of an object-oriented database system still allows us to maintain a rule base that is more exact than one augmented with a relational database. In both a relational environment and an object-oriented setting, a rule base is stored for the database. However, in a relational setting, a query is directed to various relations (either actual or virtual). Consider the preceding example that placed the rule in the rule base implying all salaries had been released at the unclassified level. This is necessary because the query involves attributes in relations. In our system, queries are directed at classes and instances. Instead of placing a rule in the rule base that says all salaries have been released at the unclassified level, we actually maintain which salaries have been released by saving the object identifier.
As an example, consider the preceding database with employee objects containing Name, SS#, Salary and Department attributes. Assume rules R1 - R5 are enforced on the database. Now let an unclassified user access some of the names in the database he is authorized to see. Only those objects whose names the user has seen will be "released" to the rule base as having their salaries upgraded to secret. So instead of placing a rule in the rule base which states that names have been released at the unclassified level, there will a rule for each object stating that the salary is now secret for that object.

This strategy of placing a rule in the rule base for each released object generates a large rule base. For a method such as this to work, the rules must be stored and maintained in an efficient and organized manner. First, redundant information in any of the rules needs to be factored out, making the method more efficient. Second, the rules must be stored in such a manner that searching for rules that apply to user queries is as efficient as possible.

**Factoring mechanism**

A strategy is presented in [43] which is a variation of the factoring mechanism which extracts the common information present in all the rules. The constraints are represented by using graphs, where nodes represent various relations. We first look at the approach presented by [43]. Then we apply this approach in an object-oriented setting.

Graphs are used to index the security constraints and to make the access more efficient. They are also used to handle each output attribute separately, thus computing the sensitivity level of all the attributes being released by a query. This reduces
the number of rules needed to express a context-based constraint. Each constraint is represented by a graph. The constraint C1 can be represented by the graph shown in Figure 6.5. 

Emp(X,Y,Z,W) is the label of the root node. Since Emp has four attributes, four edges originate at this node. The nodes found at the end of these edges represent the corresponding attributes. The condition $Z > 100K$ is specified by a node that points to Salary(Z). C1 has no affect on any other attributes, so no edges originate at other nodes. To use this graph, we find the edge originating from Name(Y) whose tail is the node labeled Level(Y,Secret) which points to the node with the condition $Z > 100K$. It is this node that classifies the salary.

When an unclassified user makes a query that requests names, this graph will be searched for a node (or nodes) that classifies names at a level greater than or incomparable to the unclassified level. The node Level(Y,Secret) does just this. Now all of the nodes pointed to by Level(Y,Secret) will be checked and the conditions in
those nodes will be negated. This will result in the modified query. Since the node Level(Y,Secret) points to the node $Z > 100K$, the query will be modified to retrieve all names of employees whose salaries are less than or equal to $100K$. The query can be expressed in logic form as: $\text{Emp}(X,Y,Z,W) \land \text{Release}(Y,\text{Unclassified})$. The negation of the condition in the node pointed to by Level(Y,Secret) will be added to the query, resulting in the following modified query: $\text{Emp}(X,Y,Z,W) \land \text{Level}(Z \leq 100K) \land \text{Release}(Y,\text{Unclassified})$.

To handle multiple queries and context-based constraints, additional information has to be incorporated into Figure 6.5 to enforce the constraints. Figure 6.6 illustrates the graph to represent C2. Two nodes have been added to imply that when names are released at the unclassified level, the salaries should not be released and vice versa. When an unclassified user makes a request for salaries after the names have been released at the unclassified level, a search will be performed to find all nodes
that classify salaries at a level greater than or incomparable to the unclassified level. The node Level(Z,Secret) will be found. The condition pointed to by Level(Z,Secret) is Release(Y,Unclassified). Since this procedure is applied at the relational level, the query will be modified to retrieve no salaries since the names have been released.

The query can be modified according to this graph but cannot be evaluated just yet. A search must now be made to determine whether names have been released. Therefore, a graph is necessary to represent environmental information. This graph is shown in Figure 6.7. As soon as any names are released at the unclassified level, this information has to be incorporated in the environmental information of the graph. If an unclassified user has accessed the names, the node Release(Y) must then be appended to the node Name(Y). When the graph is searched for any nodes dealing with the name, the node Release(Y,Unclassified) will be found. Negation of the condition Release(Y,Unclassified) is False. Therefore, the query cannot be modified and posed to the database. The query instead will be converted to the null query and no salaries will be retrieved.
Integrating the rule base into our model

With modifications, the strategy from [43] can be applied in the object-oriented setting as well. A rule base will exist for the entire database schema. This rule base will be searched in the same manner as the rule base existing for relational schemes since it contains the static constraints imposed on the database. Graphs representing the environmental information can be modified to more closely reflect the exact state of the database and to be more specific about what information has been released at various levels.

These ideas can be incorporated into our existing security model to provide mandatory and discretionary security for simple, content-based and context-based constraints. Without the rule base to modify the queries, certain constraints could not be enforced. For example, suppose all entities in the employee database have an associated security classification label. These labels are placed with the entity at the time of the creation of the entity. So if an employee has a salary of 80K, the name is unclassified. However, if the salary is increased (say by some procedure that gives raises at the end of the year), there is no way to guarantee that the security label associated with the name will be changed to secret if the salary is increased to more than 100K. Context-based constraints are clearly not enforced with a static policy such as labels on objects and classes. Therefore, the augmentation of the rule base is a necessity to enforce these three types of constraints.

Let us now illustrate how this rule base fits into the security picture of the existing model. When a user wishes to query a database (say the employee database), a secure view is created for this user. This view can be represented by the secure query graph discussed earlier. The database is therefore minimized so that queries can be posed
to a much smaller subset that the entire schema graph. The user’s secure query graph can internally be represented as a virtual class as discussed earlier in this chapter. The rule base can be consulted to create this initial secure query graph for the user.

As a user poses queries to the database, the queries are modified via the rule base. As information is passed to the user, the release of such information is recorded in the rule base by adding various rules. Every new query must then be modified by the rule base before being posed to the database. Three possibilities exist applying to the secure query graph for the user. Depending on the application, it could be very expensive to “prune” the secure graph every time the user makes a query and the view space has to be reduced (when information is released that can affect the access to other related information). We can choose not to update the secure query graph and only perform query modification as one option. The query will still be modified, so no unauthorized information should be released even though the information exists in the user’s query graph. A second option would be to prune the query graph only when the user makes an additional query to the database involving the pruned information. This type of dynamic pruning would not be as expensive because the pruning would only be performed if it were necessary. The third option would be to prune the query graph to enforce a secure query graph at all times.

**Summary**

This chapter has addressed secure query processing. The basic view mechanism was first reviewed as it applies to multilevel-secure relational database systems. Then a sophisticated view mechanism for object-oriented database systems was reviewed. Modifications of this new view mechanism were then incorporated into our model.
These modifications include the caching of the virtual relation, a definite distinction between attributes and methods, and a differentiation between a view and a query.

The notion of a secure view was presented in the context of a schema graph, a query graph and a secure query graph. A schema graph is a pictorial representation of the entire schema of the database, while a query graph and a view graph are each subsets of this larger graph, representing a domain for the query and the allowable, authorized view for the user respectively. The graphs can be established in two different orders, depending on the applications of the database. The schema graph is always the first, or base graph to be obtained. If many different users with different clearances make the same queries against a database, it is more efficient to first derive the query graph and then the view graph. If one user makes several different queries on the data, it is more efficient to apply the security constraints first and derive the view graph, followed up by obtaining a query graph for each query.

Query modification was presented as a means of enforcing secure query processing. Simple, content-based and context-based constraints were discussed as the constraints to be addressed in a secure environment. To extract the appropriate constraints to perform query modification, we proposed the addition of a rule base to the database system. The rule base not only allows for dynamic modification of the security constraints, but also allows user inference to be controlled by maintaining, as part of the rule base, previously released information that might contribute to the user's ability to infer unauthorized information.

To manage the rule base, constraints and environmental information have been represented by graphs, thus minimizing the number of rules necessary and making searching through the rule base more efficient. Environmental information can also
be represented by using a graph and searched when queries are made. The next chapter summarizes the contributions of this dissertation and presents topics for further research.
CHAPTER 7. CONCLUSION

Overview

We have presented an integrated approach to security in an object-oriented database environment. The four main contributions of this research are:

1. The presentation of a new object-oriented data model.

2. The integration of a security policy into the data model. This policy is not dependent on classes inheriting their security classification labels from superclasses.

3. The description of a transaction model which is used in conjunction with the data model and security policy to provide secure transaction processing.

4. Strategies for secure query processing have been proposed.

We now briefly summarize each on these important contributions.

A major research emphasize at the current time is that of integrating database technology with object-oriented concepts. Object-oriented systems are better suited to handle certain application areas, such as multimedia databases and design databases, because the object-oriented data model allows for constructs to define semantic relationships among objects. It is easily extensible and allows for evolution of the schema.
Object-oriented database systems reduce the semantic gap between the complex applications and the data storage to support these applications. Chapter 1 deals with the necessary concepts and provides a road-map to this dissertation.

Although considerable research efforts have been directed at object-oriented database systems, no standard object-oriented model exists at the present time. Certain “core” concepts represent the commonly accepted and fundamentally important ideas found in areas involved in object-oriented applications. These are discussed in Chapter 2 and include such things as objects and identity, attributes, methods, message passing, classes and hierarchy and inheritance. Related works, such as the relational model and various semantic models are presented to prepare the reader for a deeper discussion of the object-oriented model.

Chapter 3 presents the object-oriented data model in detail. Additional concepts are added to the core concepts presented in Chapter 2. Aggregate objects and version objects are introduced and integrated into the model. Class variables (including aggregate class variables and shared class variables) are defined with several examples given.

Four hierarchies are incorporated into our new data model. With these four hierarchies, the relationships among the database objects can be expressed in a richer and more complete format. The four hierarchies are:

1. Class/Subclass Hierarchy (IS-A). This hierarchy represents the class/subclass structure of the database objects. It is this hierarchy that illustrates the inheritance from class to subclass. The subclass of a class represents a specialization of the class and a superclass of a class denotes a generalization of the class.
Problems inherent in this hierarchy, such as name conflicts, are addressed and resolved in our model.

2. Class/Instance Hierarchy (IS-AN-INSTANCE-OF). This hierarchy illustrates which objects are instances of which classes. Instances also inherit from their classes, so there is a relationship between the class-instance hierarchy and the class/subclass hierarchy. We have separated these two hierarchies because of the two roles that classes perform. First, classes act like templates in defining what characteristics subclasses must have (although subclasses will have additional attributes). The class/subclass hierarchy thus provides a visual display of the database schema. Second, the class acts as a grouping constructor, collecting all like objects (instances). In this sense, the class-instance hierarchy is similar to the concept of a relation in the conventional sense.

3. Aggregate Object Hierarchy (IS-PART-OF). Conventional database models require the objects to be flat in order to be stored in a relational format. Many times, the value of an attribute in an object is another object. In this case, the object references another class. The domain of an attribute in this scenario is actually another class. A component reference further enhances the aggregate object hierarchy by incorporating the concept of the IS-PART-OF relationship.

4. Version Hierarchy (IS-A-VERSION-OF). With the advanced applications possible in an object-oriented environment, we include the concept of versioning as part of our model. Version objects are related to other version via the VERSION-OF relationship or the DERIVED-FROM relationship.
Very clear and concise data model specifications applying to each of the above hierarchies are defined in Chapter 3. Within the framework of these specifications, we show how the new model satisfies the requirements and design objectives in an object-oriented environment.

A database must provide efficient access to the collection of data it stores but often must also guarantee a certain measure of security as well. Chapter 4 presents our security policy. We concentrate on securing the data values by allowing classes to derive security classification labels from their instances and logical instances.

Our policy involves the derivation of several security properties. We have divided these properties into four areas:

1. Security properties for objects. These properties apply to instances of classes and attributes of those instances. We introduce the notion of a negative authorization list for objects.

2. Security properties for classes. These properties apply to classes and address both roles that classes play, namely that of a template and that of a grouping constructor. Properties also address the aggregate and shared class variables introduced as part of our model in Chapter 3.

3. Security properties for methods. Since access to any object in the database must be via message passing that invokes a method, secure execution of all methods is of vital concern. We allow methods to operate within a certain secure range if necessary. We present security properties that all method activations must adhere to.
4. Security properties for versions. Security properties for versionable objects and their versions have been presented. Particular attention is directed at any object related by the “derived-from” relationship.

We integrate the security policy with the data model to develop a formal and precise model of object-oriented database security. We analyze information flow within the data model, including flow from class to subclass, class to instance, object to object, instance to version instance and instance to class. Forward, backward and transitive flow are also addressed.

Using the security properties for objects, classes, methods, and versions we propose a security policy to enforce mandatory and discretionary access controls. To implement the security policy, we present our Method Invocation Algorithm. We then prove that if all entities adhere to our security policy that no information flow violation can occur in our system.

Transactions and concurrency control are also major concerns in a database environment. Chapter 5 deals with object-oriented transaction management and processing. We introduce a new locking algorithm based on conventional hierarchical locking schemes. The Class Hierarchy Locking Protocol addresses locking in the class/subclass and class/instance hierarchies. We introduce a new lock called $R^*$ to allow us to lock a class and all descendants, as opposed to simply locking the class with an $R$ lock. We prove our CLL protocol to be correct.

Next we address locking in the aggregate/composite object and the object version hierarchies. Both the Composite Object Locking Protocol and Version Locking Protocol are shown to be correct.
Secure transaction processing is described using nested transactions. We present a transaction security policy and show that no unauthorized flows of information are allowed if this policy is in effect. Failure and recovery are also addressed, as failure can result from an attempted access to unauthorized information.

Secure query processing is addressed in Chapter 6. We discuss the notion of a secure view and introduce the notion of a secure view graph. Simple, content-based, and context-based security constraints can be handled by our model by augmenting a rule base with the database to perform query modification. User inference is addressed by retaining environmental information as rules in the rule base that may ultimately be used to modify a later query.

**Further Research**

Our model provides a range of security levels for an object and specifies the legal operating range for methods. We allow a method to upgrade its security classification level, but at any given time, a method has only one security level. Possibilities exist to allow for two labels to be associated with a method, thus offering a finer grain of execution depending on whether the method is reading or writing or both. Security labels could then be associated with incoming messages and outgoing information.

Investigation into appropriate query languages or the development of new query languages for object-oriented database systems is a natural extension to this dissertation. The query languages currently used in conventional systems do not take full advantage of the object-oriented environment.

An opening exists for further research on development and maintenance of the augmented rule base. Various query modification techniques should be explored to
take full advantage of any possibility of application in the object-oriented model.

There has been a significant amount of research done in the area of distributed computing, but only a small portion of that research applies to security in such an environment. The ideas presented in this dissertation are applicable in such an environment and research opportunities in this area exist also.

The theory of inheritance and subtyping has been a popular topic in the object-oriented environment, particularly the topics of type theory and type checking. Applying present type checking methods could produce various results in our security model if we viewed security labels as types. This would allow us to monitor the behavior of objects as well as allowing us to perform static and dynamic security checking within the security model itself.

Serializability in object-oriented database systems is not well-defined. Proof mechanisms for traditional serializability (such as acyclic graphs, etc.) may not be sufficient in the object-oriented environment. Research in the area of correctness criteria for object-oriented database serializability needs to be undertaken.
REFERENCES


[33] A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, MA., 1983.


