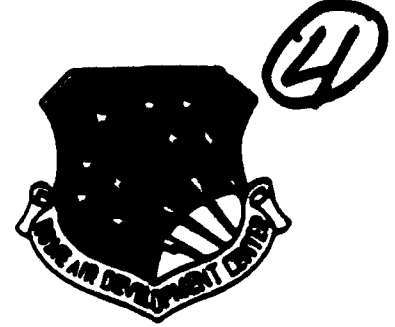


DTIC FILE COPY



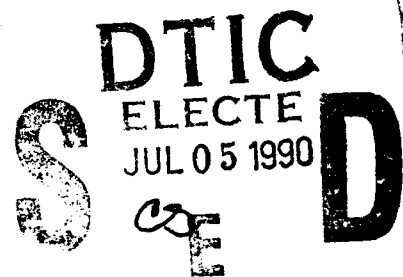
RADC-TR-90-91  
Final Technical Report  
May 1990

AD-A224 491

# INTEGRATING AN OBJECT-ORIENTED DATA MODEL WITH MULTILEVEL SECURITY

George Mason University

Sushil Jajodia, Boris Kogan



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

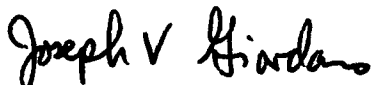
Rome Air Development Center  
Air Force Systems Command  
Griffiss Air Force Base, NY 13441-5700

90 07 5 058

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-91 has been reviewed and is approved for publication.

APPROVED:



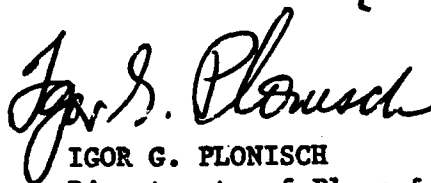
JOSEPH V. GIORDANO  
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.  
Technical Director  
Directorate of Command & Control

FOR THE COMMANDER:



IGOR G. PLONISCH  
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

# REPORT DOCUMENTATION PAGE

Form Approved  
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 1990	3. REPORT TYPE AND DATES COVERED Final Jun 89 to Dec 89	
4. TITLE AND SUBTITLE INTEGRATING AN OBJECT-ORIENTED DATA MODEL WITH MULTILEVEL SECURITY			5. FUNDING NUMBERS C - F30602-88-D-0028, Task B-9-3622 PE - 35167G PR - 1068 TA - 01 WU - P4	
6. AUTHOR(S) Sushil Jajodia, Boris Kogan			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) George Mason University Dept. of Information Sys & Sys Engineering 4400 University Drive Fairfax VA 22030-4444			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-91	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (COTD) Griffiss AFB NY 13441-5700			11. SUPPLEMENTARY NOTES RADC Project Engineer: Joseph V. Giordano/COTD/(315) 330-2925 The University of Dayton is the prime contractor of this effort.	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) A new security model for object-oriented database systems is presented. This model is a departure from the traditional security models based on the passive object - active subject paradigm. This model is a flow model whose main elements are object and messages. An object combines the properties of a passive information repository with those of an active agent. Messages are the main instrument of information flow. The chief advantages of the proposed model are its compatibility with the object-oriented data model and the simplicity with which security policies can be stated and enforced.				
14. SUBJECT TERMS Object-Oriented Trusted Systems			15. NUMBER OF PAGES 44	
Data Base Management Systems Multilevel-Security			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

### ACKNOWLEDGEMENTS

This work was partially supported by the U. S. Air Force, Rome Air Development Center through subcontract # RI-64155X of prime contract # F30602-88-D-0028, Task B-9-3622 with University of Dayton. We are indebted to RADC for making this work possible.

<b>Accession For</b>	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special
<b>A-1</b>	



## 1. Introduction.

Recently, several proposals appeared in the literature dealing with security models for object-oriented databases. While some of them are of considerable interest and merit (see Section 6 for a discussion of related work), they seem to lack in intuitive appeal because they do not appear to model security in a way that would take full advantage of the object-oriented paradigm.

Our goal in the present work is to construct a database security model that dovetails with the object-oriented data model in a natural way. The result, we hope, is a set of principles to help design and implement security policies in a clear and concise fashion.

The object—subject paradigm of Bell and LaPadula [1] is widely used in work on security. An object is understood to be a data file or, at an abstract level, a data item. A subject is an active process that can request access to objects. Every object is assigned a classification, and every subject a clearance. Classifications and clearances are collectively referred to as security levels (or classes). Security levels are partially ordered. A subject is allowed a read access to an object only if the former's clearance is equivalent to or higher (in the partial order) than the latter's classification. A subject is allowed a write access to an object only if the former's clearance is equivalent to or lower than the latter's classification. The above two restrictions are intended to ensure that there is no flow of information from high objects to low subjects. For otherwise (since subjects can represent users<sup>†</sup>) a breach of security occurs wherein a user gets access to information for which he or she has not been cleared.

---

<sup>†</sup> It is a mistake, however, to completely identify subjects with users.

Most security models today are based on the traditional Bell—LaPadula paradigm. While this paradigm has proven to be quite effective for modeling security in operating systems as well as relational databases, it appears somewhat forced when applied to object-oriented systems. The problem is that the notion of object in the object-oriented data model does not correspond to the Bell—LaPadula notion of object. The former combines the properties of a passive information repository, represented by attributes and their values, with the properties of an active agent, represented by methods and their invocations. Thus, the object of the object-oriented data model can be thought of as the object and the subject of the Bell—LaPadula paradigm fused into one.

Continuing the examination of the object-oriented model from the security angle, one arrives at the realization that information flow in this context has a very concrete and natural embodiment in the form of messages. Moreover, taking into account encapsulation, a cardinal property of object-oriented systems, messages can be considered the only instrument of information flow.

The main elements of the proposed model can be sketched out as follows. The system consists of objects (in the new, object-oriented sense). Every object is assigned a unique classification. Objects can communicate (and exchange information) only by means of sending messages among themselves. However, messages are not allowed to flow directly from one object to another. Instead every message is intercepted by the message filter, a system element charged with implementing security policies. The message filter decides, upon examining a given message and the classifications of the sender and receiver, what action is appropriate. It may let the message go through unaltered; or

it may completely reject it (e.g., when a low object sends a message to a high object requesting the value of one of the latter's attributes); or it may take some other action (to be discussed later).

The main advantages of the proposed model are its compatibility with the object-oriented data model and the simplicity and conceptual clarity with which security policies can be stated and enforced.

One comment is in order at this point. Even though all objects are single-level (in the sense of having a unique classification assigned to the entire object and not assigning any classifications to individual attributes or methods), this does *not* preclude the possibility of modeling multilevel entities, as will be demonstrated later.

In this paper, we focus on the problem of mandatory access control. Discretionary access control is not addressed here.

## 2. Object-Oriented Data Model.

An object-oriented database is a collection of *objects* communicating via *messages*. Each object consists of a unique identifier, a set of *attributes* and a set of *methods*, the latter being essentially pieces of code. Each attribute has a *value*, which can change over time.

An object can invoke one of its methods in response to a message received from another object. A method invocation can, in turn, (1) directly access an attribute belonging to the object (read or change its value); (2) invoke other methods belonging to the object; (3) send a message to another object; or (4) create a new object.

There is a special type of object, called *user* object. A user object represents a user session with the system. User objects differ from regular objects in that, in addition to being able to invoke methods in response to messages, they can also invoke methods spontaneously.<sup>†</sup> User object can be created only by the system, at the login time.

Let us formalize now the central elements of the object-oriented data model. We postulate a finite set of domains  $D_1, D_2, \dots, D_n$ . Let  $D$  be the union of the domains augmented with a special element *nil*, i.e.,  $D = D_1 \cup D_2 \cup \dots \cup D_n \cup \{nil\}$ . Every element of  $D$  is referred to as a *primitive object*. Let  $A$  be a set of symbols called *attribute*

---

<sup>†</sup> The notion of spontaneous method invocation may seem rather arbitrary at first. It is, however, necessary in order to avoid running into a version of the chicken-and-egg paradox. Namely, if a message can be sent only through a method invocation (see property (3) of method invocations) and if a method can be activated only by a message received from another object, then how does any processing in such a system ever get initiated? (One has to insist that either the egg or the chicken come first.) In reality, we want a user to be able to initiate a system activity, e.g., by typing a string of characters on the keyboard. This would serve as a signal for the corresponding user object to initiate a method. We choose to think of this as a "spontaneous" initiation, because the keyboard and any signals that it sends are external to our model.



*names*,  $I$  a set of *identifiers*, and  $M$  a set of finite strings of code called *methods*. Let  $V$  be a set of *values* defined as follows:  $V = D \cup I$ . That is, a value is either a primitive object or an identifier.<sup>‡</sup>

**Definition 1.** An *object* is either a primitive object or a quadruple  $o = (i, a, v, \mu)$  such that  $i \in I$ ,  $a = (a_1, \dots, a_k)$  where  $a_j \in A$  for all  $j$  ( $1 \leq j \leq k$ ),  $v = (v_1, \dots, v_k)$  where  $v_j \in V$  for all  $j$  ( $1 \leq j \leq k$ ), and  $\mu \subset M$ .  $\square$

Definition 1 states that an object is defined by its identifier, an ordered set of attribute names, an ordered set of corresponding values, and a set of methods. We assume that every object has a unique identifier, i.e., for any two objects  $o_s = (i_s, a_s, v_s, \mu_s)$  and  $o_t = (i_t, a_t, v_t, \mu_t)$ ,  $o_s = o_t$  iff  $i_s = i_t$ . The uniqueness of object identity is commonly considered a fundamental property of object-oriented systems [6].

We will use the following notation in the rest of the paper. Let  $o_s = (i_s, a_s, v_s, \mu_s)$  be an object. Then  $i(o_s)$  denotes the object identifier,  $i_s$ ;  $a(o_s)$  denotes the list of attributes,  $a_s$ ;  $v(o_s)$  denotes the list of attribute values,  $v_s$ , and  $\mu(o_s)$  denotes the object's set of methods,  $\mu_s$ .

**Definition 2.** A *message* is a triple  $g = (h, p, r)$  where  $h$  is the *message name*,  $p = (p_1, \dots, p_k)$ ,  $k \geq 0$ , is an ordered set (list) of primitive objects or object identifiers called the *message parameters*, and  $r$  is the *return value*.  $\square$

Similarly to the notation used for objects, we let  $h(g)$ ,  $p(g)$ , and  $r(g)$  denote the name, the parameter list, and the return value of message  $g$  respectively.

<sup>‡</sup> A more general object model would also permit a value to be a set of identifiers by defining  $V = D \cup I \cup 2^I$ . However, for the sake of simplicity we forego this generalization in the present paper. The results developed here do not depend on this simplification.

An object sends a message by invoking a system primitive  $SEND(g, i)$  where  $i$  is the identifier of the receiver object. The value  $r(g)$  is computed by the method activated in the receiver upon the arrival of  $g$  there and returned to the sender.<sup>§</sup>

**Definition 3.** The *interface*  $f_o$  of object  $o$  is a function  $f_o: H \rightarrow \mu(o) \cup \{void\}$  where  $H$  is a set of all possible message names.  $\square$

The interface of object  $o$  determines which messages  $o$  responds to. Those are the messages whose names,  $h$ , are such that  $f_o(h) \neq void$ . If  $f_o(h) = void$ ,  $o$  does not respond to messages whose name is  $h$ . Moreover, the interface determines which particular method, out of the set of methods,  $\mu(o)$ , defined for object  $o$ , is to be invoked, depending on the name of the given message.

We have defined methods as strings of code. Now we are in a position to give a more formal definition of methods.

**Definition 4.** Let  $o$  be an object. A *method*  $m$  defined for  $o$  ( $m \in \mu$ ) is a function  $m: P \rightarrow 2^{a(o) \times V} \times 2^{G \times I} \times V$  where  $P$  is a set of all possible parameter lists.  $\square$

Definition 4 states that a method maps a list of parameters into a triple. The first element of the triple is a set (possibly empty) of attribute-name—attribute-value pairs where the names are drawn from the set of the object's attribute names. The second element is a set (possibly empty) of message—identifier pairs. The third element is a value (a primitive object or an object identifier).

---

<sup>§</sup> As we shall see in the next section, sometimes the security component of the system will have to interfere in the matter of computing  $r(g)$ .

In response to a message  $g = (h, p, r)$ , an object  $o$  invokes a method  $m \in \mu(o)$  such that  $m = f_o(h)$  (we assume that  $f_o(h) \neq \text{void}$ ). Then, the value  $m(p)$  is computed (this corresponds to executing the method's code with the argument list  $p$ ). The computation results in  $m(p) = ( \{ (a_1, v_1), \dots, (a_s, v_s) \}, \{ (g_1, i_1), \dots, (g_t, i_t) \}, v_j )$ . The semantics of this are as follows. Attributes  $a_1, \dots, a_s$  of  $o$  are updated with new values  $v_1, \dots, v_s$  respectively; messages  $g_1, \dots, g_t$  are sent to the objects with identifiers  $i_1, \dots, i_t$  respectively; and  $v_j$  is returned to the sender of  $g$ . Note that for some  $k$  we could have  $i_k = i(o)$ , i.e., an object can send a message to itself. This, for instance, can serve as a mechanism for invoking other methods within the same object.

Objects are used to model real-world entities.<sup>†</sup> This is done by associating properties, or facets, of an entity with attributes of the corresponding object. The attribute values are, then, instantiations of those properties. For instance, a country can be represented in a geographic object-oriented database by an object  $o$  where  $a(o) = (\text{COUNTRY\_NAME}, \text{POPULATION}, \text{CAPITAL}, \text{NATIONAL\_FLAG}, \text{FORM\_OF\_GOVERNMENT})$  and  $v(o) = (\text{"Albania"}, 117, i(o_1), i(o_2), i(o_3))$ . The values of the first and second attributes are a string and an integer, respectively; the values of the rest of the attributes are references to other objects that, in turn, describe the capital, the national flag, and the form of government of the nation of Albania.

Note that an object's methods, unlike its attributes, do not have counterparts in the real-world entity modeled by the object. The purpose of methods is quite different. It is to provide support for basic database functionality such as querying and updating objects.

---

<sup>†</sup> As we will see later, a single entity may be modeled by more than one object.

A realistic object-oriented model should also contain the notion of constraints. For instance, an attribute of an object may be allowed to assume values only from a restricted subset of domains or object identifiers. To simplify the exposition, we choose to disregard the issue of constraints in this paper. However, it should be a simple matter to incorporate this notion in our security-data model.

### 3. Object-Oriented Security Model.

An informal exposition of our security model was given in Section 1 in terms of objects with unique security level assignments exchanging messages subject to some security constraints. This section is devoted to developing a formal model of object-oriented security, in accordance with that general idea.

#### 3.1. Security Levels and Information Flow.

The system consists of a set  $O$  of objects (see Definition 1) and a partially ordered set  $S$  of *security levels* with ordering relation  $<$ . A level  $S_i \in S$  is said to be *dominated* by another level  $S_j \in S$ , this being denoted by  $S_i \leq S_j$ , if  $i = j$  or  $S_i < S_j$ . For two levels  $S_i$  and  $S_j$  that are unordered by  $<$ , we write  $S_i <> S_j$ .

There is a total function  $L: O \rightarrow S$ , called *security classification* function, i.e., for every  $o \in O$ ,  $L(o) \in S$ . In other words, every object has a unique security level associated with it.

#### 3.2. Characterization of Information Flows.

The main goal of a security policy must be to control the flow of information among objects. More specifically, information can *legally* flow from an object  $o_j$  to an object  $o_k$  if and only if  $L(o_j) \leq L(o_k)$ . All other information flows are considered *illegal*.

In the Bell-LaPadula model this objective is achieved by prohibiting read-ups and write-downs. That is, a subject is allowed to read an object only if the security level of the subject dominates the security level of the object. Similarly, a subject is allowed to update an object only if the security level of the former is dominated by that of the latter.

In our model, due to the property of encapsulation, information transfer among objects can take place either (1) when a message is passed from one object to another or (2) when a new object is created. In the first case, information can flow in both directions: from the sender to the receiver and back. The *forward* flow is carried through the list of parameters contained in the message, and the *backward* flow through the return value. In the second case, information flows only in the forward direction: from the creating object to the created one, e.g., by means of supplying attribute values for the new object.

A transfer of information does not have to occur every time a message is passed. An object acquires information by changing the values of some of its attributes. Thus, if no such changes occur as a result of a method invocation in response to a message, then no information transfer has been enacted.<sup>†</sup> We say that the forward flow has been *ineffective*. This situation is analogous to taking pictures with an unloaded camera. The information in the form of light is flowing into the camera but not being retained there.

Similarly, if the return value of a message is *nil*, the backward flow has been ineffective. To eliminate the covert channel associated with the receiver object's security level being dynamically changed (in which case the sender can get back a sequence of *nil* and non-*nil* return values if it repeatedly sends messages to the same object), we have to require that all security level assignments be static. That is, the level associated with an object at creation time cannot be changed. If, however, the security level of the real-

---

<sup>†</sup> This is true because an object has no means of registering the very event of a message arrival. Therefore, a covert channel is not possible of the type wherein a message causing a state change encodes a 1 and a message causing none is a 0.

world entity that the object models must be changed, then a new object has to be created. The new object will be exactly like the one that it replaced, except for the new security level to reflect the desired change.

A *transitive* flow from an object  $o_1$  to an object  $o_2$  occurs when there is a flow from  $o_1$  to a third object  $o_3$  and from  $o_3$  to  $o_2$ .

All types of flows discussed until now can be termed *direct* flows. Now, consider what happens when an object  $o_1$  sends a message  $g_1$  to another object  $o_2$ , and  $o_2$  does not change its internal state (the values of its attributes) as a result of receiving  $g$ , but instead sends a message  $g_2$  to a third object  $o_3$ . Further, suppose that  $p(g_2)$  contains some of the parameters of  $p(g_1)$ . If, then, the invocation of  $f_{o_3}(h(g_2))$  with parameters  $p(g_2)$  results in updating  $o_3$ 's state, a transfer of information has taken place from  $o_1$  to  $o_3$ . There is no message exchange between  $o_1$  and  $o_3$ , nor was  $o_3$  created by  $o_1$ , therefore this flow cannot be considered direct. Moreover, there is no flow from  $o_1$  to  $o_3$ , therefore this is not a transitive flow either. This is an instance of what we call an *indirect* flow of information. Note that an indirect flow can involve more than three objects. For example, instead of updating its state,  $o_3$  could send a message to a fourth object that would result in updating the latter's state.

Both direct and indirect illegal flows of information should be prevented (this would also take care of all the transitive flows) if the system is to be secure.

We assume that access to internal attributes, object creation (creation by an object of an instance of itself), and invocation of internal methods are all implemented by having an object send a message to itself.<sup>†</sup> We now define four built-in messages for that

<sup>†</sup> There are existing object-oriented database systems that, in fact, use this kind of implementation, e.g., GemStone.

purpose. First, however, it is necessary to modify the definition of message in Section 2.

**Definition 2a.** A *message* is a triple  $g = (h, p, r)$  where  $h$  is the *message name*,  $p = (p_1, \dots, p_l)$ ,  $l \geq 0$ , is an ordered set (list) of *message parameters* where  $p_j \in V \cup A \cup M \cup S$ , and  $r$  is the *return value*.  $\square$

The difference is that now a parameter can be a method, an attribute name, or a security level as well as a value.

A *read* message is a message sent by an object  $o$  to itself defined as  $g = (READ, (a_j), r)$  where  $a_j \in a(o)$ . A read message results in binding  $r$  to the value of attribute  $a_j$ .

A *write* message is a message sent by an object  $o$  to itself defined as  $g = (WRITE, (a_j, v_j), nil)$  where  $a_j \in a(o)$ . The effect of sending a write message is an update of attribute  $a_j$  with value  $v_j$ .

An *invoke* message is a message sent by an object  $o$  to itself defined as  $g = (INVOKE, (m), r)$  where  $m \in \mu(o)$ . Method  $m$  is invoked as a result of this message.

Finally, a *create* message is defined as  $g = (CREATE, \{v_1, \dots, v_k, S_j\}, r)$  where  $p$  is a list of attribute values appended with a security level. When sent by an object  $o$  to itself, a create message results in a new object being created. This new object is assigned an identifier  $i$  by the system. The object inherits attributes and methods from  $o$ . The attributes are initialized with the values  $v_1, \dots, v_k$ . The new object is assigned security level  $S_j$ , specified by  $o$ . The identifier  $i$  is returned to  $o$  as  $r$ .



### 3.3. Message Filtering Algorithm.

The *message filter* is a security element of the system whose goal is to recognize and prevent illegal information flows. The message filter intercepts every message sent by any object in the system and, based on the security levels of the sender and receiver, as well as some auxiliary information, decides how to handle the message.

In this subsection, we outline an algorithm used by the message filter and show that it indeed prevents all illegal information flows.

Let  $g = (h, p, r)$  be a message. Let  $o_1 = (i_1, a_1, v_1, \mu_1)$  and  $o_2 = (i_2, a_2, v_2, \mu_2)$  be the sender and the receiver objects respectively. Let  $t_1$  be a method invocation in  $o_1$  that was responsible for sending  $g$ . Finally, let  $t_2$  be the invocation of the method  $f_{o_2}(h)$  in  $o_2$  after receiving  $g$ . We assume that every method invocation  $t$  has a *status*  $s(t)$ . The status is either  $U$  (unrestricted) or  $R$  (restricted). The default is  $U$ .

The message filtering algorithm is given in Figure 1. We now argue informally that this algorithm works correctly, i.e., guarantees to prevent any illegal flow of information among objects.

The first part of the algorithm (Case A) deals with message sending between two distinct objects, while the second part (Case B) takes care of the situation when an object sends a message to itself. There are four subcases of Case A, by the number of possible ways in which two security levels — those of objects  $o_1$  and  $o_2$  — can be related. Also, there are four subcases (with yet finer subdivisions) of Case B, by the number of built-in messages.

Subcases (2)—(4) prevent direct illegal information flows between two distinct existing objects. When the two security levels are unrelated, this is done by blocking the message completely (subcase 2). When the sender's level is strictly lower than the receiver's level, the return value is set to *nil* by the message filter to prevent any leakage of information from the receiver to the sender (subcase 3). Finally, when the sender is at a higher security level, the invocation of the method (in the receiver) in response to the message is given the restricted status by the message filter (subcase 4). This would prevent the receiver from making updates to its local attributes (see subcase 1.b of Case B) thereby retaining sensitive information that could have been extracted from the message.

Direct illegal flow during the creation of a new object is prevented by letting the create message pass only if the security level specified for the *new object dominates* that of the creator object (subcase B.3).

It remains to show now that indirect illegal flows are also prevented by our algorithm. To understand how this is done, it is helpful to think of *trees* of method invocations. Such a tree is shown in Figure 2. Let  $t_o$  be the root.  $t_o$  is a "spontaneous" method invocation within a user object. Let  $t_i$  be an internal node in this tree. Then  $t_i$ 's children are the method invocations that were initiated either within the same object directly by  $t_i$  or in another object as a result of receiving a message sent by  $t_i$ . Thus, the entire tree can be viewed as the execution of a user request.

Information can be carried down a method invocation tree along the parent—child links. Let  $t_1$  and  $t_2$  be two internal nodes. Suppose that  $t_1$  is an invocation of a method

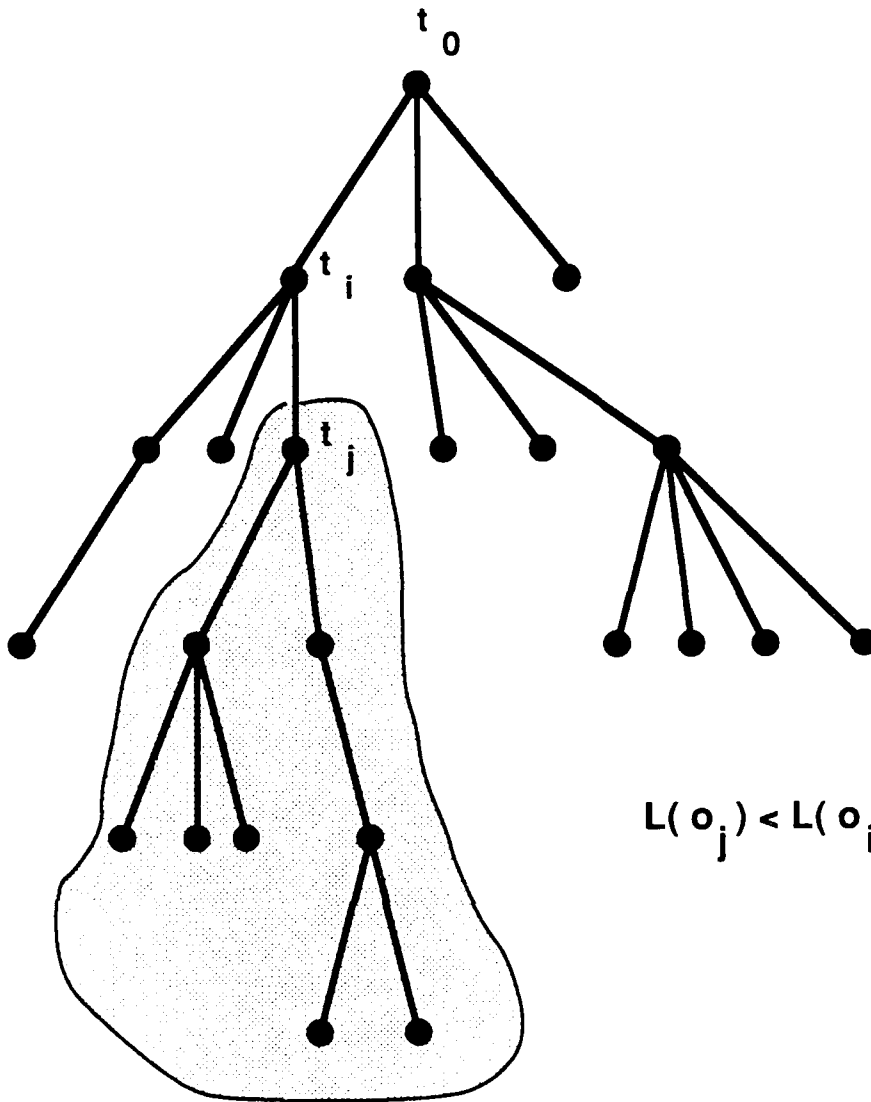
**CASE A:**  $o_1 \neq o_2$

- (1) **if**  $L(o_1) = L(o_2)$   
    **let**  $g$  **pass**;  $s(t_2) \leftarrow s(t_1)$
- (2) **if**  $L(o_1) \langle \rangle L(o_2)$   
    **block**  $g$
- (3) **if**  $L(o_1) < L(o_2)$   
    **let**  $g$  **pass**;  $r \leftarrow nil$ ;  $s(t_2) \leftarrow s(t_1)$
- (4) **if**  $L(o_2) < L(o_1)$   
    **let**  $g$  **pass**;  $s(t_2) \leftarrow R$

**CASE B:**  $o_1 = o_2$

- (1) **if**  $h = WRITE$ 
  - (1.a) **if**  $s(t_1) = U$   
    **let**  $g$  **pass**
  - (1.b) **if**  $s(t_1) = R$   
    **block**  $g$
- (2) **if**  $h = READ$   
    **let**  $g$  **pass**
- (3) **if**  $g = (CREATE, \{v_1, \dots, v_k, S_j\}, r)$ 
  - (3.a) **if**  $s(t_1) = U$  **and**  $(S_j < L(o_1) \text{ or } S_j \langle \rangle L(o_1))$   
    **block**  $g$ ;
  - (3.b) **if**  $s(t_1) = R$   
    **block**  $g$
  - (3.c) **if**  $s(t_1) = U$  **and**  $L(o_1) \leq S_j$   
    **let**  $g$  **pass**
- (4) **if**  $h = INVOKE$   
    **let**  $g$  **pass**;  $s(t_2) \leftarrow s(t_1)$

**Figure 1. The Message Filtering Algorithm.**



RESTRICTED

$$L(o_j) < L(o_i) \Rightarrow s(t) \leftarrow R$$

Figure 2. Method-Invokation Tree

in object  $o_1$ ,  $t_2$  in object  $o_2$  such that  $L(o_2) < L(o_1)$ , and  $t_2$  is a child of  $t_1$ . Then, by making  $t_2$  restricted (as in A.4) we cut off forward direct flow from  $o_1$  to  $o_2$ . If we also want to prevent indirect illegal flows from  $o_1$  to objects other than  $o_2$ , as a result of the same message from  $o_1$  to  $o_2$ , we have to make sure that all ancestors of  $t_2$  are made restricted as well. This is taken care of in subcases A.1, A.3, A.4, and B.4.

The general idea of a message filter is similar to that of a law filter introduced by Minsky and Rozenshtein [11], although their work has no direct relation to security.

In the standard kernelized architecture, the message filter does not have to be made trusted, because the actual mandatory access control is delegated to the trusted kernel that includes a reference monitor. The kernel is part of the operating system, which is based on the traditional read/write primitives to implement data access. However, newly emerging operating systems based on message-sending primitives may radically change the place of the message filter in the security structure of the overall system. It seems that, under such conditions, it would be quite natural to make the filter part of the kernel, thus substituting it for the reference monitor in this new architecture. This would obviously necessitate making the message filter trusted. In our future work, we plan to study this alternative together with its implications.

#### 4. Class Hierarchy and Security.

The notion of *classes* is usually considered very important for object-oriented databases, if not for object-oriented systems in general. Most existing object-oriented databases support classes.

The notion of classes is akin to that of relations in relational databases. Objects of similar structure (types and names of attributes) and similar behavior (methods) are grouped into classes, just like, tuples of the same structure, in relational databases, are grouped into relations. The parallel to relational systems does not go very far, however. First, in relational databases, there is no notion analogous to that of object behavior. Second, classes in object-oriented databases are represented by objects that contain information (in the form of attributes) on the names and types of attributes of the constituent *instance* objects of the class as well as the methods common to them. Objects of this kind are called *class-defining* objects, or simply class objects. Thus, there is essentially no distinction in representation of data and metadata in object-oriented systems.

We assume that the reader has a basic familiarity with the notions of *inheritance* and class *hierarchy*. A typical class hierarchy has a class OBJECT at its root. It also includes a special class CLASS such that every object defining a class is an instance of CLASS.

In Section 3.2 we discussed ways by which objects can transfer information to one another. Message sending and object creation were mentioned in this connection. We, then, went on to define several types of information flow. With the introduction of classes and inheritance, two more (implicit) ways of information transfer are added.

Since a class object (a class-defining object) contains structure and behavior information for all its instance objects, the latter have an implicit read access to the former. Thus, an information flow exists from a class object to an instance object. We refer to this type of flow as a *class—instance* flow.

Since classes inherit attributes and methods from their ancestors in the class hierarchy, a class object has an implicit read access to all its ancestors. Therefore, there is an information flow down along all hierarchy links. This type of flow is designated *inheritance* flow.

It is easy to see that an inheritance flow is illegal unless the level of a class object dominates the level of each of its ancestors. Similarly, a class—instance flow is illegal unless the level of an instance object dominates that of its class.

Our approach to dealing with illegal inheritance and class—instance flows is to implement the classification and inheritance features by means of message passing. For the details of such an implementation see [11]. The purpose achieved by this approach is to make the *implicit* flows discussed above *explicit*, i.e., realized by means of messages. As a consequence, class—instance and inheritance flows can be checked by the message filter, just like forward, backward, and indirect flows are.

It is still a good idea, though, to place the following constraints on the way the security levels of instance objects and subclasses objects relate to those of the corresponding class objects.

#### **Security-Level Constraint 1.**

If  $o_j$  is an object of class  $c_j$  ( $c_j$  also denotes the corresponding class object),  
then  $L(c_j) \leq L(o_j)$ .

### Security-Level Constraint 2.

If  $c_i$  and  $c_j$  are classes such that  $c_j$  is a child of  $c_i$  in the class hierarchy, then  
 $L(c_i) \leq L(c_j)$ .

It is important to understand that the above constraints are *not* introduced for security reasons — security is still handled by the message filtering algorithm because all flows, including the class—instance and inheritance flows, are explicitly cast in the form of messages — and therefore, a violation of these constraints will not lead to a violation of security. Instead, a violation of Constraint 2, for example, will result in the breakdown of inheritance mechanism by creating a situation wherein a class object is prevented by the message filter from gaining access to a method it inherits from its parent class, because the security level of the child does not dominate that of the parent, as required by the constraint.

Note that Constraint 1 is automatically satisfied by the message-filtering algorithm (see subcase B.3) at the instance-creation time. It is interesting to note, though, that this feature was originally included in the algorithm to prevent the illegal direct flow to the newly created object at the creation time rather than the illegal class—instance flow, which can take place at any time after the instance is created. However, the provision works equally well in both cases.

Constraint 2 is not automatically satisfied by the message filtering algorithm, but the latter could be modified for that purpose. Alternatively, the constraint could be enforced



by supplying the object CLASS with a method for creation of new classes that would include provisions for checking that the security level of the new class is in the prescribed relationship to the levels of its parents.

## 5. Modeling Multilevel Entities with Single-Level Objects.

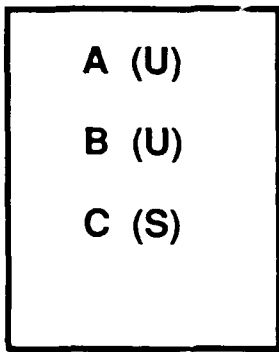
In an object-oriented data model, objects are used to model real-world entities. Therefore, it may seem somewhat discouraging that our security model insists that all objects have to be “flat,” i.e., at a single security level. Much modeling flexibility would be lost if multilevel entities could not be represented in our database.

In this section we will demonstrate that restricting objects to be single-level does not have to imply that the same type of restriction exists for entities that we are trying to model. We will do so by means of a simple example.

Suppose that there are two security levels:  $U$  (unclassified) and  $S$  (secret), the latter dominating the former. Consider an entity  $e$  characterized by attributes  $A$ ,  $B$ , and  $C$  such that  $A$  and  $B$  are at level  $U$  and  $C$  is at level  $S$ . ( $e$  could be a collection of information pertaining to an employee where  $A$  is the employee’s name,  $B$  is the home address, and  $C$  is the salary.) The intention is to allow access to  $C$  only for users with secret clearance. All other users can access only  $A$  and  $B$ . Entity  $e$  can be represented by objects  $o_1$  and  $o_2$  such that  $a(o_1) = (A, B)$ ,  $a(o_2) = (A, B, C)$ ,  $L(o_1) = U$ , and  $L(o_2) = S$ . Object  $o_2$  is the internal representation of entity  $e$  for users with the secret clearance, while  $o_1$  is the representation of  $e$  for all other users. The example is illustrated in Figure 3. Attributes of entity  $e$  have individual security labels (shown in parentheses). This is in contrast to objects  $o_1$  and  $o_2$ , which only have labels at the object level.

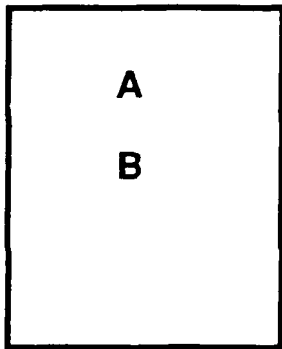
Suppose now that we have an entire collection of entities of the same type as  $e$  (e.g., a set of employee entities), i.e., entities with unclassified attributes  $A$  and  $B$  and a secret attribute  $C$ . Let us call this type of entity  $X$ . In our model each entity of this type will be

e



---

$o_1$  (U)



$o_2$  (S)

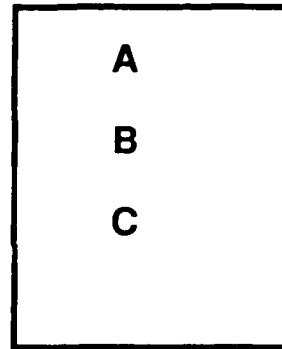


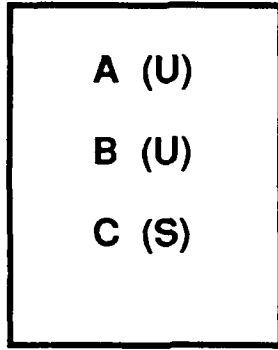
Figure 3. Representing a multilevel entity by multiple single-level objects

represented by two objects: one for the users with the secret clearance and one for all others. Thus, we end up with two classes of objects for one type of entity. The distinction between the two classes is based on security, not semantics, as would normally be the case in object-oriented databases. Let  $XU$  be the class of the unclassified objects and  $XS$  the class of the secret objects representing entities of type  $X$ .

It is convenient, for modeling purposes, to relate classes  $XU$  and  $XS$  in the class hierarchy. Namely, if  $XS$  is made a child of  $XU$ , then it can inherit from  $XU$  attributes  $A$  and  $B$  and add to them a locally defined attribute  $C$ . Figure 4 shows the relevant segment of the hierarchy. Note that the class object  $XU$  is placed at security level  $U$ , and  $XS$  at level  $S$ . The effect of this is that not only do the unclassified users have no access to the values of attribute  $C$  in entities of type  $X$ , but also they are not even aware of the existence of this secret attribute because access to the class object  $XS$  is prohibited to them. It is possible to place the class object  $XS$  at level  $U$ , while keeping instances of  $XS$  at level  $S$ . In that case, the unclassified users will be aware of the existence of attribute  $C$  but not of any values of it in instance objects. Note that such a dichotomy between the class-object level and the level of its instances is in conformity with Security-Level Constraint 1. The choice of label for  $XS$  depends on the policy decision.

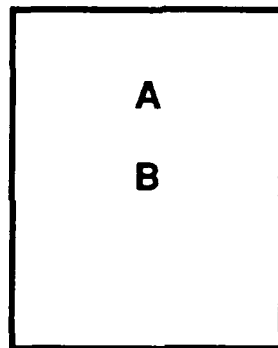
To carry our example a little further, suppose that there is a second type of entity that we have to model, type  $Y$ .  $Y$  consists of the same attributes at the same security levels as  $X$  plus a new attribute  $D$  at level  $U$ . The *conceptual* class hierarchy (or schema) is shown in Figure 5. In that schema,  $Y$  is a child of  $X$ .

X



---

XU (U)



XS (S)

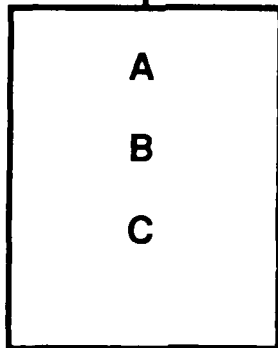
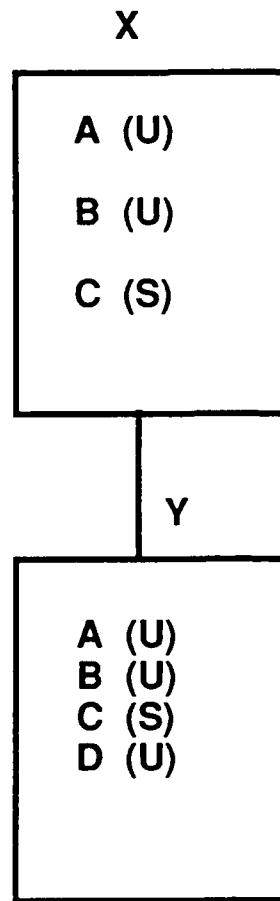


Figure 4. Representing a type of multilevel entities by a hierarchy of classes of single-level objects



**Figure 5. Conceptual schema for types X and Y**

Let us now address the question of how this schema can be implemented in our model. Using the idea of Figure 4, we arrive at the *implementation* schema for our database, shown in Figure 6. The implementation schema takes into account security level assignments to attributes in the conceptual schema and transforms the latter into the form ready for actual implementation in a system that uses our security paradigm. In particular, we have four classes in our implementation schema:  $XU$ ,  $XS$ ,  $YU$ , and  $YS$ .  $XU$  represents the view of  $X$  for uncleared users;  $XS$ , the view of  $X$  for users with the secret clearance;  $YU$ , the view of  $Y$  for uncleared users; and  $YS$ , the view of  $Y$  for users with the secret clearance.

In Figure 6, links between classes represent inheritance relationships among classes. It is helpful to distinguish between two kinds of inheritance in the implementation schema: *semantic* inheritance and *security* inheritance. The actual inheritance mechanism is identical in both cases, but the motivation is different. Semantic inheritance corresponds to the usual notion of inheritance in object-oriented databases. It is intended to represent the semantic relationships among data types found in the conceptual schema. The notion of security inheritance, on the other hand, is introduced solely for the purpose of representing multilevel entities in our security paradigm. Thus, for instance,  $YU$  is a subclass of  $XU$  in the semantic sense because this relationship reflects the specialization of the entity type  $X$  into  $Y$  by adding to the former a new attribute  $D$ . On the other hand,  $XS$  is a subclass of  $XU$  in the security sense because  $XS$  reveals a new attribute of entities of type  $X$  that is not visible to uncleared users. Note that the notion of security inheritance is in agreement with Security-Level Constraint 2, which requires that the security

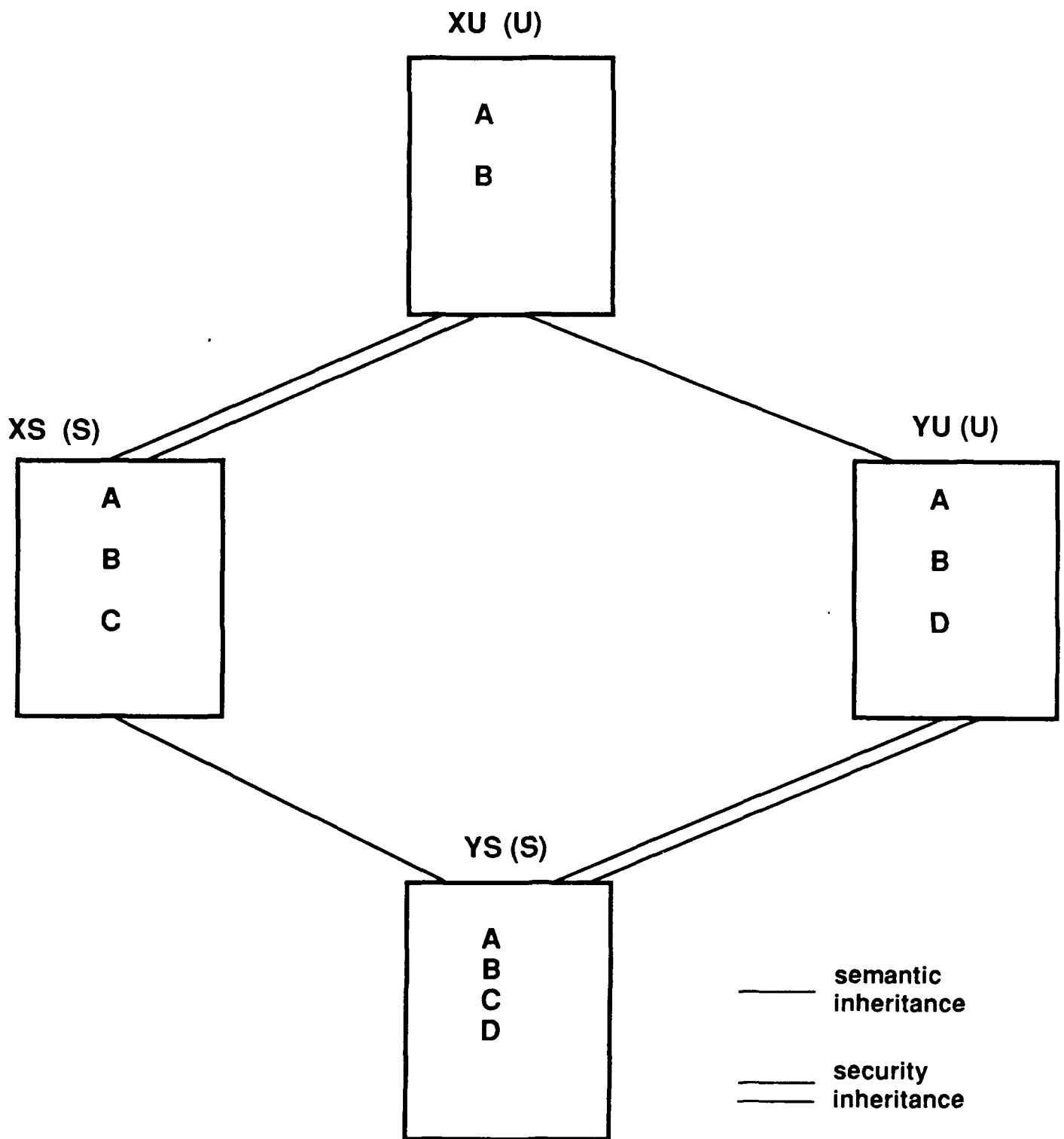


Figure 6. Implementation Schema



level of a class dominate that of its ancestors.

Instance objects, as was discussed earlier in this section, do not have to be at the same security level with their class object. By the same token, instance objects may sometimes be placed at different levels with one another, just like it may be required that real-world entities of the same type have different security classifications. Our model does not prohibit this, although it would not make sense of course (but not violate security) to have some instance objects at levels higher than the corresponding class object.

## 6. Review of Relevant Research

Object-oriented approach has been a major area of research in the context of programming languages, knowledge representation, and databases for some years now (e.g., see [7, 13, 17]). In spite of this, there has been relatively little work on security related issues in the object-oriented databases, although some work does exist. Initial efforts in [2, 3, 12] handle only the discretionary access controls. Meadows and Landwehr [10] are the first to model mandatory access controls using object-oriented approach, however, their effort is limited to considering the Military Message System. Spooner in [14] takes a preliminary look at the mandatory access control and raises several important concerns. In [4, 5, 8, 15, 16], objects can be multilevel. This means, for example, that an object's attributes can belong to different security levels, which in turn means that the security system must monitor all methods within an object. As we have argued in the introduction, we consider it to be contrary to the spirit of the object-oriented paradigm.

Finally, Lunt and Millen in [9] mention some problems associated with having multilevel objects. In their model, only single-level objects are permitted; however, the notion of subjects is still retained, and subjects are assigned security levels.

## 7. Conclusions and Future Work.

An examination of the object-oriented data model leads one to believe that there is much in it, particularly in the notion of encapsulation, that makes this model naturally compatible with the notion of security. However, until now, relatively little use has been made of this apparent compatibility.

This paper is part of an effort to develop a better understanding of the interactions between multilevel security and the object-oriented data model. This interaction, in our opinion, can be very subtle, and for that reason, we chose a formal approach. We wanted to state precisely all critical assumptions, which are necessary if we hope to use this paper as a departure point for further research.

We believe that there is much more interesting work to be done in the area of *object-oriented multilevel security*. In particular, we would like to be able to construct a complete and unified formal model of data and security based on the object—message paradigm. Initial steps have been taken in this direction in the present paper, but more remains to be done. For example, we would like to formalize the notion of classes and inheritance in a way adaptable to security concerns.

In Section 5, we presented some ideas for representing multilevel entities using multiple objects at different security levels. These ideas were illustrated by an example. The subject clearly merits further study, and perhaps one should address the issue of designing an algorithm for multi-object representation of multilevel entities.

The issue of polyinstantiation in the context of object-oriented databases has not been discussed here. We plan to study this in the future to see how this relates to

polyinstantiation in relational databases.

Implementing the class and inheritance mechanisms by message passing is essential to our approach to enforcing security. In a system that follows such an implementation, all information flows are rendered explicit, and therefore controllable uniformly by the message filter. Consequently, our future work should address this issue of implementation, as it relates to modeling security, at a more detailed level.

Finally, an additional research direction of significant interest is the study of the potential benefits of message-based operating systems to implementation of the proposed security model.

## References

1. D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Unified Exposition and Multics Interpretation," The Mitre Corp., March 1976.
2. Klaus R. Dittrich, Martin Hartig, and Heribert Pfefferle, "Discretionary Access Control in Structurally Object-Oriented Database Systems," in *Database Security, II: Status and Prospects*, ed. Carl E. Landwehr, pp. 105-121, North-Holland, Amsterdam, 1989.
3. Eduardo B. Fernandez, Ehud Gudes, and Haiyan Song, "A Security Model for Object-Oriented Databases," *Proc. IEEE Symposium on Security and Privacy*, pp. 110-115, May 1989.
4. T. F. Keefe, W. T. Tsai, and M. B. Thuraisingham, "A Multilevel Security Model for Object-Oriented System," *Proc. 11th National Computer Security Conference*, pp. 1-9, October 1988.
5. T. F. Keefe and W. T. Tsai, "Prototyping SODA Security Model," *Proc. 3rd IFIP WG 11.3 Workshop on Database Security*, September 1989.
6. Setrag N. Khoshafian and George P. Copeland, "Object Identity," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, September 1986.
7. Won Kim and Frederick H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, Mass., 1989.
8. Teresa F. Lunt, "Multilevel Security for Object-Oriented Database System," *Proc.*

9. Teresa F. Lunt and Jonathan K. Millen, "Secure Knowledge-Based Systems." Interim Technical Report, Computer Science Laboratory, SRI International, August 1989.
10. Catherine Meadows and Carl Landwehr, "Designing a Trusted Application in an Object-Oriented Data Model," in *Research Directions in Database Security*, ed. Teresa Lunt, Springer-Verlag, Berlin, To appear.
11. Naftaly H. Minsky and David Rozenshtein, "A Law-Based Approach to Object-Oriented Programming," *Proc. Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 482-493, October 1987.
12. Fausto Rabitti, Darrell Woelk, and Won Kim, "A Model of Authorization for Object-Oriented and Semantic Databases," *Proc. Int'l. Conf. on Extending Database Technology*, pp. 231-250, March 1988.
13. Bruce Shriver and Peter Wegner, eds., *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, 1987.
14. David L. Spooner, "The Impact of Inheritance on Security in Object-Oriented Database Systems," in *Database Security, II: Status and Prospects*, ed. Carl E. Landwehr, pp. 141-160, North-Holland, Amsterdam, 1989.
15. M. B. Thuraisingham, "A Multilevel Secure Object-Oriented Data Model," *Proc. 12th National Computer Security Conference*, pp. 57-65, October 1989.
16. M. B. Thuraisingham, "Mandatory Security in Object-Oriented Database System,"

*Proc. Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 203-210.

17. Stanley B. Zdonik and David Maier, eds., *Readings in Object-Oriented Database Systems*, Morgan Kaufman, San Mateo, Calif., 1990.



*MISSION*  
*of*  
*Rome Air Development Center*

*RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C<sup>3</sup>I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C<sup>3</sup>I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.*