An Audit Model for Object-Oriented Databases

Boris Kogan
Sushil Jajodia

Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444

ABSTRACT

Auditing capability is one of the requirements for secure databases. A secure database management system, among other things, has to provide not only facilities for recording the history of all updates and queries against the database but high-level support for querying this history as well. In this paper, we present an audit model for object-oriented databases that satisfies both requirements. Our model offers several additional advantages: (1) it imposes a uniform logical structure upon both the current and the audit data; (2) it results in *zero-information loss*, i.e., there is never any loss of historical or current information in this model; and (3) since it captures the entire database activity, a complete reconstruction of every action taken on the database is possible. We show how this third aspect can be exploited to provide high-level support for expressing audit and other database queries, and therefore, we make a complete audit trail methodology available.

1. INTRODUCTION

In situations where the data is sufficiently sensitive, an audit trail becomes a necessity. Unfortunately, existing databases make a distinction between the current and past data. A logical structure exists only upon the *current* data to allow its retrieval through a query language; any old information is either deleted or stored in the form of a *log*. Such an "audit trail" has an *ad hoc* structure, and generally, cannot be queried. In this paper, we present an audit model for object-oriented databases that imposes a uniform logical structure on both the current and the audit data. It provides facilities for both recording the history of updates and queries as well as querying this history.

The primary distinguishing components of our model are (1) *auditable data objects*, which differ from regular data objects in that they encapsulate history information that enables audit operations to be performed on them, (2) *user objects*, which capture all information related to the users, and (3) *transaction objects*, which are a special type of objects that capture users' activity in an auditable form. There is never any loss of historical or current information in this model, thus it is possible to reconstruct every action taken on the database. Since this latter aspect can easily be exploited to provide high-level support for expressing audit and other database queries, we make a complete audit trail methodology available.

The organization of this paper is as follows. We begin in Section 2 by listing the auditing requirements for databases. In Section 3, we introduce our basic object model, which is an abstraction that does not take into account possible auditing needs of a database system. In Section 4, we enhance this basic model by adding to it the elements needed to facilitate auditability. In Section 5, we show how our model can be used to provide high-level support for manipulating the audit data. Finally, Section 6 concludes the paper.

2. AUDIT REQUIREMENTS

Auditing has long been recognized as being a critical element in secure systems. Using the audit data, it is possible to reconstruct events in a system. Therefore, auditing can be used to provide accountability for all user actions, including those affecting security. This latter capability can also serve as a deterrent to improper behavior by unauthorized as well as authorized users.

The ability to reconstruct events has different meaning to different people. Most of the research on auditing has dealt with *system* auditing that can be used for the purpose of intrusion detection and prevention (see [3,6,9], for example). This is in contrast to *database* auditing whose primary purpose is attributability of updates and queries [2,4,7].

0-8186-2280-6/91 $1.00©1991 IEEE

In the database context, an audit trail is a mechanism for a complete reconstruction of every action taken against the database: *who* has been accessing *what* data, *when*, and in what *order*. Thus, database auditing has three basic objects of interest:

- The user — who initiated an activity in the database, when, etc.
- The transaction — what was the exact transaction that was initiated.
- The data — what was the result of the transaction, what were the database states before and after the transaction initiation.

In addition to the actual recording of all events that take place in the database, an audit trail must also provide high-level support for querying the audit data, i.e., an audit trail must have the capability "for an authorized and competent agent to access and evaluate accountability information by a secure means, within a reasonable amount of time and without undue difficulty" [1].

Existing databases clearly fail to meet the above requirements. The central point of this paper is to propose a new data model for object-oriented databases which meets these objectives by providing a convenient mechanism for both recording and querying accesses to the database.

3. BASIC CONCEPTS

The reader should be familiar with the basic concepts of object-oriented approach (e.g., see [5, 8, 10]), although almost all relevant notions are described below.

The internal view of an *object* can be defined as a triple (o, A, M) where o is a unique object identifier, A is a set of *instance variables*, and M is a set of *methods*. Every instance variable is a pair $(A_i, V(A_i))$, where A_i is an instance-variable name, and $V(A_i)$ (the value of A_i) is either a primitive object (e.g., integer or string of characters) or an object identifier. Similarly, every method is a pair $(M_i, P(M_i))$, where M_i is a method name, and $P(M_i)$ is a program. We will refer to instance variables and methods by their names rather than by name-value (or name-program) pairs.

Objects can send and receive *messages*. A message $\mu = (m, l)$ is a pair consisting of a message name m and a parameter list $l = (p_1, ..., p_k)$, where $p_1, ..., p_k$ are message parameters. Note that the number of parameters, k, can be zero, in which case l is empty.

An object responds to a message $\mu = (m, l)$ by invoking the corresponding internal method. For the sake of simplicity we assume that every message has the same name as the method to which it corresponds. For example, suppose that objects of a class QUEUE respond to a message DEQUEUE. Then the method that implements such a response will also be called DEQUEUE. Therefore, the correct response to μ is finding the method of the same name and executing its program, i.e., executing $P(M_i)$ such that $M_i = m$ using the parameters in l. If such M_i cannot be found among the methods defined for the object, the object does not respond to the message. The execution of m is completed by sending a *return value* (a primitive object or an object identifier) back to the object from which μ came. The process of sending a message to an object and receiving a return value upon the due completion of the required method execution will be referred to as an *operation* on the object.

The external view of an object is (o, F), where $F = \{M_1, ..., M_n\}$ is the object's *interface*. F contains the names of all the messages to which the object responds (or, equivalently, the names of al the methods defined for the object). Thus, the interface defines the operations available on the object. The external view of an object is derived from its internal view by pruning out all instance variables (both names and values) as well as programs used for methods. The dichotomy between the internal and external views is significant in that it reflects the intent to hide the internal structure of the object and the implementation of its methods. Thus, only the interface is visible from the outside. Note that this is consistent with the traditional notion of objects as abstract data types.

4. THE AUDIT MODEL

In this section, we will enhance the basic object-oriented data model to facilitate auditability. The enhanced model requires three kinds of objects:

- *Auditable data objects* — These objects encapsulate the history of all changes to their own states.
- *User objects* — These objects represent database users and contain all relevant information associated with the users.
- *Transaction objects* — These objects are used to record the user activities.

We describe each of these in turn next.

4.1. Auditable Data Objects

In Section 3, we associated a value with every instance variable of an object. However, when auditing needs are taken into account, it is more convenient to associate an instance variable A_i with a *set* of values

rather than a single value. Moreover, each value in the set is labeled with the identity of the user responsible for the creation of this value, as well as the time the user originated the respective request. Thus, in the new definition,

$$V(A_i) = \{ (v_i^1, u_i^1, t_i^1), ..., (v_i^r, u_i^r, t_i^r) \}.$$

The set of triples is ordered by timestamps t_i^j. $V(A_i)$ is intended to represent the entire history of the instance variable A_i rather than just its current value.

We define the current value of an instance variable A_i to be the value associated with the largest timestamp of all the triples in $V(A_i)$. Let $cur(A_i)$ denote the current value of A_i. Then

$$cur(A_i) = v_i^m \text{ s.t. } t_i^m = \max_{1 \le j \le r}(t_i^j).$$

But since $V(A_i)$ is an ordered set, we can simply state that $cur(A_i) = v_i^r$, where v_i^r is taken from the last triple of $V(A_i)$.

When an operation on an object results in updating an instance variable, the update is carried out in our model not by deleting the old value and inserting the new one, but rather by adding a new triple $(v_i^{r+1}, u_i^{r+1}, t_i^{r+1})$ to $V(A_i)$.

A method's code, just as an instance variable's value, can be changed over time. To preserve complete auditability, $P(M_i)$, similarly to $V(A_i)$, records the history of such changes. Thus,

$$P(M_i) = \{ (g_i^1, u_i^1, t_i^1), ..., (g_i^s, u_i^s, t_i^s) \}.$$

Like $V(A_i)$, $P(M_i)$ is ordered by timestamps t_i^j. For every j, $1 \le j \le s-1$, g_i^j is the program associated with M_i from time t_i^j until t_i^j, when it is replaced with g_i^{j+1} by user u_i^{j+1}. The current program for M_i is denoted $cur(M_i)$. Note that $cur(M_i) = g_i^s$.

4.2. User Objects

We distinguish between data objects and user objects, the latter representing users containing all relevant information associated with the users, including unique user identifiers. For auditing purposes, every user object maintains a history of transactions that the user initiated over a period of time. The next subsection explains how this history information is structured.

When no ambiguity can arise, we will refer to user objects simply as users.

4.3. Transaction Objects

A *transaction* is defined as an operation performed on an object by a user, as contrasted with an operation performed by another object. (Note that the definition could be extended by allowing a transaction to be a *sequence* of operations.) Thus, a transaction is initiated when a user sends a message to an object. The method invoked by this message can send other messages to other objects, and so on. Consequently, a transaction consists of nested operations on objects.

A transaction execution can be represented by a *transaction tree* — a directed tree with labeled nodes and edges. In such a tree, nodes represent objects, while directed edges represent messages. The root node represents the user who initiated the transaction. Every other node is the recipient of a message from its parent in the process of the transaction execution. The root must have only one child to be consistent with our definition of a transaction as a single message dispatch from a user to an object. In a transaction tree, messages travel downward, along the tree edges, while return values travel upward, from child to parent , i.e., against the edge direction.

In the course of a transaction execution, some of the objects involved may have their internal states modified, i.e., have the current value of an instance variable changed. The corresponding transaction tree differentiates between the nodes that represent such objects and all other nodes. Nodes of the first kind shall be called *black*; nodes of the second kind, *red*.

Figure 1 shows a sample transaction tree. Red object nodes and the root are denoted by single circles; black nodes are denoted by double circles. A user u, represented by the root, initiates a transaction by sending a message μ_1 to an object o_1. The resulting method execution in o_1 causes the dispatch of messages μ_2, μ_3, and μ_4 to objects o_2, o_3, and o_4, respectively. Then o_2 sends μ_5 to o_5, while o_4 sends μ_6 to o_6 and μ_7 to o_7. Finally, objects o_8 and o_9 get messages μ_8 and μ_9, respectively, from their parent — o_7 — in the message tree. The transaction is completed when u gets a return value from o_1.

Every transaction is assigned a unique timestamp at the initiation time. Thus, timestamps can be used to uniquely identify transaction trees. In Figure 1, the entire tree is labeled by t, the timestamp of the transaction that the tree represents.

Every time a user runs a transaction, a corresponding transaction tree is created and saved as an object. This *transaction object* constitutes the third kind of objects that are present in the database (the

other two were data objects and user objects). Once created, a transaction object is never modified. Thus, the activity of all users is recorded as a *forest* of transaction trees. Each tree in the forest can subsequently be traversed during audit operations.

The transaction forest represents a second source of audit information, complimenting the information contained inside auditable data objects in the form of value sets associated with instance variables (see Section 4.1). The information from the two sources is related in the following way. When an instance variable A of an object o is updated during an operation $\mu(m, l)$ on o, a new triple (v, u, t) must be added to the value set $V(A)$. The value v is computed in the process of the execution of the method m. The other two elements of the triple, however, cannot be computed internally. They are determined by the transaction tree within which the current operation μ is nested. The root label of that tree gives us the identity of the user responsible for the update — u, while the timestamp of the update — t — is set to the timestamp of the entire tree.

5. SUPPORT FOR QUERYING THE AUDIT DATA

Now that we have organized all database activities as objects, we show how these objects can support high-level queries (operations) for the purpose of review and analysis of the audit data.

5.1. Operations on Auditable Data Objects

When a basic object is recast in an auditable form, i.e., when single values of instance variables and single programs of methods are replaced by value sets and program sets respectively, that should not affect the existing operations in the object's interface. That is, one would expect all of the old operations to be still available in the auditable object. Moreover, they should have the same semantics as before. However, the interface may be expanded to include new, audit-related operations.

Let F^b be the interface of a basic object o. When o is made into an auditable object, the new interface becomes F^a. In accordance with the above, we require that $F^b \subseteq F^a$. Then, $F^a - F^b$ contains the audit-related operations. The exact make-up of $F^a - F^b$ will depend on the needs of the particular application. The following is a sample list of possible operations.

Last_User ()

This operation returns the identity of the last user to modify the object. The method executing this operation checks the current values of all the instance variables and returns

$$u_i^l \text{ s.t.}$$

$$cur(A_i) = v_i^l \text{ and}$$

t_i^l is the largest timestamp of $cur(A_j)$ for all $A_j \in A$

Program(M, t)

This message is a request for the body of the program of method M (where $M \in F^b$) valid at time t. Assuming that the history of the programs associated with method M is $P(M) = \{(g^1, u^1, t^1), \ldots, (g^s, u^s, t^s)\}$, in response to *Program*(M, t) the object returns

$$Program(M, t) = \begin{cases} g^s & \text{if } t \geq t^s \\ g^j \text{ s.t. } t^j \leq t < t^{j+1} & \text{otherwise} \end{cases}$$

The next subsection contains more material on data object auditing.

5.1.1. Sending Messages in the Past

The complete audit functionality may require the ability to perform query operations on (send messages to) objects "in the past." Let t_0 be the current time. To send a message μ to object o in the past means to determine the outcome of a given operation if it were performed at some time $t \leq t_0$, Such operations (messages) will be referred to as *dated*. In a relational database, this is analogous to asking for the value of an attribute of a given tuple valid at time t. However, in the object-oriented context, the important difference is that the result of the operation depends not only on the values of the object's instance variables at t but also on the code that was used to implement the corresponding method at that time.

Another difference is that, because of the notion of data abstraction in object-oriented databases, it is assumed that instance variables of a particular object are not visible from the outside. Therefore, it is more appropriate in this context that an auditor collects information about the past states of an object by performing dated operations on it rather than asking to see the values of specific instance variables in the past.

We assume that the following three system calls can be used by methods:

read(A_i)
(Returns the current value of instance variable

A_i.)

write(A_i, v)
(Changes the current value of A_i to v.)

send(μ, o)
(Sends message μ to object o.)

In a conventional object-oriented system, the semantics of such calls are straightforward. In the context of our audit model, however, an explanation is in order.

Let us first consider the case of a method execution in response to a regular (i.e., undated) message $\mu_x = (m_x, l_x)$. When a call **read**(A_i) is encountered in the program for method m_x (recall that the names of the messages and their corresponding methods are identical), which is under execution, the system executes such a call by returning $cur(A_i)$, the current value of A_i. A call **write**(A_i, v) results in adding the tuple (v, u, t) to the set A_i, where t and u are the timestamp and the user id, respectively, of the current transaction tree (i.e., the transaction tree containing μ_x). Finally, **send**(μ, o) is executed just like in a conventional object model.

In order to facilitate the mechanism for sending messages in the past, we modify the syntax of system calls to allow for an additional parameter. In particular, **send**() now has three parameters instead of two. The third parameter is a time value, and the syntax of the call becomes **send**(μ, o, t). When the time parameter is not supplied, it is assumed to be t_0, the current time. When $t \geq t_0$, the call is interpreted as before. When $t < t_0$, the value t is used to date the message.

A dated message changes the execution procedure for the corresponding method. Responding to a message $\mu_x = (m_x, l_x)$ from a user o_x, dated $t_x < t_0$, an object invokes the method with the name m_x. However, the program that is executed is not necessarily the current one, $cur(m_x)$, but the one that is contemporaneous with t. Let us denote that program $prog(m_x, t)$. Assuming that $t < t_0$ and the history of the programs associated with method m_x is $P(m_x) = \{(g_x^1, u_x^1, t_x^1),..., (g_x^s, u_x^s, t_x^s)\}$, the program in question is computed as follows:

$$prog(m_x, t) = \begin{cases} g_x^s & \text{if } t \geq t_x^s \\ g_x^j \text{ s.t. } t_x^j \leq t < t_x^{j+1} & \text{otherwise} \end{cases}$$

Note that the $prog(m_x, t)$ would also be the program returned in response to the message $Program(m_x, t)$ (see Section 5.1).

Let us refer to the execution procedure described above, i.e., the execution of a method in response to a dated message, as a *dated* execution. In a dated execution, a call **read**(A_i) encountered in the body of the program is automatically replaced with **read**(A_i, t). The modified call is executed as follows. Instead of returning $cur(A_i)$, the system returns the value of A_i that was current at time t. Let us denote that value $val(A_i, t)$. If the history of the values of A_i is

$$V(A_i) = \{(v_i^1, u_i^1, t_i^1),..., (v_i^r, u_i^r, t_i^r)\},$$

then $val(A_i, t)$ is computed as

$$val(A_i, t) = \begin{cases} v_i^r & \text{if } t \geq t_i^r \\ v_i^j \text{ s.t. } t_i^j \leq t < t_i^{j+1} & \text{otherwise} \end{cases}$$

It is natural to require that an operation performed in the past be not allowed to modify the objects (otherwise, it will be like changing the past). Accordingly, a call **write**(), when encountered during a dated execution, is simply omitted.†

5.2. Traversing Transaction Trees

In this subsection we specify some audit operations that could be defined on individual transaction trees. It is not our intention to be complete, but rather to illustrate the possibilities allowed by the model.

As we mentioned in Section 4.3, a transaction object is never modified. Thus, none of its methods can contain a **write**(). The external view of a transaction object is (t, Q), where t is the timestamp of the corresponding transaction (t serves as the object's identifier) and Q is a set of audit operations (methods) allowed on the object t.

Throughout this subsection we will use the tree in Figure 1 as an example. Every message shown below is followed (in square brackets) by the response it would generate if sent to the tree object of Figure 1.

User()
[Response: (u)]

This is a request for the identity of the user who initiated transaction t.

Objects_Accessed()
[Response: (o_1, o_2, ..., o_9)]

† We assume that a transaction never reads what it has written, otherwise omitting the write calls may give inconsistent results. Note, however, that this assumption does not force us to sacrifice any functionality, because any transaction can be rewritten in the required form.

When this message is sent to a transaction object, a list is returned to the sender that consists of all the nodes of the transaction tree other than the root. Those are all the objects on which an operation was performed in the course of the transaction. In our example, the list will contain $o_1, o_2, ..., o_9$, i.e., all the nodes of t with the exception of u. This method can be based on any tree traversal algorithm.

Objects_Modified()
[Response: (o_1, o_4, o_5, o_6)]

This is similar to *Objects_Accessed*, except that only black nodes from the tree are selected. In our example, the list returned in response to this message will contain o_1, o_4, o_5, and o_6.

Operation(o_i)
[Response: (μ_i), if $1 \le i \le 9$; NULL, otherwise]

The purpose of this message is to determine what kind of operation, if any, was performed on the object o_i as part of the execution of t. The corresponding method traverses the tree searching for the node labeled o_i. If such a node is found, the label of the edge coming into the node is returned as a response. Otherwise, the method returns NULL, indicating that no operation was performed on o_i as part of the transaction t. Note that the response — $\mu_i = (m_i, l_i)$ — contains the name of the message (operation) as well as the list of parameters that were used with the message.

5.3. Transaction Forest

To facilitate operations on individual transaction objects, we must provide convenient access paths to them. For instance, it is essential to be able to access transaction objects by their user identity. The most natural way to provide this form of access is to associate each group of trees representing transactions initiated by the same user with the corresponding user object. This can be done by including a set-valued instance variable H_j in every user object u_j. H_j is a *transaction-history* instance variable. Its value is a set containing the unique timestamps of all the transactions that have been initiated by u:

$$V(H_j) = \{ t_j^1, ..., t_j^k \}$$

With this arrangement, every user object provides a convenient handle on the transaction trees for all those transactions for which the user has been responsible.

It is also important to provide access to transaction objects by their timestamps. This can be accomplished by creating a special transaction-forest object F with a set-valued instance variable R whose value is the set of all transactions that have been run in the system up to the present:

$$V(R) = \{ t_1, ..., t_n \}$$

Figure 2 illustrates the structure of the transaction-forest. The four rectangles are user objects; the triangles denote transaction trees; and the diamond denotes the forest object. The user u_1 has executed transactions t_1, t_2, t_3, and t_4. The set consisting of these transaction objects constitutes the transaction history of u_1 and makes up the value of H_1. This is indicated by the arrows going from u_1 to $t_1, ..., t_4$. The transaction history of u_2 includes a single transaction tree t_5. The user object u_3 has in its transaction-history set transaction trees $t_6, ..., t_{10}$. Finally, the user u_4 was responsible for transactions t_{11} and t_{12}.

5.4. Audit Operations on User Objects

The following audit operations can be included in the user-object interface.

Transaction_History(t', t'')

When sent to a user object u_i, this message returns a subset of the current value of H_i consisting of the timestamp identifiers of all transactions executed between the times t' and t'' by u_i:

$$\{ t \mid t \in V(H_i) \text{ and } t' \le t \le t'' \}$$

The time parameters t' and t'' can be omitted, in which case the entire $V(H_i)$ is returned. For example, *Transaction_History*(u_4) (see Figure 2) returns $\{ t_{11}, t_{12} \}$. Now, if more information is required about a specific transaction object from this set, the auditor can send appropriate messages to that object (whose identity is now known).

Objects_Accessed (t', t'')

This operation performed on the user object u_i will return the identity of all data object modified by any transaction initiated between the times t' and t'' by u_i. When the time parameters are omitted, it is assumed that the entire period of the user's activity is of interest. In that case, the list of nodes from all the trees associated with that user is returned. In order to successfully perform this operation, the object u_i will have to send appropriate messages to the individual

trees connected to it. Note that the name of the message coincides with the one used on individual tree objects. The semantics, however, are different, and so are the methods employed.

Objects_Modified (t' , t'')

This operation is similar to the above, except that only black nodes are returned.

5.5. Operations on the Transaction Forest

An audit interface is provided with the object F, to facilitate the type of audit operations in which access to transaction trees must be by their timestamps rather than user identities. Recall that object F maintains a handle on the entire transaction forest (Figure 2). Some useful audit operations on F follow.

User (t_i)

This operation returns the identity of the user who executed transaction t_i. To perform this operation it is necessary to access the transaction tree t_i and get the label on its root. Note that this would involve sending a message from F to t_i.

Last_Transaction ()

This operation returns the identity of the last user to execute a transaction on the database along with the timestamp of that transaction. Upon receipt of this message, the forest object invokes a method that finds the largest timestamp in $V(R)$, say t_k, and sends the message *User* () to t_k.

The following three operations are similar to the ones defined for individual user objects, except that here all trees must be checked, not just the ones associated with a particular user.

Transaction_History(t' , t'')

This operation returns the timestamps of *all* transactions executed in the system between t' and t''.

Objects_Accessed (t' , t'')

This operation returns the non-root nodes of all the trees corresponding to transactions between t' and t''.

Objects_Modified (t' , t'')

This operation is similar to the above, except that only black nodes are returned.

5.6. Discussion

The audit information in our model is concentrated in two places: the auditable data objects and the transaction trees. The auditable objects contain the history of changes to their own state. The transaction forest records the user activity.

The model results in *zero information loss*, i.e., the entire database activity can be fully reconstructed using the audit information that we keep. The easiest way to argue this point is to make a comparison to the traditional transaction log model. The transaction log which is characterized by zero information loss records the read/write actions of all transactions executed in the system (see [7, page 537], for example). The log contains the identities of the data items concerned. When a write operation is involved, the before and after images of the items are also recorded.

In our model, the transaction forest performs the function analogous to that of the log. The difference is that, since we are dealing with complex objects rather than flat data items (records), we must also record the type of operations performed on them.

The transaction forest records only the structure and relative order of the transactions (using timestamps), and not the before and after images of the data objects involved. That information, however, is not lost thanks to the history sets contained in the objects themselves. So the function of the log is split among the transaction trees and auditable objects.

6. CONCLUSION

Auditing capability is one of the requirements for secure databases. A secure database management system, among other things, has to provide facilities for not only recording the history of updates and queries but querying this history as well. In this paper, we have presented an audit model for object-oriented databases that satisfies both requirements. It imposes a uniform logical structure on both the current and the audit data that results in zero-information loss, i.e., the entire database activity can be fully reconstructed using the audit information we keep. Moreover, we provide high-level support for expressing audit and other database queries, and therefore, we make a complete audit trail methodology available.

Our new model seems to be a building block for a general-purpose object-oriented database system that holds the promise for a perfect logical organization of current as well as the audit data. To harness the promise of our model, further theoretical advances have to be made, and then a prototype has to be developed. It is presumed that this will lead to further investigation of the audit trail capabilities in answering questions dealing with possible breaches and other problems.

Acknowledgement

We are indebted to John Campbell, Howard Stainer, and Mike Ware for their support and encouragement which made this work possible.

References

1. "Department of Defense Trusted Computer System Evaluation Criteria," Department of Defense, National Computer Security Center, December 1985.
2. David D. Clark and David R. Wilson, "A comparison of commercial and Military computer security policies," *Proc. IEEE Symp. on Security and Privacy*, pp. 184-194., 1987.
3. Dorothy E. Denning, "An intrusion-detection model," *IEEE Trans. of Software Eng.*, vol. 13, no. 2, pp. 222-232, February 1987.
4. Sushil Jajodia, Shashi K. Gadia, Gautam Bhargava, and Edgar H. Sibley, "Audit trail organization in relational databases," in *Database Security, III: Status and Prospects*, ed. D. L. Spooner and C. Landwehr, pp. 269-281, North-Holland, Amsterdam, 1990.
5. Won Kim and Frederick H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*, Addision-Wesley, Reading, Mass., 1989.
6. Teresa F. Lunt, "Automated audit trail analysis and intrusion detection: A Survey," *Proc. 11th National Computer Security Conf.*, pp. 65-73, October 1988.
7. Ravi Sandhu and Sushil Jajodia, "Integrity mechanisms in database management systems," *Proc. 13th National Computer Security Conf.*, pp. 526-540, October 1990.
8. Bruce Shriver and Peter Wegner, eds., *Research directions in object-oriented programming*, MIT Press, Cambridge, 1987.
9. Henry S. Teng, Kaihu Chen, and Stephen C-Y Lu, "Adaptive real-time anomoly detection using inductively generated sequential patterns," *Proc. Symp. on Security and Privacy*, pp. 278-284, May 1990.
10. Stanley B. Zdonik and David Maier, eds., *Readings in Object-Oriented Database Systems*, Morgan Kaufman, San Mateo, Calif., 1990.

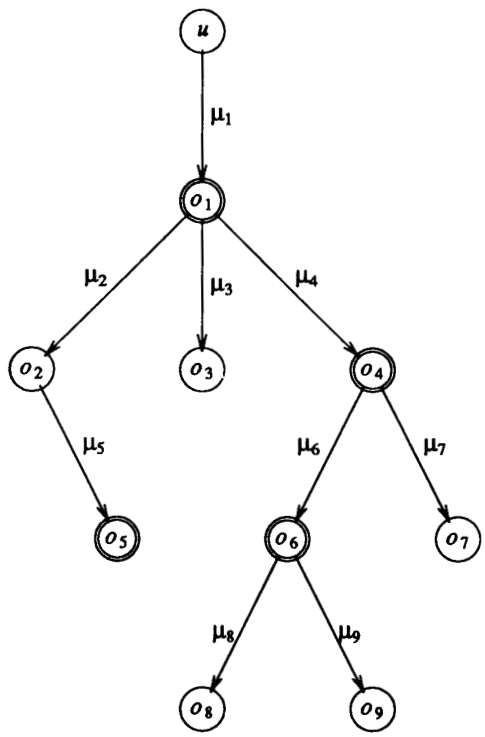

Figure 1. A transaction tree.

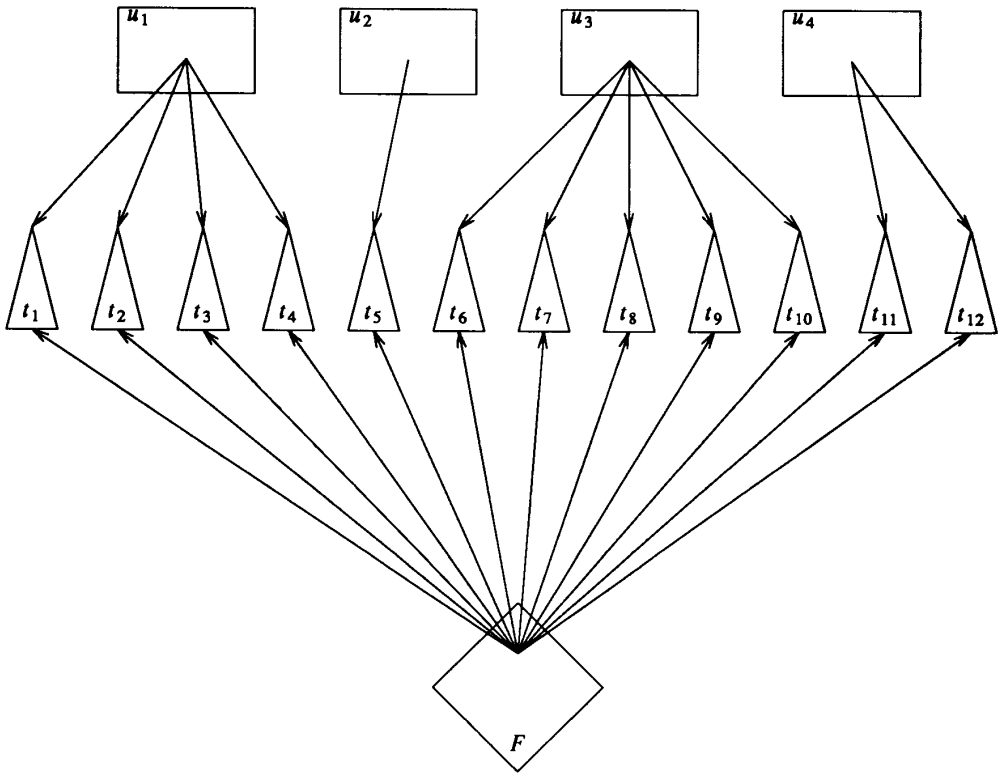

Figure 2. A transaction forest.