Proceedings of the IASTED International Conference
Applied Modelling and Simulation
September 1-3, 1999 Cairns, Australia

A Security Policy Manager For Multilevel Secure Object-Oriented Database Management Systems

Ramzi A. Haraty
Lebanese American University
P.O. Box 13-5053
Beirut, Lebanon
rharaty@lau.edu.lb

Abstract

Much attention is being directed toward the development of secure database applications. Such systems are critical for both military as well as sensitive commercial applications. The majority of research in security and multilevel secure database management systems are focused on relational systems. However, with the pervasiveness of complex applications, research in object-oriented security has gained more prominence. In this paper, we describe a secure algorithm for security policy management in multilevel secure object oriented database management systems (MLS/OODBs) based on artificial intelligence techniques.

Keywords: Artificial Intelligence, Object Inheritance, Security Policy, and Topological Sort Order Procedure.

1. Introduction

An object is meant to represent a concept in the real world. Each object belongs (or is an instance of) a single class. A class is viewed as having two parts, a *structure* and a *behavior.* The structure is the instance variables and the methods of the class define its behavior. Classes are encapsulated entities, and the public methods for each class provide the user interface to that class, hiding the implementation details. Classes can be either base or derived. Derived classes inherit from one or more base classes. The set of classes in OODB is organized into a class hierarchy and the schema of each class includes the schema of all of its superclasses.

An object-oriented database is a system which provides all the functionality of a traditional database such as persistence, integrity, transaction management, concurrency control, recovery, query processing and security, as well as object-oriented features such as data abstraction, encapsulation, inheritance, object identity, intelligence, versioning and better performance for complex applications [1].

Data abstraction provides the necessary facilities for incorporating more complex data types such as images, voice segments, vectors, etc. The *Object-Oriented Data Model* provides a better, more powerful, and often more efficient data model. Object-oriented data modeling is closer to real world modeling and therefore, it is more intuitive. Information is modeled in the form of classes and objects that capture the structure and behavior of real world entities. OODB systems maintain unique *Object Identifiers* (OID) for every object. Therefore, eliminating the need for arbitrarily primary keys. This feature solves many integrity issues and speeds up the database access. The access speed is improved due to the reduction of expensive joins on attributes. *Encapsulation* couples the data and its associated operations in an atomic unit. This practice has the benefit of hiding the details of the data from the user. Furthermore, the implementation of the operations (methods) may be modified without invalidating the applications that use them. OODB systems are more *intelligent* than traditional databases. This is mainly due to encapsulation, which gives the database the ability to *reason* about its domain, integrity, validation, and consistency. This awareness in part of the OODB systems triggers appropriate methods to deal with any possible problem. *Versioning* provides the ability to maintain multiple versions of each object allowing design teams to speculate with "what if" scenarios. Better *performance*, is often achieved by applications which need to display complex objects. Such applications can perform two to three orders of magnitude better in an object-oriented environment [1]. This is due to the fact that, it is easier and faster to follow pointers than to join multiple tables.

Inheritance encourages data and code reusability and incremental development. Organizing generalized classes at the top of the hierarchy and deriving specialized classes from them allows us to incrementally augment or extend the database functionality [2]. However, inheritance may get in direct conflict with the security guidelines of the system. For instance, an object may inherit no access default (upon creation of the object) from one superclass and read access only from another. Which default access should the object inherit?

The aim of this paper is to address the questions just posed above. Specifically, it addresses the issue of security when it comes to object inheritance in a MLS/OODB. We will identify a set of security constraints and later describe a security policy manager that interacts with these security constraints in order to achieve a secure and trustworthy MLS/OODB.

This paper is organized as follows: Section 2 presents the security model for MLS/OODBs. Section 3 discusses the security policy manager and its different techniques. Section 4 describes an example of the how security manager works with multiple inheritance. Section 5 contains the conclusion.

2. The Security Model For MLS/OODBs

There are two standard types of security in database systems: *discretionary security* and *mandatory security.* Discretionary security restricts access to data items at the discretion of the owner. Most commercial database management systems employ some form of discretionary security by controlling access privileges and modes of users to data [3]. Discretionary security is not adequate in a multilevel secure environment however, because it does not prevent Trojan horse attacks and provides a low level of assurance. Mandatory security restricts access to data items to cleared database users. It is widely employed in military applications and provides a high level of assurance.

Numerous commercial and military applications require a MLS/OODB. In an MLS/OODB, database users are assigned classifications levels, and data items are assigned sensitivity levels. It is the responsibility of the MLS/OODB to ensure that users can access only those data items for which they have been granted a clearance.

We use the standard military security approach that consists of two components: a set of security classes and a set of non-hierarchical compartments. The security classes are totally ordered from the lowest to the highest as follows: unclassified << confidential << secret << top secret. Within each security class there can be zero or more compartments (for example, conventional, chemical, and nuclear).

We say that a class, S1 dominates another security class, S2, if S1 is hierarchically higher than S2 and contains all of its compartments. We refer to users, or the processes that execute on behalf of users, as subjects. Users are trusted, but processes are not. Objects, on the other hand, correspond to data items. The Bell-LaPadula model defines two security policies commonly accepted in a system that enforces multilevel security [4]:

- The Simple Security Policy: A subject is allowed read access to an object if the subject's classification level is identical to or higher than that of the object's sensitivity level.

- The * - Policy: A subject is allowed write access to an object if the subject's classification level is identical to or lower than that of the object's sensitivity level.

These policies, although important, are not complete for an object-oriented setting. Additional policies are needed to ensure security. These constraints can be summarized in the following policies:

- The Class Security Policy: The sensitivity level of a class must be identical to or lower than the sensitivity level of its subclasses.

- The Instance Security Policy: The sensitivity level of all instances (objects) of a class must be identical to or higher than that of its class.

These policies guarantee that proper access to objects will not be violated directly.

3. The Security Policy Manager

The security policies mentioned above need to be augmented with a trustworthy mechanism that determines for a given subclass of objects what polices and security values to inherit from its superclasses. This is especially complicated when an object inherits from more than one superclass. We propose a security management mechanism that is partially based on the Topological Sorting Procedure [5] to solve this problem. The security manager is part of the Trusted Computing Base (TCB) of the OODB. The TCB is the totality of the protection mechanisms within the OODB, the combination of which is responsible for enforcing a security policy [6]. The security policy manager also determines what new security values to supply for newly created objects.

In OODB the slots in an instance are determined by that instance's superclasses. If a superclass has a slot, then the instance inherits that slot. Sometimes slot values are specified after an instance is created. Alternatively, the slot values of an instance may be specified, somehow, by the classes of which the instance is a member. By writing down, in one place, the knowledge that generally holds for individuals of that class, one can benefit from the characteristics of sharing centrally located knowledge.

One way to accomplish knowledge sharing is to use when-created procedures associated with the classes of which the instance is a member. The expectations established by when-created procedures are called defaults.

In the simplest class hierarchies, no more than one when-created procedure supplies a default value for any particular slot. This is the method used by a variety of OODBs, as currently most of them do not support multiple class inheritance. From a security perspective, this creates no problem as the inheritance mechanism works straightforward. Often, however, several when-created procedures, each specialized to a different class, supply default values for the same slot. How can the OODB decide which when-created procedure to follow? This is complicated by the fact that a given slot may inherit contradictory security defaults, which may lead to insecure access and therefore a violation of the security policy. This is where the security policy manager comes in.

First, the security policy manager learns about the special case in which no class has more than one Is-a link (is a member of the class slot) and no class has more than one Ako link (a kind of slot). Once this foundation is in place, the security policy manager learns about more complicated hierarchies in which class have multiple inheritance links.

One way to decide which when-created procedures to use, albeit a way limited to single-link class hierarchies, is to think of classes themselves as places where procedures can be attached. For example consider the class hierarchy in Figure 1. One of the procedures is attached to A, and the other to B. Because each class is the class hierarchy has only one existing Ako link, it is easy to form an ordered list consisting of D and the classes it belongs to. This ordered list is called the class-precedence list:

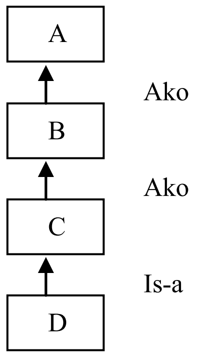

Figure 1. A Simple Class Hierarchy.

<u>*Class-Precedence List:*</u>

D
C
B ← procedure
A ← procedure

So now, when D is created, and according to the class-precedence list, two procedures supply default values for D. Which one to choose from? The security policy manager resolves this conflict in favor of the most specific one – the one that is first on the class-precedence list – i.e., B.

When there is more than one Is-a link above an instance or more than one Ako link above a class, then the class hierarchy is said to branch. Because the class hierarchy branches, the security policy manager must decide how to flatten the class hierarchy into an ordered class-precedence list.

One choice the security policy manager has is to use depth-first search. Depth-first search makes sense because the standard convention is to assume that information from specific classes should override information from more general classes. Left-to-right search makes sense too, but only because the security policy manager needs some way to specify the priority of each direct superclass, and the standard convention is to specify priority through the left-to-right superclass order. However, the security policy manager must modify depth-first search slightly, because the security policy manager wants to include all classes exactly once on the class-precedence list. To perform exhaustive depth-first search, the security policy manager explores all paths, depth first, until each path reaches either a leaf class or a previously encountered class.

To keep a class's superclasses from appearing before that class, the security policy manager modifies the depth-first, left-to-right search by adding the up-to-join procedure. The up-to-join procedure stipulates that any class that is encountered more than once during the depth-first, left-to-right search is ignored until that class is encountered for the last time.

The security policy manager, when employing the depth-first, up-to-join procedure for computing class-precedence lists still leaves something to be desired – the class order on the class-precedence list may change because left-to-right order, by convention, is supposed to indicate priority. The security policy manager is also designed to handle situations where each direct superclass of a given class should appear on class-precedence lists before any other direct superclass that is to its right.

The topological-sorting procedure, that is part of the security policy manager keeps direct superclasses in order on class precedence lists. Thus, the security policy manager knows the order of a class's direct superclasses on the class's class-precedence list as soon as it knows how the direct superclasses are ordered. The security policy manager does not need to know the entire structure of the class hierarchy.

The first step in forming a class-precedence list for an instance using topological sorting is to form an exhaustive list consisting of the instance itself and all classes that can be reached via Is-a and Ako links from that instance. This list constitutes raw material for building the class-precedence list; it is not the class-precedence list itself.

The next step is to form a list of pairs for the one instance and the many classes on the raw-materials list [we refer to both the instance and the classes on the raw-materials list as items].

To form a list of pairs for an item on the raw-materials list, think of passing a fishhook through the item and that item's direct superclasses. Next walk along the fishhook from barb to eyelet while making a list of pairs of adjacent items encountered on the hook.

The next step is to look for an item that occupies the left side of one or more pairs, but does not occupy the right side of any pair. To make it easier to refer to such an item, let us say that it is exposed. Whenever you find an exposed item, the security policy manager adds it to the end of the class-precedence list and strikes out all pairs in which it occurs. However, it may be the case that in the process of building the class-precedence list, two classes get exposed. The security policy manager's tiebreaker is to select the class that is a direct superclass of the lowest precedence class on the emerging class-precedence list. The process is repeated until all the fish hook pairs are eliminated.

4. An Example

Consider the class hierarchy in Figure 2. The raw-materials list, for instance I, contains: I, F, C, B, G, D, H, E, and A. The next step is to form, using the fish hook approach, a list of pairs for the one instance and the many classes on the raw-material list.

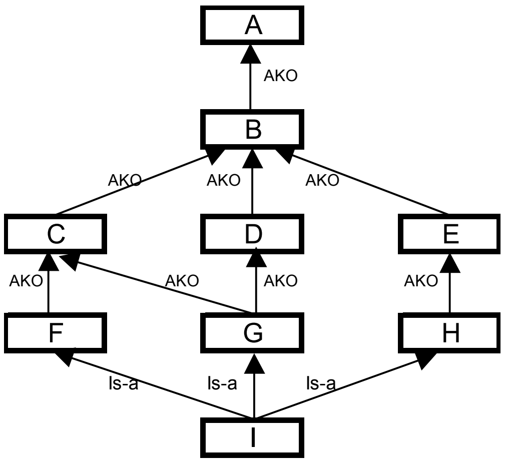

Figure 2 Another Class Hierarchy.

Following the fish hooks for all the items on the raw-materials list results in the following pairs:

Node	*Fish Hook Pairs*
I	I-F, F-G, G-H
F	F-C
C	C-B
B	B-A
G	G-C, C-D
D	D-B
H	H-E
E	E-B
A	A

The next step is to look for an exposed item and add it to the end of the class-precedence list.

Class-Precedence List:

I
F
G ← procedure
C ← procedure
D
H
E
B
A

Suppose two default access procedures when-created procedures were constructed, one for C and the other for G, and suppose that instance I was just created (after the two when-created procedures were constructed). Which default access to follow? The security policy manager would choose G since it appears before C on the class-precedence list.

5. Conclusion

This paper presented security policies specifically developed for MLS/OODBs. These policies are required since the security constraints of the Bell-LaPadula model would not be enough to prevent information flow in violation of the security policy. This paper also presented an algorithm that it directly aimed at resolving conflicts, when it comes to security in inheriting slot values from superclasses. We believe that this algorithm is accurate, flexible, and secure.

References:

[1] Hakimzadeh, H., *Object-Orientation Primer.* (Department of Computer Science Technical Report (NDSU-CSOR-TR-1992-20). North Dakota State University - Fargo, ND. 1992).

[2] Coad, P. and Yourdon, E., *Object-Oriented Analysis.* (Yourdon Press Computing Series, 1991).

[3] Griffiths, P. P. and Wade, B. W., *An Authorization Mechanism for a Relational Database System.* (ACM Transactions on Database Systems, Volume 1, Number 3, 1976).

[4] Bell, D. E., LaPadula, L. J., *Secure Computer System: Unified Exposition and Multics Interpretation.* (Technical Report MTR-2997, The MITRE Corporation. Bedford, MA, March 1976).

[5] Winston, P. H, *Artificial Intelligence*. (Addison-Wesley Publishing. Reading, MA, May 1993).

[6] Department of Defense, *Trusted Computer Systems Evaluation Criteria.* (National Computer Security Center, 1985).

Biography

Ramzi A. Haraty is an Assistant Professor of Computer Science at the Lebanese American University in Beirut, Lebanon. He received his B.S. and M.S. degrees in Computer Science from Minnesota State University - Mankato, Minnesota, and his Ph.D. in Computer Science from North Dakota State University - Fargo, North Dakota. His research interests include database management systems, artificial intelligence, and multilevel secure systems engineering. He has well over 35 journal and conference paper publications.