CHAPTER 3

3. RULE PROCESSING IN ACTIVE ORDBMS

3.1 TRIGGER RULE CONFLICTS IN EXECUTION MODEL

Active object-relational database systems extend the normal functionality of ORDBMS with support to monitor the changes to the database state and react to specific situations automatically without user intervention. The reactive behavior of active ORDBMS is represented by the triggers in the form of Event-Condition-Action (ECA) rule paradigm. A trigger in active ORDBMS SQL3 standard is an ECA rule that is activated by a database state transition and has SQL3 predicate as condition and a list of SQL3 statements as an action. A trigger is activated whenever a specified event occurs. The event is usually an INSERT, DELETE, or UPDATE on a particular table monitored by the trigger. Once the trigger is activated, an optional specified condition is checked and if the condition is true, an action is executed. If the condition is omitted, it is considered as true and an action is executed.

Triggers are used in a variety of significant ways in today’s database systems and applications. In general, triggers are classified into following categories based on the function and behavior [Ceri et al., 2000]:

- Constraint-preserving Triggers: Signal integrity constraint violations and force rollbacks of the violating transactions.
- Constraint-restoring triggers: Detect integrity constraint violations and modify the database contents in order to restore integrity.
- Invalidating triggers: Signal and mark integrity constraint violations, allowing applications to respond appropriately.
- Materializing triggers: Compute materialized derived information from simple scalar values to aggregate values to complex views, either by incremental modifications or complete refresh.
- Metadata triggers: Maintain consistency across system catalogs.

- Replication triggers: Replicate, migrate, or log information and/or modifications from one table or database (primary copy) to another one (the secondary copy).

- Extenders: Manage new types of data (e.g., validate input) and keep specialized external structures consistent with the base data.

- Alerters: Notify or push information to users in the form of messages based on publish/subscribe model.

- Ad-hoc triggers: Implement business rules, for example supply-chain management or any other application specific logic.

In order to support the active functionality for monitoring and reacting to specific event occurrences, each active object-relational database system has a rule definition facility that is used to define the specification of ECA rules for describing reactive behavior generally referred to as the knowledge model and also possess an execution model that determines the rule processing at runtime [Paton and Diaz, 1999]. The behavior of triggers is based on the execution model and it captures the runtime characteristics of the rule processing. In active ORDBMS, rule processing occurs when a database state is changed by the execution of update, insert, and delete operations requested by the user application. After each database command, the system checks for triggers eligible for execution. The existence of triggers in a database system affects its execution model significantly. The execution model determines the interaction of triggers and database applications.

The development of an execution model is based on two dimensions: the application and trigger granularities. The application granularity is the level at which trigger activation is detected. After each database command, the system checks whether there are activated triggers and executes them. Figure 3.1 shows the options for choosing the application granularity: a database session, database transaction, and a database operation. The finer application granularity leads to more possibilities for trigger usage systems such as user notification, application procedures, communication, integrity enforcement and protection. If the application granularity is
the database session, then messages can only be sent after the session has finished, which limits interaction with the user. However, if the application granularity is a database command, then messages are sent after each command and it is possible to define triggers which interact with the user application. Therefore, the database command level granularity is better than other application granularities for trigger execution.

![Diagram showing application granularities](image)

**Figure 3.1 Application granularities**

The trigger granularity is the level of atomicity at which trigger components are executed and Figure 3.2 shows the trigger granularities. In T2, the event condition, and action are individually atomic and in T3, the low-level operations that constitute the event, condition and action are atomic and the granularity becomes finer from left to right.

Consider a trigger system based on application granularity A2, i.e. a database transaction Tr, and trigger granularity T2, i.e., a trigger split into its three basic components E, C, and A. Then for an application run AG = {Tr1, Tr2, Tr3,...Trn} and for trigger execution TG = {E1, C1, A1, E2, C2, A2,...Em, Cm, Am}. The outcome of the system is determined by the scheduler over AG and TG. An example schedule is < Tr1, E1, C1, A1, Tr2, E2, C2, A2...>. Clearly, only meaningful schedule should be generated and the semantics of such schedules should be defined by extending the transaction semantics.
The significant aspect of active ORDBMS lies in the development of a prominent rule execution model for rule processing. The rule execution model of the active ORDBMS has been characterized by the various phases such as event detection, signaling of events, rule scheduling and execution. An important issue concerned with the development of rule execution model of an active ORDBMS is to process trigger rule conflict, when multiple rules are triggered at the same time. The trigger rule conflict occurs when an event activates several trigger rules simultaneously. In order to address rule conflict issue, active ORDBMS provide a mechanism known as rule scheduling, which performs the ordering and execution semantics of multiple triggered rules.

**Rule ordering**

The common policy adopted by most rule execution model of an active ORDBMS research prototypes as well as commercial systems use static (deterministic) priority schemes to specify the order in which rules are selected for
execution. Static priorities are determined either by the system based on rule creation time or by the user as an attribute of a rule. In the latter case, a rule is selected from a collection of multiple triggered rules for execution using numeric or relative priority schemes. Rules are placed in order using a numerical scheme, in which each rule is given an absolute value that is its priority, or by indicating the relative priorities of rules by stating explicitly that a given rule must be fired before another when both are triggered at the same time.

The user can explicitly specify a relative priority between particular rules by defining precedes relationship between them. If the user has specified that rule R1 precedes R2, and if both R1 and R2 have been triggered, then R is considered first for execution, regardless of the default total ordering. User-defined relative priorities are transitive; that is if, R1 precedes R2 and R2 precedes R3, then R1 precedes R3 even if R1 is not triggered. User-defined priority schemes are becoming increasingly important in active ORDBMS for providing deterministic behavior in rule execution. Rule processing with user-defined priority system is repeatable in nature i.e., for a given set of rules and priorities, the rules are considered for execution in the same order if the same set of transactions is executed twice on the same database state.

**Execution semantics**

The various execution strategies described for handling multiple triggered rules in execution models are:

Concurrent execution – All triggers are executed in concurrent and multiple triggers are executed at a time and the outcome of this strategy is predictable when the triggers do not interfere.

Ordered sequential execution – All triggers are executed sequentially and their execution order is chosen either at random or by priority given in the trigger definition. In this strategy one trigger is executed at a time and is useful when there exists a strict ordering based on specialization of the triggers. When the triggers do not interfere, the effect of a random execution order is the same as parallel execution and otherwise it results in unpredictable system behavior. Priority based selection
offers the application the possibility to control the execution order and this might lead to predictable outcomes.

Single trigger execution – Single trigger is executed and it is chosen at random or by priority. Random rule selection makes the rule system as non-deterministic one and the priority based single rule execution is useful when there exists a strict ordering based on the specialization of triggers.

To use triggers correctly, their execution should have a predictable result and therefore concurrent execution should be seen as an optimization technique. Whenever an event triggers multiple rules that execute over distributed sources of data or multiple rules that are to be triggered by many different events at the same time for advanced applications, it is important to make use of rule scheduling with concurrent execution of all rules to speed up the rule execution process for improving the system performance in active ORDBMS.

The research works in ADBS integrated with relational database systems support rule scheduling based on static priorities using numeric or relative scheme with sequential execution of all rules [Paton et al., 1999; Rasoolzadegan and Meybodi, 2010]. In active ORDBMS standard (SQL3), multiple rule triggering has been performed by the rule scheduler based on the system defined priority at rule creation time with sequential execution of all rules [Kulkarni et al., 1999; Paton and Diaz 1999].

However, modern large scale applications demand support for handling complex data, detecting events from different sources and the evaluation of conditions and the execution of actions performed on different situations, an enhanced rule scheduler is required to improve the ordering and execution of multiple triggered rules in active ORDBMS [Dube and Wu, 2009; Schaff et al., 2010; Wai Yin Mok et al., 2013]. Thus, the rule scheduler should support both sequential and concurrent rule execution based on user-defined priority scheme is an important and recent research work in active ORDBMS.
3.2 PROBLEM STATEMENT

This research work is aimed at rule scheduling and execution for trigger processing in active object-relational database systems. The problem is divided into three subproblems.

1. To specify the user-defined priority scheme to ensure predictable rule execution. In this research work, a development is made on how to extend the rule scheduling effectively with user-defined priority scheme for ordering rules to process trigger rule conflict set.

2. To perform rule execution with sequential, concurrent and the combination of both execution strategies to achieve the faster execution of multiple triggered rules. This work includes concurrent rule execution using nested transactions to improve system performance by reducing the execution time of multiple triggered rules.

3. To design and develop an ECA rule definition system for applications.

3.3 CONCURRENCY IN RULE EXECUTION

Concurrency is a property of a system representing the fact that multiple activities are executed at the same time. The concurrent execution of activities can take place in different environments such as single-core processors, multi-core processors, or even multiple machines as a part of a distributed system. Apart from recent hardware trends in multi-core and processor systems, the use of concurrency in applications is motivated by performance gains. The three ways in which concurrent execution is used to improve the system performance [Cantrill and Bonwick, 2008]: (i) reduce latency – a unit of work is executed in shorter time by subdividing into parts that are executed concurrently. (ii) Hide latency – multiple long-running tasks are executed together by the underlying system. This is particularly effective when the tasks are blocked because of external sources they must wait upon, such as disk or network I/O operations. (iii) Increase throughput – By executing multiple tasks concurrently, to make the system able to perform more work.
In recent years, there has been a growing need for active (or event-driven) systems that react automatically to events [Wasserkrug et al., 2012]. The event-driven system in the database has been impacted to manage and react to events in advanced applications. Multiple rule firing causes a large set of rules to be fired, and it occurs when an event causes more than one rule to be fired in active database systems. Depending on the level of concurrency allowed, a hierarchy of rules fired by the same event execute in the system.

In ADBS, concurrency in rule execution can be achieved in three ways such as inter-rule concurrency, intra-rule concurrency and both inter and intra-rule concurrency [Harder and Rothermel, 1993]. In inter-rule concurrency, rules are executed concurrently as if they are atomic transactions. Inter-rule concurrency is attained by executing the rules concurrently. In the case of intra-rule concurrency, rules are derived into subcomponents and those subcomponents are executed concurrently. Intra-rule concurrency is achieved by dividing a rule into a condition and an action, and executing them concurrently. The benefits of both inter-rule and intra-rule concurrency to be combined together provide the third alternative of flexible concurrent rule execution model.

3.4 NESTED TRANSACTIONS FOR CONCURRENT RULE EXECUTION

The concept of nested transactions supports the design and implementation of flexible control structure for concurrent execution of transactions in database systems. Nested transactions play an important role in improving concurrency which is not possible with flat transactions [Buchmann, 1999]. Nested transactions allow concurrency within a transaction. A nested transaction is a hierarchy of subtransactions, where each subtransaction contains other subtransactions. Nested transactions divide an existing transaction into a parent transaction and one or more subtransactions. Each parent transaction controls the execution of its subtransactions specifying which subtransactions execute concurrently.

A nested transaction is represented as a tree structure where the root of the tree is the main transaction consisting of a number of subtransactions. Each subtransaction
can in turn be a nested transaction. Since nested transactions are represented as a tree structure, the standard tree notations such as parent, child, ancestor, descendent, superior and inferior also apply to it. The root transaction which is not enclosed in any transaction is called the top-level transaction. Transactions having subtransactions are called parents and their subtransactions are the children. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. The superior or inferior relation is the non-reflexive relation of the ancestor or descendant. The set of descendant of a transaction together with its parent/child relationships is called the transaction’s hierarchy.

The root of the tree is called a root or top-level transaction. The root has one or more children and similarly children of the root also have their own children. The failures into subtransactions are localized by dividing the transactions into smaller granules. Subtransactions can abort independently without causing the abortion of the whole transaction hierarchy. When a transaction aborts, all of its descendants are also aborted, but other transactions are not affected. Thus nested transactions are very useful in terms of system modularity.

![Nested transaction tree example](image-url)
An example of a nested transaction tree is shown in Figure 3.3. In the nested transaction tree example, the root is represented by top-level transaction $T_R$. The children of subtransaction $T_C$ are $T_D$, $T_F$ and $T_G$ and the parent of $T_C$ is $T_B$. The inferiors of $T_C$ are $T_D$, $T_E$, $T_F$, and $T_G$ and the superiors are $T_B$ and $T_R$. The descendant and ancestor sets of $T_C$ additionally contain $T_C$ itself. The hierarchy of $T_C$ is depicted as the subtree spanned by $T_C$ descendants.

Using nested transactions four different kinds of parallelism used among subtransactions are: only sibling, only parent-child, parent-child and sibling and no parallelism.

- Sibling parallelism: In this case, parallelism is supported among siblings. Thus parent stops its execution while its children are running concurrently. Transactions can share objects with the parent without concurrency control mechanism and certain concurrency control scheme is required for objects shared among siblings.

- Only parent-child parallelism: In this kind of parallelism, each parent can concurrently execute with one of its children while other children wait. Thus only transactions along a single path in the hierarchy execute concurrently. This simplifies concurrency control scheme, as only transactions along the same path needs to be synchronized.

- Parent-child and sibling: This kind of parallelism allows any combination of transactions and subtransactions in the hierarchy can run in parallel. This case supports maximum parallelism but at the same time it causes high overhead to achieve concurrency control and usually difficult to implement.

- No parallelism: This case supports sequential execution of transactions in the hierarchy. Thus no concurrency control scheme is required.

In nested transactions, ACID properties are valid for top-level transactions, but only a subset of them holds for subtransactions [Harder and Rothermel, 1993]. A subtransaction may commit or abort independent of other transactions. Aborting a subtransaction does not affect other transactions outside of its hierarchy hence they
protect the outside transactions from internal failures. Nested transactions are proposed as a suitable tool to implement concurrent rule execution [Dayal et al., 1990; Beeri and Milo, 1991; Buchmann et al., 1995] and also to handle long running activities [Dayal et al., 1990; Dayal et al., 1991] in active database systems. Nested transactions has also been suited to handle long running transaction environments of advanced database applications include engineering design, cooperative work, software development processes, computer based design and manufacturing systems [Madria, 1997].

The advantages of using nested transactions are summarized:

- Achieve decomposition and finer grained control of concurrency than flat model.
- Intra-transaction concurrency allows execution of a long transaction into concurrently running smaller parts bringing an increase in overall efficiency and decreasing response time.
- System modularity allows the modules of a transaction program to be designed and implemented independently and facilitates a simple and safe composition of the transaction.

Nested transaction model has been designed to extend the flat transaction model to provide the ability to define transactions within other transactions by splitting a transaction into hierarchies of subtransactions [Moss, 1985]. Nested transaction model is a set of subtransactions that may recursively contain other subtransactions forming the complete transaction tree or hierarchy of transactions. The top level transaction can have number of child transactions and each child transaction can also have nested transactions. A child transaction may start after its parent has started, and may commit locally. The committed local result is, however, released only when all of its parents up to the root have successfully terminated. Transactions have to commit from the bottom upwards and a transaction abort at one level does not have to affect a transaction in progress at a higher level. Hence this model is also termed as closed nested transaction and it provides increased modularity and intra-transaction concurrency for subtransactions than the flat transaction model.
The nested transaction model is considered as a suitable tool to implement rule execution in ADBS since it provides a good model for concurrent rule execution and it can handle nested rule firing together with multiple triggering of rules [Paton and Diaz, 1999; Buchmann, 1999]. Nested rule firing occurs when the condition evaluation or action execution of a rule causes other rules to be fired and multiple firing of rule occurs, when an event causes more than one rule to be fired. However if a transaction hierarchy as a big module, its subtransactions are designed independently as submodules, high degree of intra-transaction parallelism within nested transactions has been required [Harder and Rothermel, 1993].

The concurrent rule execution with nested transaction model brings along concurrency control issues. The concurrency control is needed to maintain database consistency and provide high performance by allowing concurrent execution of rules. The concurrency control among nested transactions has prevented the database updates performed by one transaction interfering with database updates and retrievals from other transactions. Within a nested transaction, concurrency control has to ensure intra-transaction concurrency to maintain acceptable data consistency demands.

Concurrency control for nested transaction model based on two-phase locking has been proposed and in nested transaction model, enhanced concurrency between subtransactions is provided by a lock inheritance mechanism. When a subtransaction completes the lock it holds are passed up to its parent who retains them. A retained lock may be subsequently obtained by a descendant of the transaction which retains it, but by no other transaction (upward inheritance of locks). The nested transaction model has been modified to include downward inheritance of locks in addition to upward inheritance [Harder and Rothermel, 1993]. In downward inheritance, locks are passed from a parent transaction to child without the parent retaining the lock. Supporting downward inheritance of locks permits certain desirable decompositions of transactions which are not possible with upward inheritance. Downward inheritance provides higher concurrency but recovery is made complex.
3.5 NESTED TRANSACTION MODEL IN RULE PROCESSING

Conventional passive database systems execute queries or transactions only when explicitly requested to do so by user or application. The design of an active database system involves specifying rules that provide reactive capability. In ADBS, system responses are expressed using ECA rules. An ECA rule is composed of an event that triggers the rules, a condition describing a given situation, and an action to be performed if the condition is satisfied. Coupling modes between event and condition and between condition and action determine when the condition should be executed relative to the occurrence of the event, and when the action should be executed relative to the satisfaction of the condition respectively.

In the ECA rule paradigm, an event occurs which is followed by the testing of condition. Based on the outcome of the condition evaluation, the action associated with the condition is executed. Whenever an event occurrence causes triggering of multiple rules at a time, scheduler provides a conflict resolution policy to order and execute the rules sequentially or concurrently. An ADBS should support both sequential and concurrent rule execution. Sequential execution is necessary when certain execution order is enforced by priorities or when rules have a predefined sequence of execution. Concurrent rule execution is important from the performance perspective of the system. Therefore, it is necessary to support a rule execution model that supports prioritized sequential and concurrent execution of rules.

In ADBS, rule execution model uses nested transaction model to support multiple rules. In the nested transaction model, the transactions are started inside the other transactions forming a transaction hierarchy. The transaction at the top of the hierarchy is called the top-level transaction and the other transactions are called subtransactions. Thus the two types of transactions that are used to handle rule execution in nested transaction model are:

- Subtransactions – When a rule is triggered by an event, a transaction is created and the rule is executed in that transaction. The rule’s coupling mode specifies the parent/child relationship between two transactions. If the rule’s coupling mode is immediate or deferred, the transaction is a
subtransaction of the transaction which generated the triggering event. The subtransaction is called a child transaction of the transaction containing the triggering event.

- Top-level transactions – Top-level transactions are transactions which have no subtransaction. When a rule is triggered by an event and if the coupling mode is detached, then the rule is executed in a top-level transaction.

As a result, in the nested transaction model, a triggered rule, where the coupling mode is separate is executed in a top-level transaction. Otherwise, rules triggered by an event are executed in the subtransactions of the triggering transaction. Thus, nested transaction model is used for modeling concurrent rule execution and it handles multiple triggering of rules that occurs when an event causes more than one rule to be fired. Multiple rule triggering is handled by executing the rules triggered by the same event as subtransactions of the transaction causing the event. In this research work, the rule scheduler uses nested transaction model for executing rules concurrently with the same priority. Concurrent execution of rules can also bring along conflict with respect to data access and the nested transaction synchronization scheme is used to handle the conflicts with locking protocol.

When multiple rules are triggered by an event, the rules are executed based on the coupling modes such as immediate, deferred and detached [Buchmann, 1999]. In the case of immediate coupling mode, when an event is detected, the triggering transaction (parent) is suspended immediately and the condition associated with the triggered event detected is executed. If the condition evaluates true, the action part of the rule is executed else the parent transaction continues. In the deferred coupling mode, when an event is detected, the parent transaction rule continues execution and the triggered rules are activated at the end of triggering transaction before it commits. In detached mode, the triggered transaction is executed as a separate transaction from the triggering transaction.
3.6 CONCURRENCY CONTROL IN NESTED TRANSACTION MODEL

Nested transaction model provides high degree of intra-transaction parallelism. The important aspect with respect to nested transaction model is the concurrency control. The objectives of synchronization among transactions are: (i) concurrency control among nested transactions has to prevent database updates performed by one transaction from interfering with database updates and retrievals from other transactions (ii) within a nested transaction concurrency control has to provide an effective and controlled concurrency between siblings.

The most common mechanism to achieve concurrency control is two-phase locking [Al-Jumah et al., 2000]. The two modes of synchronization used in locking are:

- Read – where multiple transactions share the object and none of them try to modify the object.

- Write – only one transaction can have exclusive control over the object and is not shared with any other transaction.

The two-phase locking algorithm presented in this research is based on the notion of nested concurrency control that follow closely with the one proposed by Moss [1985]. Any transaction can acquire a lock on an object in a specific mode (Read/Shared or Write/exclusive) and it holds the lock in the same mode until its termination (commit or abort) or until it explicitly upgrades the lock. Also a transaction can retain a lock held by a subtransaction when it commits by inheriting the lock. A retained lock is like a place holder, transaction that retains a lock actually has no access to the object it locks, all does is to ensure correctness with respect to acquiring locks. The locking algorithm for concurrency control is shown in Figure 3.4.
The rules stated above only allow upward inheritance of locks at commit time. A parent can therefore inherit the locks held by children only when they commit and an abort does not affect the inheritance of locks. The transaction calls used in this model are begin, commit and abort transaction. The following are the operations that take place when transaction calls are made to active ORDBMS:

- **Begin transaction:** This starts the transaction and any fetches of objects in the database within transaction has to notify the lock manager and the transaction is given the object based on the locking rules specified in the locking algorithm. Update of object is performed in process memory. If the transaction terminates normally with an explicit commit transaction, then modified objects are written back to the persistent store. If there is an abort transaction, none of the modified objects are written back to the persistent store. Each access to an object needs to consult with lock

---

**Figure 3.4 Locking algorithm for concurrency control**

Rule 1: Transaction T may acquire a lock in exclusive(X)-mode if no other transactions hold the lock in X-mode or in shared (S)-mode and all transactions that retain the lock in X or S-Mode is ancestors of T.

Rule 2: Transaction T may acquire a lock in S-mode if no other transactions hold the lock in X-mode and all the transactions that retain the lock in X-mode are ancestors of T.

Rule 3: When subtransaction T commits, the parent of T inherits T’s locks (both held and retained). After that the parent retains the locks in the same mode in which T held or retained them.

Rule 4: When a top-level transaction commits, it releases all the locks it holds or retains.

Rule 5: When a transaction aborts, it releases all locks it holds or retains. If any of its ancestors hold or retain any of these locks they continue to do so.
manager to decide whether it can get access to that object based on the locking rules.

- **Commit transaction**: The objects held in process memory that have been modified are written back to persistent store. Locks held by the transaction on that object are released and a check is done for any other transaction that is waiting to hold a lock on that object. The lock is given again based on the locking rules.

- **Abort transaction**: None of the objects held by that transaction are written back to the persistent store and all locks held by transactions are released.

### 3.7 Modeling of Rule Processing

The execution model of an ECA rule system specifies how the rules are processed by the database system at runtime. Based on the functioning of rule execution model, rule processing is modeled. In this work, rule execution model performs rule processing into two phases: (i) event detection and condition monitor – event detection and condition monitoring starts as soon as the rule manager receives the message, ‘raise event’ from an event detector. The rule manager process then retrieves the appropriate event from the database and determines the rules that are associated to this primitive event. The rules associated with the detected events are evaluated and the rule manager stores the rules whose conditions are evaluated to true in a conflict set for scheduling and execution. (ii) rule scheduling and execution – the rule scheduler selects and executes a rule from the conflict set according to the conflict resolution policy. Multiple rules triggered by the same event occurrence are executed according to their priorities and coupling mode.

In this research work, the model of an active database system is based on an object relational view of data. Database is a set of relational tables \{r_1, r_2, ..., r_m\} where each table r_i, for i= 1, ..., m is an ordered sequence of tuples < t_{i1}, ..., t_{ik} >. Tuple \( t =< v_1, ..., v_n > \) where \( v_i \) belongs to a domain of attribute \( a_i \) is the smallest unit of information processed by rule transactions. Data manipulation language is a general purpose algorithmic language with an embedded set of the following database commands.
Append command \(< A, t, r >\) appends tuple \(t\) at the end of the table \(r\).

Delete command \(< D, r, \phi >\) deletes from table \(r\) all tuples that satisfy condition \(\phi\).

Update command \(< U, a_1 = f_1(a_1, \ldots, a_n), \ldots, a_n = f_n(a_1, \ldots, a_n), r, \phi >\) updates the values of attributes \(a_1, \ldots, a_n\) from \(r\) in all tuples that satisfy condition \(\phi\).

Select command \(< S, r, \phi >\) selects all tuples from \(r\) that satisfy condition \(\phi\).

The abstract modeling of an active ORDBMS rule execution model shown in Figure 3.5 is constructed based on the active database abstract architecture in Figure 2.2. The proposed model consists of the four components such as Rule Manager Process, Rule Scheduler Process, Evaluator Process and Transaction Process. The rule manager component models the event detector and the condition evaluator. The rule scheduler component models the scheduler to order and execute the rules and the evaluator component models the query evaluator of the rule system. The transaction component models queries of transactions from applications to the database systems. The proposed model also includes the data stores such as current state of DB (database), past state of DB, Rule Base and Conflict Set. The current and paste state of the DB stores the current and past states of the database system. Rule Base maintains a set of rules defined in the system and Conflict Set maintains the current conflict set of rules. The processes read and write update the data stores.

In this model, each trigger rule takes the form: rule name: event IF condition Do action. The event-condition-action (ECA) rule is represented by \(< e, c, a >\) where ‘e’ represents an event, ‘c’ is a condition and ‘a’ is an action. Event is an attempt to execute one of the database commands by a database application or a rule. An event is described by its name and the two types of events are primitive and complex events. A primitive event is a set of database operations such as insert, delete, and update on a table in an object-relational database system. A complex event is a combination of primitive events using logical, arithmetic operators. The rules are triggered by primitive as well as complex events that occur during the execution of an application program. Condition is a query on the current database state and/or on the transition.
data, i.e. the data changes that caused the rule to become triggered. Condition $\phi$ is defined as a Boolean expression constructed over a schema of table $r$. Action is a sequence of database operations both read and write on the current database state.

Figure 3.5 Abstract model of an active ORDBMS rule execution model

An active rule ($r$) is defined by a triggering event ($r_e$), condition and an action. A rule instance of ‘$r$’ results from the triggering of ‘$r$’ due to the occurrence of an instance of $r_e$ and is represented by $<r, r_i, e_i>$ where $e_i$ denotes the instance of $r_e$ that triggered $r$ and $r_i$ is the instance identifier. Executing a rule instance $<r, r_i, e_i>$ consists in computing its condition and performing its action when the condition is evaluated to true. Every time an event arises, the rules triggered by this event are executed based on the function $<r, r_i, e_i>$.

The execution of set of rules is based on the notion of events and a rule instance execution is described by the state transition diagram depicted in Figure 3.6. There are four active states: triggered, evaluated, interrupted and wait. State triggered
is the initial state of transaction (T) where the task is waiting for the command begin_rule. The task represents rule instance triggering the rules. After this state, T enters in active state ‘evaluating’ to compute the condition, sends the message cond-eval with the result of the computation, then T enters in the wait state. At this state T depends on the result of the condition evaluation: if the condition has the value false, T enters in abandon state and executes the ending task. If the condition is true, T enters into the evaluated state, and waits for the begin_action command. T enters in the executing state after receiving the begin_action command and executes the action.

![State Transition Diagram](image)

**Figure 3.6 State transition diagram for rule instance execution**

Whenever an event trigger multiple rules it is necessary to support a rule execution model to handle multiple triggered rules. The rule scheduler is responsible for handling rule conflicts when multiple rules are triggered at the same time by an event. In the presence of multiple rules, the scheduler need to support prioritized serial execution of rules, concurrent execution of all rules or a combination of both. The rule execution semantics are expressed as a function, exeShd, which takes as input a database and a schedule. The schedule consists of a sequence of actions which are to be executed on the database. The occurrence of an event causes the triggering
of rules and that cause the execution of an action. The execution continues until the schedule becomes empty. This model assumes that for each type of event E detectable by the two system-defined database objects: has_occurred_E and change_E. The former is non-empty if event E occurred during the execution of last action, and it contains information about the occurrences of the event E. The change_E contains information about the changes that occurrences of event E made to user defined database objects during the execution of the last action.

For example, in an active object-relational database there can be for each user-defined relation R a set of six system-defined relations:

- occurred_insertion_R, which would be non-empty if one or more INSERT statements on relation R occurred during the execution of the last action.

- change_insertion_R, which would contain set of new tuples inserted into relation R during the execution of the last action.

- occurred_deletion_R, which would be non-empty if one or more DELETE statements on relation R occurred during the execution of the last action.

- change_deletion_R, which would contain set of tuples deleted from relation R during the execution of the last action.

- occurred_update_R, which would be non-empty if one or more UPDATE statements on relation R occurred during the execution of the last action.

- change_update_R, which would contain a set of pairs (old_tuple, new_tuple) for each tuple of R which has updated during the execution of the last action.

The event part of an ECA rule is either has_occurred_E or change_E, for certain event E. The identifiers has_occurred_E and change_E can also occur within the rule’s condition and action parts. The rule is said to be triggered if its event part is non-empty and both syntactic and semantic triggering of rules are supported in this work. Syntactic triggering happens if the rule’s event query is has_occurred_E and instance of event E occur and semantic triggering happens if the rule’s event query is
change_E and instance of event E occur and makes changes to the database. The condition part of the rule is a boolean-valued query. The rule’s event-condition query is the conjunction of its event and condition parts, where an event part is to be true if it is non-empty and false otherwise. The rule fires if it is triggered and its condition evaluates to true i.e. if its overall event-condition query evaluates to true.

A rule has list of one or more actions and it has also a coupling mode, which is either immediate or deferred. With immediate coupling mode, if the rule fires then its actions are prefixed to the current schedule, while with deferred coupling mode, they are suffixed. Multiple rules with the same coupling mode fire, then the actions of higher priority rules precede those of lower-priority ones on the schedule. Thus a total ordering of rules imposed on the set of ECA rules. The function Sched-rule determines the order with respect to the current database state based on the priority with coupling mode.

In this research work, ECA rules are identified by rule-id, if E is the event that triggers rule r, then has_occured_r returns the query has_occured_E, change_r returns the query change_E, condition r returns the rule’s condition query, action r: list of actions and mode r: coupling mode. The function triggers takes a rule action and returns a list containing the rules that may be triggered by that action based on the rule’s priority. This work supports rules that are triggered by primitive events generated by the execution of database modification operations such as insert, update and delete performed by user application. Rule execution takes place before or after the transaction generating the triggering event. The function Sched-rule determines the order to process trigger rule conflicts based on the user defined priority scheme and the function exeShd supports combination of both concurrent and sequential execution of rules.
3.8 SUMMARY

In this chapter the rule execution model of an active object-relational database systems for trigger rule processing has been described. The need for rule scheduling and execution of multiple rules for handling trigger rule conflicts has been discussed. The problem statement has been specified: i) the development of a rule scheduler for ordering rules based on user defined priority scheme and executing rules sequential, concurrent and the combination of both ii) defining the rule system for specifying the ECA rules for applications.

The importance of concurrency and the concurrent execution of multiple triggered rules with nested transaction model have been described. The abstract modeling of an active ORDBMS rule execution model has been described for rule processing. In this model, rules are represented in the form of ECA rule paradigm and each trigger rule takes the form: rule name: event IF condition Do action. The execution of set of rules is based on the notion of events. A rule instance execution with four active states such as triggered, evaluated, interrupted and wait has been described by the state transition diagram. The scheduler orders and executes the rules that are triggered by primitive events generated by the execution of database modification operations such as insert, update and delete performed by user application.