A Model for Active Object Oriented Database

Catriel Beeri   Tova Milo

The Hebrew University of Jerusalem*
{beeri,tova}@humus.bitnet

Abstract

This paper presents a logical model for an active object oriented database. The main idea is to use standard concepts of OODB such as encapsulation and inheritance. In particular triggers are special methods encapsulated in the appropriate object. The execution model uses nested transactions. The model is shown to be flexible and to generalize previous proposals.

1 Introduction

Conventional DBMS's execute only operations that are explicitly specified in users' requests, or in application code. In active databases operations can act as *triggers* that cause the execution of other operations. Such a facility has many uses, such as authorization and access logging, integrity constraint maintenance, and alerting [4, 7, 10, 12, 17, 19, 20, 16]. Its advantages include, among others, code reuse, and centralization of code storage and management [6, 18]. Code for integrity checking and maintenance is written, stored, and managed by the database administration, and is 'called' automatically by applications, without any reference in the application code. The incorporation of a trigger facility into a database system requires careful planning, so that it can be easily integrated with the data model and the processing mode of the system.

In DBMS's, operations are organized in *transactions*: collection of operations that are considered as atomic units for concurrency and recovery purposes[14]. This has been refined to a model of *nested transactions* [3, 13], in which transactions can have subtransactions, thus allowing for a hierarchical organization of processing. It is natural to view triggered operations as subunits of the users' transactions, that is, as subtransactions. Thus, recently proposed models for active databases [11] are based on the nested transaction concept.

An important trend in DBMS evolution is the development of d sophisticated database models, in particular *object-oriented* databases (OODB's) [1, 2]. OODB's supports rich data structuring mechanisms, hierarchical and general relationships between objects, and the representation of data processing in the form of code (i.e., methods) that is stored and managed by the DBMS. The advantage of code reuse and centralized and uniform code management is shared by active databases. The hierarchical organization of structure and processing in OODB's is highly compatible with the nested transactions model. This paper is a study of the integration of these three paradigms: *active databases, OODB's*, and *nested transactions*.

An active database model needs to address several issues: How are triggers specified; when are triggers enabled and executed; how is the environment of trigger execution specified and implemented; how do we specify and control concurrently enabled triggers. Each of these has received considerable attention in the literature, [6, 7, 10, 12, 18, 19, 20, 16]. We consider these issues in the context of the object-oriented paradigm, and present an integrated execution model that is based on objects and their structure, methods, and nested transactions. The following are the significant points of our approach:

Our model uses extensively OODB concepts such as encapsulation and inheritance. For lack of space we present here only a logical model, a framework, not a specific system. Issues relevant to performance are not included.

In section 2 we present an OODB and nested transaction models, and relate them to active behavior. The following sections consider the extensions to an active object-oriented model. Section 3 extends the description of methods to include triggering information. Section 4 describes how information about triggered actions is stored. An execution model is described in section 5. These three sections assume a simplified situation, where triggering and execution of triggered actions occur in a single object. Section 6 presents the general multi-object case. Section 7 present a review and comparison with previous work, and conclusions are presented in Section 8.

2 The Database Model

There is no single accepted OODB model. However, there are certain characteristics of OODB's that are widely agreed upon [1, 2], which are sufficient for the development of our approach. In the following we briefly describe these characteristics. We also briefly describe a general nested transaction model.

2.1 Objects, Methods, Classes and Encapsulation

An object oriented database is a collection of objects. Every object has a unique, unchanging *identity*, and a *state*. Updates are reflected in changes of objects' states. In addition, objects are associated with operations that can be performed on them, called *methods*, that serve to perform both updates and retrievals. A method of an object is invoked by a message sent to the object, possibly with parameters. A method may directly access the object's state, or invoke other methods of the same or other object(s). An object *encapsulates* both data (its state) and behavior (its methods), and can only be manipulated using the (external) methods associated with it. In our model the active aspects of an object's behavior are part of the object, like its state and methods.

In relational databases, the triggering events are operations such as retrieve and update, or transactions. In OODB's the interaction with the database is through the methods. Therefore, we postulate that the methods are the triggering events. In particular, one can view the database as one big object, and each transaction as a method of this object. Hence, there is no loss of generality in this approach. Similarly, the triggered operations are methods (often internal).

The common structure and behavior(methods) of objects is factorized in *classes* [1]. Active behavior is also defined at the class level.

2.2 Inheritance and Overriding

Inheritance is used to factor out common structure and behavior. If a class C has a subclass D than D *inherits* the structure and methods of C; it may have additional data and methods. We extend the inheritance mechanism to the active components of the behavior. If a method is defined to be a trigger of some action in a class, it has this property in all subclasses. Further, the inheritance mechanism applies to all the properties and behavior associated with the triggering facility. The benefits of inheritance for active databases are the same as for regular databases, namely flexibility, and economy of specification.

Some OODB's allow redefinition (*overriding*) of methods in subclasses, even redefining methods for individual objects in a class. The mechanism, if it exists, applies also to triggers.

2.3 Nested Transactions and Methods

In the nested transaction model a transaction may execute both operations and subtransactions. Subtransactions may also invoke subtransactions, so the execution of a transaction forms a tree in which the

[1] In some systems, part of this information may be given in type definitions.

leaves represent atomic actions and the internal nodes represent subtransactions. The executions of several transactions form a forest. Specific models are obtained by imposing restrictions or specific protocols; however, for this paper the differences are irrelevant. The advantages of nested transactions include increased concurrency and resiliency to failures. For example, when a subtransaction aborts, its parent is notified, and may then decide to execute an alternative subtransaction, or to change its execution plan, or to abort itself. Concurrency control and recovery for nested transactions have been investigated quite thoroughly. See, for example,[3, 13].

The nested transactions model is highly compatible with OODB's. Indeed, a transaction in an OODB executes operations on objects, by invoking methods. A method execution looks atomic from the transaction's viewpoint, but is often not atomic when viewed inside the object. It is natural to model it as a subtransaction. Methods invoked by other methods also can be viewed as subtransactions. One can view the database as a 'big' object, and the transactions as methods of this object. Subtransactions correspond to methods that are invoked by this method. We extend this idea further, following [7, 11] to include triggered operations as subtransactions.

Method invocations and terminations are the triggering events in our model. By the discussion above, this covers the complete range from database transactions to individual object operations.

3 Methods and Triggered Actions

In this section, we consider how and where properties of triggered actions are defined and represented. These properties determine, e.g., scheduling of the triggered actions, and their relationships to the triggering methods in terms of behavior under failures and concurrency, as explained in Section 5.

As stated previously, method executions cause the triggering of operations. We view methods (and also triggered actions) as entities with properties. We describe a method as a tuple:
type $method = [m-name, m-code]$.
The first attribute is the method's name, the second is the method's code. This tuple is included in a class definition. When the method *m—name* is invoked on any of the class objects, the code *m—code* is executed. Inheritance applies to this tuple in the sense that this name is associated with this code also in subclasses; overriding allows us to replace the code.

To enable methods to serve as triggers, we extend the definition of a method to the following triple:

type $method$ =

$$[m-name, m-code, \{triggered-action\}]$$

Each element in $\{triggered-action\}$ is a name of an action. Whenever the method is executed on an object, the actions in the set are triggered (although they are not necessarily executed immediately).

3.1 An Example

We shall use the following as a running example.

A bank's database stores information about customers accounts. A customer may have several accounts: current account, savings account, and so on. Each customer is represented by an object that includes a subobject for each of the customer's accounts. In addition, the customer's credit standing is summarized in a field, which we take for simplicity to be just the total balance of the accounts. This is a derived field; its value is kept current by appropriate triggers: Whenever any account is updated an action is triggered that recomputes the balance by summing the individual account balances.

3.2 Triggered Actions Scheduling

When should a triggered action be executed ? Many proposals do not support explicit specification of triggered operation scheduling; it is determined by a fixed algorithm embedded in the system [10, 17, 19, 20]. The most elaborate scheme to date seems to be that of HIPAC [7], that offers *immediate, deferred*, and *decoupled* execution (See section 7). We believe that a more flexible definition of scheduling is both desirable and feasible.

Consider, in the banking example, a transaction that updates the balance in several of a customer's accounts, and after some additional processing retrieves the new total balance. Since account balances are being updated, the total balance must be recomputed. However, there is no need to perform this update immediately; it must be executed only before the total balance is retrieved. Postponing the recomputation has the advantages that it can be performed once only, although several accounts were updated, and that the system may have some freedom in choosing the precise time when it is performed. We would like to be able to specify such behavior.

Our solution is to allow a programmer to specify an **execution interval**, in which the triggered action must be executed. An interval is specified by describing its end points. These points are *events* recognizable by the system. We consider events related to the execution of methods (not necessarily the method that triggered the action). In the above example, suppose that the transaction invokes two methods: The accounts are updated by the method *update−accounts* (which, in turn, calls the method *update−account* for each account). This is the triggering method. Then the total balance is retrieved by *get−balance*. The execution interval for the triggered action that updates the total balance starts at the end of *update−accounts* execution, and ends just before *get− balance* begins. Thus, the beginning and end of method execution are events.

A method may be executed several times within a transaction, and any number of times in other transactions. Obviously, an event refers to a specific execution of a method. It follows that the system must provide a mechanism for referring to method executions. Recall that every method execution is a node in the forest of the nested transactions in a system execution. The programmer of a triggering method cannot refer to other transactions; but should be able to refer to other nodes in the tree that contains that method. In our example, the programmer may label the executions of *update−accounts* and *get− balance*, and use those labels as event denotation in the execution interval. It is also desirable to refer to methods by name or type, such as "any retrieval". See section 5. Also other events e.g., time points, such as '12:00 noon', or 'monday morning' should also be specifiable. We consider those in section 6.2.

In addition, it seems desirable to have events constructors, to create complex events. For example **min** and **max** specify the first or last of a set of events, **and** specifies that all events in a set need to occur, and so on. Finally, to be able to specify only one end point of an interval, it is useful to have the event *any time*.

In summary, we envision an *event specification* language, that supports naming conventions for events, and event expressions. Since we do not describe a specific system, we do not present a specific language.

An execution interval is described by the tuple:

$$\textbf{type}\ \textit{execution-interval} = [\textit{start-event}, \textit{end-event}]$$

The triggered action must be executed as early as the start event and no later than the end event.

We conclude with two observations. First, for each event, the system must be able to detect that it has occurred, for otherwise it cannot ensure that triggered actions are executed in the appropriate time. Second, it is in principle possible to specify an interval such that the start event follows the end event. This is considered as error; it is the the programmer's responsibility to provide correct specifications.

In general, there may exist several triggered actions that are simultaneously eligible for execution. This subject is considered in section 5 and 6.

3.3 Failures

Transaction management is concerned with concurrency and failures, and these need also be addressed for triggered actions. We leave the discussion of concurrency to Section 5, and consider here failures. It is necessary to specify what happens to a triggered action in case the triggering method aborts, and what should be done in case a triggered action fails.

It is convenient to assume that the system has a *transaction management sublanguage*, that is used in triggered action definitions for specifying what should be done in the various failure cases. We do not present any specific language, but the discussion below presents options it should support.

3.3.1 Triggering Method Failure

There are two options in this case. Either the abort of the triggering method causes the rollback of the triggered action if it has already been executed and its deletion otherwise; or the triggered action is executed as planned. In the bank example, when an account update is aborted, there is no need to recompute the total balance. But in an access control system that uses triggers for logging of update requests, those triggers should be executed regardless of whether their triggering updates succeed or abort.

This idea is not new; in the HIPAC system [7] irreversible triggering is achieved by the *decoupling* mechanism (the triggered action that logs the access is executed as a separate top transaction) However, in HIPAC whenever the triggered action is not decoupled, it is a subtransaction of the triggering method and is aborted if that method aborts. We propose to allow the triggered action not to be aborted, independently of whether it is a subtransaction of the triggering method. This can be specified by *abort* and *ignore* statements in the appropriate part of the triggered action description.

An issue that deserves attention is the relationship with the execution interval specification. What happens if a triggered action is scheduled for execution after the termination of a method that failed? One option is that in this case the termination event never occurs, hence the triggered action will never be executed. This means that irrespective of the specification given by the programmer, the triggered action is in effect aborted and it can be deleted. Better option is to distinguish explicitly between successful and aborted terminations, and specify the execution interval in terms of these event types.

3.3.2 Triggered Action Failure

In the banking example, the recomputation of the total balance is essential for the correctness of subsequent retrievals. If the execution of this trigger fails, the system must either roll back the balance update that triggered it, or retry to execute the trigger and block retrievals of the total balance until it succeeds.

In general the options that one should be able to specify for such failures should include *ignore* and *abort(trans)* for specifying that the failure of the triggered action should be ignored, or should cause some (sub)transaction in its transaction tree to be aborted. Other useful options include: *try n times before deciding to abort*, and *run t instead.* (the second option allows, in particular, to execute a compensating transaction for the triggering method.) More generally, combinations of these basic options are useful, for example: "Try to run the action at least 5 times and if they all fail try to run *t* instead. If *t* also fails, abort the transaction".

3.4 The Triggered Action Description

The description of a triggered action should include the action it executes, i.e. a name of the method to be executed, and the object where it is to be executed. This may be the same object as the one in which the triggering method executed, or a different object. In the banking example, the 'update total balance' operation is executed on the same customer object in which the balance updates took place. Similarly, it includes the parameters passed from the triggering method execution. In summary, the information about each triggered action is recorded by a tuple:

type $triggered-action-description$ =
    $[triggered-action,$
    $act-to-perform(location, parameters),$
    $execution-interval,$
    $scheduling-information,$
    $triggered-action-fail,$
    $trigger-fail]$

The attribute $triggered-action$ contains the name of the trigger. Its value serves to connect this tuple with the method specifications where the same value appears in the third field. $act-to-perform$ names the action to be taken, the object where it is to be executed, and parameters. Of the other attributes, only $scheduling-information$ has not yet been described. It records information, such as a priority, that may be used by the system to determine which of a set of enabled triggers is to be scheduled for execution.

Naturally, the inheritance and overriding mechanisms apply to the new data that we have now added to classes. Method and trigger descriptions apply to all objects in the class, and are inherited by subclasses. Each field in these tuples, except for the name field , may be redefined. Thus, one can add or delete triggered actions, change the behavior under failures of an action, and so on.

4 Active Objects

When a method is executed, actions may be triggered, but they need not be executed immediately. Hence there is a need to store information about triggered actions, until they are executed. The information is derived from the data recorded in *triggered−action−description*, possibly by instantiation of attribute values.[2] We discuss in this section the structure used for storing this information, and avoid the consideration of where, i.e., in which object, this information is stored. For simplicity, we assume for now that both triggering and execution of an action occur at the same object, and the information is stored in this object. The multi-object environment will be considered later.

4.1 Object Structure

We extend the structure of the object state so as to contain information about triggered actions that are **waiting for execution** at the object. The regular part of the state is referred as the *passive* part, and the new information as the *active* part. For each action waiting for execution, there is a record of the following form:

type *active−action* =

$[triggered-action(parameters),$

$triggering-method-identity(location), \ldots]$

The triggering method identity identifies both its name and the execution, since several instances of a method may execute concurrently, and it is necessary to relate a given triggered action instance with the appropriate execution of the method. In the present context, *location* that describes the object where the triggering method executed is redundant, but it is needed in the general case.

Whenever a triggering method executes on an object, elements that represent the method's triggered actions are inserted into the active part of the object, and kept there until after the corresponding actions have been successfully executed.

[2]E.g., a template representing the triggering method is replaced by its actual identity.

4.2 Methods for Handling The Active Part

The information in the active component is an integral part of the object. It can be accessed by any authorized user, and is also used by the system for executing the triggered actions. Following the OODB paradigm, the active part is accessed only by methods. We mention three methods (actually, methods types) that are relevant here. One is used for inserting *active−actions*, the second is used for retrieving and executing, and the third for deleting triggered actions when no longer needed. These methods share the properties of regular methods, they are inherited, and may trigger other operations. They are not visible to application programs.

We can use the same three methods for handling all triggers in all classes, But we suggest to use for every *triggered-action* a special insertion method to handle its insertion. The following example illustrates why this may be useful. Recall that the *update−balance* method triggers a total balance recomputation. But several account updates for the same customer will trigger several instances of this action, and obviously only one is really needed. This problem can be solved by using for this trigger a special insertion method, that inserts a record for the triggered action only if there is no record for the same action in the object. Thus, the first update to any account of the customer causes the insertion of an *update−total* triggered action, but subsequent updates do not insert this action again, as long as the trigger has not been executed yet.

The use of special insertion methods enables us to explicitly define how new triggered action relates to the waiting actions, to cancel the triggered action, or merge it with other actions. It is possible to have a general insertion method that is used for all triggered actions, and define special insertion behavior by overriding. Another option is to record the method to be used in an attribute, called *insert−method−name*, that is added to the *triggered−action* record.

5 An Execution Model

So far we have presented fragments of our approach, such as what information concerning triggers is needed, and how it is recorded. In this section we describe a mechanism, that is responsible for all the activities of method execution, trigger recording and execution (but is still essentially restricted to a one object situation).

The basic idea is that when a method is invoked by a message, a *method processor* takes control. It is responsible for executing the method, recording its triggers, and executing triggers that need to be executed. In the following we refer to the execution of this processor when a method m is invoked as *extended-m*.

5.1 General Execution Plan

In the set of actions triggered by a method m, one can distinguish two subsets: actions that do not use any of the output parameters of m, and those that do. Those in the second set must be scheduled after m terminates. We denote the first set by T_1 and the second by T_2. The execution of the method processor when m is invoked is composed of five sub-transactions:

- The first sub-transaction, called *insert–start–m*, inserts records corresponding to the elements in T_1 into the active part of the object by invoking the insertion method of the triggered action, as a subtransaction. *insert − start − m* may also chooses the insertion order, or carries some maintenance activity such as changing priorities of elements in the active part of the object. At the end of the insertion, the active part contains the information about all the triggered actions of in T_1, as well as about all triggered actions that where triggered by some other methods, and have not yet been executed.

- The second subtransaction, called *execute − start − m*, now checks the contents of the active part of the object and executes the triggered actions that must be executed before the beginning of m, and possibly some that may be executed now but can also be executed later. Each of these actions is also executed by the method processor and may also trigger some actions.

- The third sub-transaction executes the method m. If m invokes other methods, a nested transaction structure is created and executed, using the method processor for each method. In particular, methods invoked by m may trigger actions.

- After the execution of m terminates, the values of the output parameters of m are well defined, so the fourth subtransaction, called *insert–end–m*, inserts records describing the actions in T_2 into the active part of the object. It thus completes the task partially executed by *insert−start−m*.

- The last sub-transaction, *execute−end−m*, executes the actions that must be executed no later than the end of m, and possibly a few others. It has the same structure as *execute−start−m*.

The following diagram describes the nested structure of *extended − m* generated by the execution of the method processor for m:

The root node *global − control − m* is the execution of the method processor. It executes the five subtransactions in a sequence, as indicated by $\prec$ in the diagram. It receives notifications about the success or failure, and in the latter case determines the course of action by analyzing the content of the active part of the object. For example, if m aborts, it may decide to abort itself, or to only delete some triggered actions, or to roll back some of those that have executed. (Note that such a rollback applies to the complete tree rooted at the aborted action.) Although such behavior might be considered abnormal if m were the root, it is now standard behavior of a nested transaction.

Also note that actions triggered by other methods may be executed as sub-transactions of *extended–m*. For example, if a previous method updated any of the accounts, then a trigger for updating the total balance was posted, and may be executed as part of a balance retrieval method.

In section 3 we defined a method as a triple that contains the method's name, code, and triggered actions. Now, we need to add components: the names of the methods that fill the roles of *insert − start − m*, *execute–start–m*, *insert–end–m*, *execute–end–m* for it. As remarked above, it may be reasonable to use

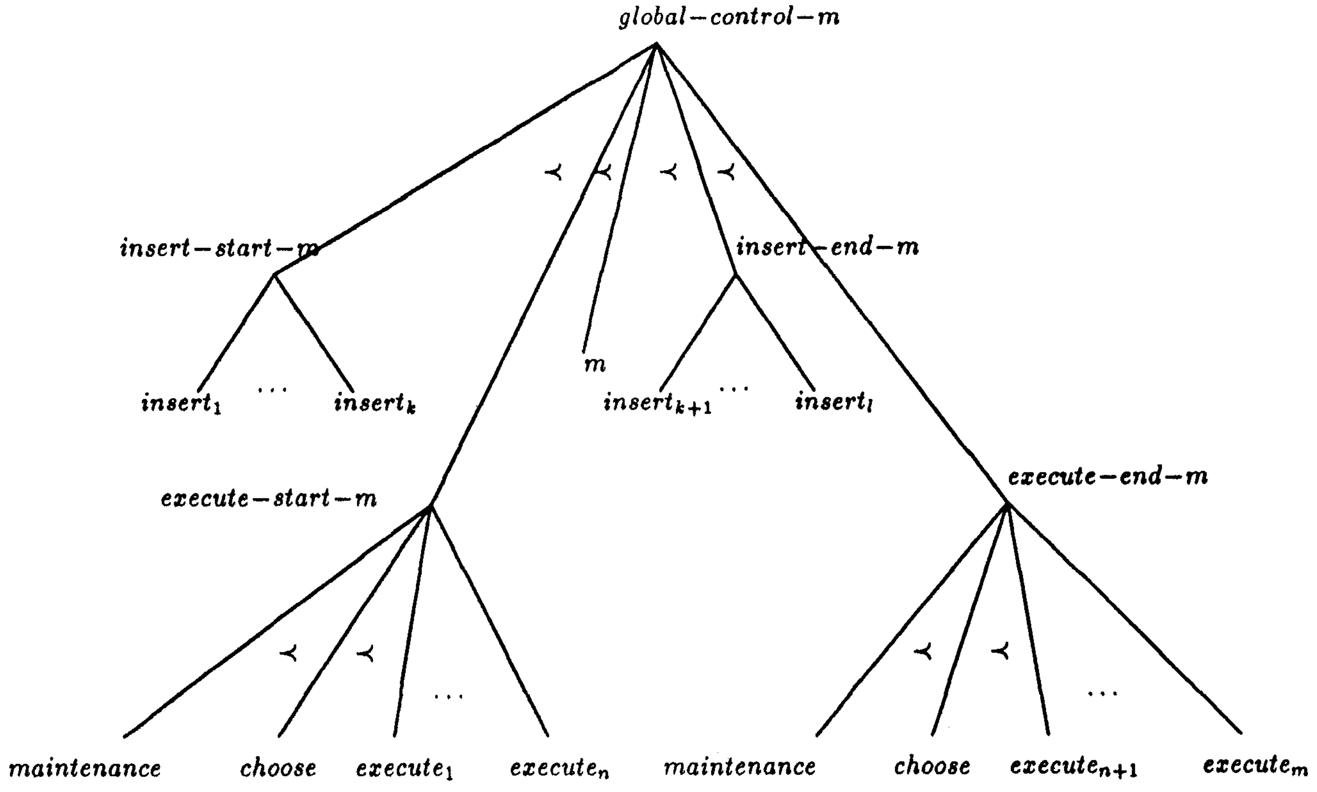

diagram 1. *extended−m*

special method for these tasks in some cases, rather than use general purpose methods.

Although the mechanism we have described is very flexible, it might seem too complicated; the effort involved in defining all the components of a method may seem to be a high price to pay. That is not the case however. Using a general method processor (which accepts m as parameter) implies a standard treatment of the triggered action execution for all classes and methods. Definitions of these standard methods are included in the class *object*, that is the root of the class hierarchy, and are inherited by all classes. Once this is given, the definition of a regular method is very simple: One must only specify the method name, its code, and the actions it triggers. This is the same amount of work as required in any other active system. The flexibility described above is still available for cases that need a special treatment by using the overriding mechanism.

5.2 The Sub-transactions

Of the five subtransactions, most of the work is done by *execute–start–m* and *execute–end–m*. It suffices to describe one of these, since they are almost identical. We present a full description of *execute–start–m*, followed by some general remarks.

The subtransaction *execute−start−m* is in charge of executing triggered actions before the start of m. It goes through a cycle composed of the following phases:

maintenance - The event 'beginning of m' is now considered as having happened. The execution intervals of the triggered actions are examined, and those whose start event 'has happened' are marked, if not already marked (perhaps in a previous execution of the method processor). In addition, actions whose end event 'has happened' (for example, those whose end event is 'beginning of m') are marked as *mandatory*. These must be executed now, before m is executed. The other marked actions are considered as *optional*. These actions can be executed at any time between now and the end of their interval.

choose - A set of actions is now chosen for execution. The considerations may include system load and performance (e.g., in a loaded system only mandatory actions may be chosen). If an empty set is chosen, then *execute–start–m* terminates. Hence, as long as mandatory actions remain, one of them at least will be chosen. It is not necessary to choose **all** the mandatory actions; those that were not chosen now will be chosen and executed subsequently. For example, it is possible to choose only the set of actions with the highest priority. Actions with lower priorities will be executed later. The *cycles* mechanism of HIPAC [11] is essentially such an algorithm.

execute - If the system supports parallel execution of subtransactions, the actions that were chosen can be executed in parallel; otherwise they are executed sequentially. The execution order among those that were chosen can be determined by analyzing the scheduling information that is stored for each action. Each of the actions is executed as a subtransaction unless it is specified to be independent (see section 5.3).

To execute an action, *execute–start–m* marks it as 'being executed', then invokes it as a subtransaction (except when it is independent) . If this subtransaction aborts, *execute–start–m* executes the program in the attribute *triggered–action–fail*, which as explained in section 3 contains a specification of what to do when the triggered action fails. Note that *execute–start–m* can execute commands such as *ignore*, *execute n times*, or *execute t instead*. However, it can not, for example, execute *abort m* (since m is not one of its sub transactions). For such a command, it informs its parent, the method processor about the failure and request the desired treatment. If the execution of a triggered action terminates successfully, the element describing it is removed from the active part.

The triggered actions that are executed are methods, so each of them may trigger actions. The execution of an action may spawn a subtree of triggered actions of its own. When the execution of this subtree terminates, control returns to *execute–start–m*. The database state may then contain information about additional triggered actions that have not executed yet. Therefore, the cycle of *maintain, choose,* and *execute* must be **repeated**.

5.3 Concurrency Control, Failures, and Recovery

A triggered action may be executed as a subtransaction of the method processor of some m, or as an independent transaction (if independence is explicitly specified). If for an action, neither the execution interval, nor any of its other parameters such as behavior in case of failure and scheduling information are related to m, than there is no reason to execute it as a subtransaction, which may delay the execution of m. If this is the case, the triggered action should be executed as an **independent transaction**, by sending an appropriate message to the transaction manager. We assume that the key word ***independent*** may be used in the specification of the triggered action for this purpose. The correctness of the specification is, as in other cases, the programmer's responsibility.

If the right end of the action's execution interval is the end of the triggering method, than it is executed as a sub-transaction of the extended triggering method. If the execution interval extends beyond the triggering method's end, than the algorithm used in *choose* determines in which transaction tree it executes. The data in the *trigger–fail* and the *triggered–action–fail* attributes must be compatible with the position that may chosen for the action in the transaction forest. One may specify that if the action aborts then so must do some or all of its parents. However, if the action is executed in a different transaction then the triggering method, it makes no sense to specify that the latter should be aborted — it may have committed already when the action fails. Of course, it is still possible to specify that a compensating transaction to the triggering method should be executed.

We recall from the way the method processor operates that triggering an action entails writing data in the active part of the object, and executing a trigger entails reading data from the active part, then modifying and finally deleting it. To ensure a serializable execution, concurrency control must apply to the active part of the object as well as on the passive part. Since every method that operates on an object must access its active part, if the whole active part is locked for each access, it may become a bottle neck. Our solution is based on locks for components of the active part of the object, and on a variant of predicate locking. Details will be presented elsewhere.

6 The General Multi Object Model

So far we have assumed a simplified situation where both triggering and execution occur at the same object. We now consider the general multi-object case, where the triggering method and the triggered actions may operate on different objects, and the events mentioned in the execution interval may occur in other objects. For example suppose that every deposit of more than 100,000$ triggers a notification to the branch manager, to be executed at 6pm. The deposit is executed on a customer's object, the triggered action on the manager object, and the event occurs at the system's clock.

Where should triggered actions be stored ? We adopt the principle that an object should encapsulate all the information relevant to it. Hence, the active part of an object must contain all the information regarding the triggered actions that will be executed at the object. It follows that each triggered action is inserted into the active part of the object on which that action operates.

The second issue is how does a method processor in an object know about events that occur in other objects. Such knowledge is needed for it to execute actions whose execution interval uses events of other objects. Our solution is based on *notifications* about the occurrences of events that are sent to it. The final issue we consider here is the notion of time events, such as 'at 6p.m.'

6.1 The Notification Mechanism

Suppose that a method m of object o triggers an action on the object o' that is scheduled to be executed when the event e occurs at an object o''. Our solution is to decompose the triggered action into a notification action on o'', whose task is to notify o' about the occurrence of the desired event e (it is scheduled to the occurrence of the event), and the original triggered action on o'. Each of those will be stored, and executed, at the appropriate object. We assume that every object has *send–notification* and *receive–notification* methods, for notifying and receiving notifications. Additionally, since execution intervals may contain complex events, it is necessary that each object should be able to store information it receives about events that occurred in other objects (or alternatively that we should be able to simplify complex events by removing events that had occurred).

In particular, $receive - notification$ is a method and is executed by the method processor just like any other method. Note that actions that should be executed when some event in another object occurs, will be executed at the end of the extended $receive - notification$, since only then it is known that the event has indeed occurred.

6.2 Time Events

The mechanism we have described is appropriate for events that are the beginning or end of a method. Now, we can easily extend it to time events, such as '6pm 11.1.91'. All that is needed is to have one *active object* in the database, call it a *timer*. For each triggered action whose execution interval contains a time event, it contains a triggered action, to send a notification to the appropriate object when the time arrives. It receives inputs from the system clock (at a sufficiently high frequency), and treats each 'clock tick' as a message from the outside world. The method processor than wakes up a notification method to send messages to the appropriate objects. The mechanism is, therefore, a special case of that described above.

7 Related Work

There have been quite a few proposals for augmenting a database with triggers or production rules [6, 7, 10, 12, 17, 18, 19, 20, 16]. We discuss here briefly some of the concepts and ideas of these proposals, and their implementation in our model.

7.1 Events

The above models typically consider relational databases and actions that are triggered by standard database operations such as retrieve and update [20, 18]. ETM [12] and HIPAC [7] have refined the *event* concept to include triggers caused by "abstract" events. We have followed and refined this approach: every method (even points in its execution) is a potential triggering event.

The HIPAC system also allows definition of composite events [8] using disjunction, sequencing. and closure. Our model does not directly support sequencing and closure but they can be simulated (i.e., programmed). For example, the *sequence* of two actions can be simulated by defining the first event as a trigger to an action that records it has occurred in a given transaction. The second event triggers the desired action, but its insertion method first checks if the active part contains the needed information about the first event. Closure is simulated similarly. Thus, in a sense, our model is more general since it allows programming of a variety of constructs, and is not restricted to a given set.

7.2 Rules

Focusing on relational databases, [20] presented *set-oriented* rules, that are triggered by an arbitrary set of changes to the database and may perform a set of changes. (For example a single set oriented rule might operate on all tuples that were inserted/updated/deleted from the database during the course of a (sub)transaction). Our model is essentially an instance oriented model. However, a set oriented approach can easily be implemented in it. We can make a (sub)transaction which operates on some class a trigger to an operation that accepts the set of changes as a parameter. In addition, each access to an object in the class triggers an action that records the changes, and sends the object id to the class, as a parameter to be used in the previous action. The overall algorithm is similar to the one described in [20]. The idea can of course be extended in various ways, since it is now under the control of the method programmer, rather than built into the system.

Generally, a production rule take the form of *when X then Y*, where X is a triggering condition(event) and Y is an action [20]. Our model considers rules of the form *when X then {Y}* , i.e all the actions that are triggered by an event are grouped together. This can be seen to be only a syntactic change.

Central to the HIPAC model is the concept of $event-condition-action$ (ECA) rules[8]. When the event occurs, the condition is evaluated and if satisfied the action is executed. Our model supports only events and actions. However, this idea can be programmed in our model: The event triggers an operation that evaluates the condition, and if the condition is satisfied it triggers[3] the action. Hipac uses similar mechanism for implementing ECA rules [11].

HIPAC also supports enabling and disabling of triggers. Following HIPAC's implementation, a new attribute can be added to triggered actions to indicate whether they are enabled. $insert-start-m$ and $insert-end-m$ will insert only enabled actions.

7.3 Triggers and Transactions

In System R [10], Postgres [17] and Sybase [19], triggered actions are typically executed in the same transaction as the triggering updates. They are either executed immediately (Sybase,System R), or deferred to the end of the transaction (assertions in system R). In Postgres, triggered actions can also occur on demand. In [20] triggers are executed at the end of the execution unit. The HIPAC model [7] supports three execution options: *immediate* - immediately when the event occurs, *deferred* - at the end of the transaction

[3]The triggering can be done by execution a "null" action which does not affect the database state, and is defined as a trigger to the desired action.

in which the event occurred, *decoupled* - in a separate transaction. In the first and second options, triggers are executed as sub-transactions of the transaction in which the triggering event occurred. In the third option the actions are executed as separate transactions, which can be either causally dependent, i.e must be serialized after the triggering event, and aborted if the triggering transaction fails, or independent. The HIPAC model supports two options for handling a rule failure: either the execution of that rule alone is aborted, or the whole transaction is aborted [9]. It can be easily seen from the discussion in the previous sections that our model supports all these features, and a few more.

7.4 Execution Order

Proposed systems offer different strategies for ordering triggered actions scheduled for the same time. In early System R [10] triggered actions are executed in a system defined order. Sybase [19] does not allow more than one trigger to be defined for an operation on a relation. Postgres [17] uses a conflict resolution strategy such that only the highest priority action is executed. In [20] several strategies are offered including partial order, and preferring least recently used rules. In HIPAC [7] triggered operations can be executed concurrently, using priority categories for ordering. It supports a cycling mechanism for deferred sub-transactions, and a pipeline mechanism for decoupled actions. We have not treated this issue in our model, but any mechanism or language for describing scheduling information can be incorporated into it, without affecting the overall structure.

8 Discussion

The model we have presented in this paper is a *logical* model that provides a clear semantics for the active behavior of an object oriented database. The integrations of the three paradigms (*active behavior*, *OODB's*, and *nested transactions*) supports a very powerful and flexible trigger mechanism. An advantage of our approach is uniformity: The standard mechanisms of OODB's and of nested transactions are applied to all the extensions needed to support active behavior. This includes inheritance and overriding, and identification of methods, actions and sub-transactions.

We have left many issues open. These include: the development of appropriate sublanguages for specification of events, scheduling, transactional structure and behavior; development of concurrency control protocols that will prevent the active components of objects from becoming bottlenecks; appropriate efficient strategies for storing active-behavior related data; mechanisms for selective execution of triggers (a feature that is very helpful for debugging). These are left for future research.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Ditrich, D. Maier, S. Zdonik. The object-oriented database system manifisto. *First International Conference on Deductive and Object Oriented Databases.* W. Kim, J-M. Nicolas, S. Nishio(eds), 1989,40-57

[2] C. Beeri. Formal Models for Object Oriented Database Systems. *First International Conference on Deductive and Object Oriented Databases.* W. Kim, J-M. Nicolas, S. Nishio(eds), 1989,370-395

[3] C. Beeri, P.A. Bernstein, N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Jutnal of the ACM 36(2):230-269,* April 1989.

[4] CODASYL Data Description Language Committee. *CODASYL Data Description Language Jurnal of Development* June 1973. NBS Handbook 113 (1973).

[5] S.Ceri, J.Widon. Deriving Production Rules for Constraint Maintenance. *IBM Research Report* RJ 7348, IBM Almaden Research Center, March 1990.

[6] Dayal,U. Active Database Managment Systems. Proceedings of the 3th International Conference on Data and Knoledge Bases. Jerusalem, Israel (June 1988)

[7] Dayal,U. et al. The HIPAC Project: Combining Active Database and Timing Constraints. Special Issues of real Time Database Systems SIGMOD Record 17,1, March 1988

[8] Dayal,U. Bushmann,D. McCarty,D. Rules are Objects Too: A Knoledge Model for an Active, Object Oriented Database Managment System. Proc.2nd International Workshop on Object-Oriented Database Systems West Germany, September 1988

[9] Dayal,U. Hsu,M. Ladin,R.. Organizing Long-Runing Activities with Triggers and Transactions. Proc of the ACM SIGMOD Int. conf. on Managment of Data. Atlantic City, NJ. May 1990.

[10] Eswarn,K.P. Specifications of Trigger System in an Integrated Databases Managment System. *IBM research Report RJ1820.* August 1976.

[11] Hsu,M. Ladin,R. McCarthy,D. An Execution Model for Active Database Managment Systems. *Proc. 3rd International Conf. on Database And Knoledge bases* Jerusalem, Israel, June 1988 .

[12] Kots,A.M. K.R.Dittrich, J.A.Mueller, Supporting Semantic Rules by Generalized Event/Trigger Mechanism. *Proc. Conf on Extending Database Technology* Venice, 1988.

[13] Moss,J.B.E. Nested Transactions:An Approach to Reliable Distributed Computing. *MIT Press* Cambridge Mass. 1985.

[14] Papadimitriou,C.M. The Theory of Database Concurrency Control. *Computer Science Press* Rockville,md.,1986.

[15] T. Risch Monitoring Database Objects *Proc. of th 15th international conf. on VLDB,* Amsterdam, 1989.

[16] E. Simon, C. de Mandreville. Deciding Whether a Production Rule is Relational Computable. *Proc. of the Int. Conf. on Database Theory,* Bruges(Belgium), Sept. 1988.

[17] Stonbreaker,M. et al. A Rule Manager for Relational Database Systems. *The POSTGRES Papers.* Univ. of Carolina, Berkley, Ca. Electronics Research Lab, Memo no UCB/ERL M86/85, 1986.

[18] Stonbreaker,M. Jhingran,A. Goh,J. Potamianos,S. *On Rules, Procedures, Chaching and Vies in Data Base Systems.* Proc of the ACM SIGMOD Int. conf. on Managment of Data. Atlantic City, NJ. May 1990

[19] Sybase,Inc. Transact-SQL User's guide 1987.

[20] Widon,J. Finkelstein,S.J. Set-Oriented Production Rules in Relational Database Systems Proc of the ACM SIGMOD Int. conf. on Managment of Data. Atlantic City, NJ. May 1990.